COMPUTER SCIENCES DEPARTMENT

University of Wisconsin - Madison

TRANSACTION MANAGEMENT FOR DESIGN DATABASES

Randy H. Katz
Shlomo Weiss

Computer Sciences Technical Report #496

February 1983

Transaction Management for Design Databases

Randy H. Katz and Shlomo Weiss[1]
Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706

ABSTRACT: We define a *design transaction* as a sequence of operations that map a consistent specification of an engineered artifact into a new consistent state or *version*. Because design transactions are unconventional, we argue that standard notions of consistency, atomicity, and durability are irrelevant for defining design transactions. We describe a design transaction management mechanism, based on version checkout and change files, that supports controlled sharing and is resilient to system crashes. The mechanism is well suited for a computing environment of engineering workstations and database service machines. It is being implemented as part of an engineering database management system project at the University of Wisconsin-Madison.

1. Introduction

Sophisticated database management techniques have been developed for "transaction processing environments," such as airline reservations, electronic funds transfer, etc., which are characterized by high volume, short duration, simple units of work. Database systems are now being employed in an increasing range of application environments with very different characteristics. These techniques are being extended to the new environments.

We are particularly interested in applying database methods to support design activities [KATZ82a]. A *design management system* handles the information about the design of complicated "engineered" artifacts. Examples of these include large software systems, multi-author documents, and integrated circuits. Database facilities are an important service provided by the system.

Complex engineered objects are designed by teams, simultaneously working on different portions of the overall design. A design management system must support the controlled sharing of design data, with mechanisms to insure that designers do

[1]Research supported by NSF Grant MCS-8201860.

2. Design Environment

The environment for design activity is substantially different than conventional transaction processing environments. These differences, and their effect on system requirements, are discussed in this section.

The debit/credit transaction described in [GRAY78] typifies conventional transaction processing. The database consists of a collection of bank account, teller cash drawer, bank branch balance, and account history records. Tellers handle customer deposits and withdrawals in real time. A typical transaction, many of which are executing simultaneously, is invoked at a terminal on behalf of a teller. It accesses a single account record, checks the balance to insure that there is sufficient funds if a withdrawal, modifies the balance, and makes the modified record available to other transactions. The teller cash drawer and branch balance records are also modified by the transaction. A history record is created to provide an audit trail of the transaction. Disastrous results ensue if more than one transaction is allowed to modify these records simultaneously, i.e., updates may be lost and the database may be left in an inconsistent state. A transaction processing system must guarantee high throughput and fast response, even when the size of the database is very large.

On the other hand, a typical "design transaction" behaves as follows (we will be more detailed in the next section). The transaction is invoked by a designer to extract a logical portion of the design from the shared design repository into his private workspace. During the lifetime of the transaction, he interacts with his data through an ensemble of application programs. These are sophisticated programs, and include: (1) editors to interactively manipulate the design data (e.g., an integrated circuit editor for mask layout), (2) generator programs to synthesize unspecified parts of the design from existing parts (e.g., a generator for programmable logic arrays), or (3) analysis programs to check for correctness of the design data (e.g., geometric design rule checkers).

When his design activities are complete, i.e., the designer believes that his data is once again consistent, he returns it to the shared repository to "release" it to other designers. Before he is allowed to do this, however, the data must pass a battery of tests for self-consistency. The validation process is complex, time consuming, and specific to the object being designed. The design system is responsible for insuring that all checks are performed in the desired sequence.

Once the design data has been shown to be self-consistent, it can be replaced in the design repository. However, older versions of a design file are rarely discarded once a new version has been created. Old versions may be needed (1) for legal purposes, (2) because they still describe a supported object installed in the field, or (3) to provide insights into the design process itself. The new version is added to those that are available on-line in the repository.

Conversational transactions, discussed in [GRAY78], superficially resemble our *design* transactions since both are non-atomic. However, there are a number of differences. *Conversational* transactions have conventional transactions as units of recovery ("nested transactions"), while design transactions support continuous saving of design changes and have the ability to recover past savepoints. With *conversational* transactions, the effects of a nested transaction can be undone only by a compensating transaction. Because *design* transactions support versions, it is possible to simply restore the modified files to their previous versions. The need for applications level consistency checking further distinguishes *design* transactions from *conversational* transactions.

Several observations are possible given these two transaction models:

(1) **Design Transactions Are Long Duration**

Designers interact with their data for long periods of time, i.e., days or weeks, while conventional transactions are of short duration, i.e., minutes at most. Thus, mechanisms that arbitrate exclusive access to shared data by forcing transactions to wait when it is not available are unsuitable in the design environment. Suspended transactions would be forced to wait for intolerably long periods of time. Real time access is critical in most transaction processing environments, yet designers are content to try again later to get the needed data. Further, conventional transactions spend most of their time in the database access routines, because the application logic is relatively simple. Design transactions spend most of their time in the associated "number crunching" applications programs. Thus they are not as closely coupled to the database system as conventional transactions.

Long duration transaction in the design environment do not have the problems of long-lived transactions described in [GRAY81]. For example, visible intermediate transaction states can be tolerated (i.e., "lower levels of consistency" are acceptable). While long running conventional transactions are usually aborted on system restart [GRAY81], long duration *design* transactions need not, and should not, be aborted (see Section 4).

(2) Design Transactions Touch Large Volumes of Data

The units of access in design transactions are large collections of related records, usually spanning several files. Conventional transactions are simple, and touch very few records. While design transactions spend a relatively small amount of time in the database system, the large volumes of data involved prohibit invoking the database system for access to individual records. [GUTT82] describes some of the performance problems. Therefore, design databases are shared repositories from which data must be extracted when needed for intensive access.

(3) Design Transactions Need More Than Serial Consistency

Correct execution of concurrent transactions has been defined in terms of serializability, i.e., the execution of concurrent transactions is consistent if their interleaved effect is the same as if they are run in some serial order. Serial consistency is unsuitable for determining whether design data is still "consistent" after update. Special validation programs must be invoked to verify the consistency of design data. Serializability theory describes when interleaved read/write accesses to shared data still obtain a "correct" (i.e., serial consistent) result. Interleaved updates are undesirable in the design environment, since it is meaningless for two designers to change the same portion of a design.

(4) Design Transactions Are Not All Or Nothing

Conventional transactions are atomic: either all updates made by a transaction become visible (it commits) or none are visible (it aborts). Intermediate states are invisible to concurrent transactions, even if the system should crash during a transaction. Recovery mechanisms insure that the database is restored to a transaction consistent state. In the design environment, as much work as possible should be recovered in a crash, even past checkpointed states if possible. Intermediate states, as long as they are file system consistent, are acceptable.

(5) Design Transactions Are Not Ad Hoc

Since designers know in advance what portions of the design they will be working on, all needed resources can be acquired at the beginning of the transaction. Because of the interactive nature of design transactions, deadlock is intolerable and must be avoided through preallocation. Deadlock detection mechanisms that abort in-progress transactions are undesirable, since valuable design work would then have to be undone.

Design transactions do not fit the conventional notions of consistency, atomicity, and durability upon which database transaction management has been built.

Serial consistency is insufficient for determining the self-consistency of design data. The data itself, rather than the order in which it is accessed, determines the correctness of a transaction. Further, simultaneous access to the same design data is unlikely as well as undesirable.

Design transactions are not atomic in the sense of conventional transactions. Visibility of intermediate states of design data may even be desirable. While only one designer is allowed to update the data, many can be reading ("browsing") it simultaneously. Others may want to check on an in-progress portion of a design. Since they are browsing the design, a lower level of consistency can be tolerated, i.e., records changing underneath them.

Designers demand that as much of their work as possible be saved in the event of a system failure. Savepoints guarantee to save changes, but it is desirable to be able to bring the database back to its *latest* possible state. While returning to a checkpointed state should be supported, we believe that system generated undo will be rare. This is a major departure from the work described in [LORI82], where the design database is automatically returned to its last checkpointed state in the event of a crash.

The durability of design transactions is also different from conventional transactions. Old versions of design data persist even after newer consistent versions have been created. Support for versions is already needed in the design environment, and can be integrated with transaction management to simplify many aspects of concurrent access and recovery. Transaction management components of existing database systems do not support versions, making these somewhat unsuitable as a starting point for design transaction support.

3. Design Transactions

3.1. Introduction

A transaction is a unit of work that maintains the consistency of a database. A set of constraints are in force at transaction begin and end, but can be violated during the transaction. In the design environment, the definition of data self-consistency is much more complex than that found in transaction processing environments. For example, an airplane design is consistent only if the airplane can still fly with its redesigned wing. In general, these constraints can only be guaranteed by invoking complicated checking programs.

Therefore, we define a *design transaction* as a sequence of database operations that map a consistent version of a design into a new consistent version. Design transactions are non-atomic units of design consistency. If the system crashes, then the designer can continue from the last safe state determined by transaction management (which may be beyond the last saved state). Old design versions are durable across transactions, i.e., an old version is not removed unless it is explicitly moved off-line.

3.2. Concurrency Control Issues

Design data is arranged so that logically related parts of the design can be accessed as a single object. Objects can be nested within other objects, forming a hierarchy of design data. For example, a **microprocessor** consists of a **data path** and a **control unit**; the **data path** consists of a **register file, shifter,** and **arithmetic logic unit**; etc. The nested structure of design data has been called a design hierarchy in [KATZ82a] and complex objects in [HASK82a][2]. We adopt the former terminology. A designer can request access to the microprocessor (the whole design), the data path (a subpart of the whole), or the register file, etc. Designers can work in parallel as long as they are in non-overlapping subtrees of the hierarchy.

[2] Note that in [HASK82a], complex objects are single level. A complex object cannot be contained within another complex object.

Multiple designers do not work on the same related pieces of the design at the same time. Changes made by one designer might conflict with those of another. Even if these changes do not overlap, merging the sets of changes together might not result as intended, since the changes may not fit together. Therefore, the appropriate unit of exclusive access is a design subhierarchy, representing a logical portion of the design, and identified by its root.

Conventional locking is not appropriate for design data. If a design subtree is unavailable because it has been acquired by another designer, then the requesting designer should not be forced to wait. Also, locks need to survive system crashes. The solution proposed in [HASK82a] is to introduce *persistent locks* that are stored in non-volatile storage, i.e., the database, to survive crashes. These can be requested without blocking if not available.

Note that the lock table is normally placed in memory for reasons of efficiency. Design transactions are less time critical and set less locks than conventional transactions. Placing the lock table on disk to insure its durability is worthwhile in the design environment, even though it is more expensive to set/reset locks.

A more appropriate paradigm for acquiring design data is to view the design database as a library [KATZ82d].[3] Design subparts are checked out to designers, who return them when done. "Concurrency control" is therefore handled by a Librarian process that traverses and manipulates the hierarchical structure of design data. It knows: (1) *what* parts of the design have already been checked out, (2) *who* has checked them out, and (3) *when* they are expected to be returned to the repository. This information is stored in the design database, and is thus durable across system crashes. A designer who must have access to data can determine who currently holds it, enabling him to request its early return.

[3]One difference with a conventional library is that books cannot be changed, but design data can.

When a designer is through with the data, it is checked back into the repository as a *new* version. The modified design data cannot become the current version ("released"), until it has passed the necessary self-consistency checks. It can be accessed by other designers who are willing to browse the possibly inconsistent data (it may change underneath them). A transaction cannot complete, and cannot make its new versions available, until the data is shown to be consistent. At commit the *in-progress* versions of all data checked out to the transactions become *current* and the previously current versions become the next *previous* versions.

The facilities provided by the Librarian more closely resemble the time-domain addressing scheme of [REED79] than a conventional lock manager. A version-based approach is well suited to the design environment, since versions of design data must already be supported (see [KATZ82b] for a discussion of version support for design databases). Our versions are different than Reed's in that design versions are created at design consistent points, rather than for individual update operations. The version-based approach simplifies many of the difficulties of concurrent access in the design environment. Designers can browse the last consistent version of the design even while the a new version is in-progress. If desired, the in-progress version can be browsed, although reads may not be repeatable. Even if they are, the data read may yield inconsistencies, since it is not yet guaranteed to be self-consistent. Browsers can continue to read the same version even when the in-progress version becomes current in the middle of browsing.

3.3. Recovery Issues

Because of the value of. design data, resiliency to system crashes is an important requirement of the database component. Conventional transactions restore the database to a transaction consistent state in the event of a crash. Design transactions are not atomic in this sense. If the system crashes in the midst of a design transaction, it may be undesirable to undo *any* work done by the transaction. Obviously, file system actions, such as page writes, must be atomic, and it must be possible to reconstruct the file system to a consistent state (from its viewpoint) after a crash.

Savepoints within a design transaction guarantee that the database can be restored to that point. However, it will be desirable to recover past a savepoint. Note that this is not possible in [LORI82].

In addition to the demands for a continuous recovery capability, the computing environment for design introduces additional problems. With the development of inexpensive engineering workstations, i.e., a graphics display, processor, hard disk, and network interface within a single package, the design environment will consist of individual workstations networked to a file server. The workstations are inherently less resilient to crashes than the file servers, because they are located in a more hostile environment (an office instead of a machine room), and because it is more difficult to use redundancy to obtain resiliency (workstations rarely have more than one disk, and almost never have tape drives). The result is that the redundancy for data on the workstation must be provided by the file server.

The recovery system is complicated by the four different kinds of failures: (1) workstation soft crash (memory buffers lost), (2) workstation hard crash (local disk data lost), (3) server soft crash, and (4) server hard crash (see [BROW81, KARS82] for descriptions of conventional transaction processing in this environment).

However, the nature of the design environment simplifies many recovery aspects. While design versions are checked out to workstations, the last consistent version resides safely on the file server. Versions are used to avoid updating in place, with its associated undo complexities. Only very limited undo capabilities are needed, for example, undo the *last* update. Transaction logs, or *change files* have a simple structure and are associated with files rather than transactions, since only one transaction can update a design file at a time.

4. Design Transaction Management

4.1. Computing Environment

The computing environment for the design transaction manager is the following. Design applications are run at workstations, which are connected via a high speed network to a shared file server. While the workstations have simple I/O configurations (i.e., a single hard disk), the file server has many devices and controllers.

Design transactions consist of four phases: *file acquisition*, *work*, *validation*, and *completion*. Work and validation must follow file acquisition, but can be intermixed with each other. Validation must be complete before a design transaction is allowed to enter completion.

A design transaction begins with a *file acquisition* phase. Design transactions are closely associated with the files they touch. When the transaction completes, it creates new current versions of its acquired files. This cannot be done until the collection is shown to be self-consistent.

A designer requests a design file (or subhierarchy of files) from the file server Librarian. If the files are still available, i.e., have not yet been checked out, then the request is granted, and the files are transfered to the workstation's disk.[4] Additional "mirrored" copies are also made in the file server, to protect the designer from local crashes. These will be used to track committed or saved changes made to the data in the workstation.

If the files have already been acquired by others, they are identified, as is the expected time of return. The designer continues with other work, and must try to acquire the files again later. All needed files must be acquired at transaction begin to avoid deadlock problems.

[4]Actually, the files are copied as needed, rather than all at once.

Once the needed files have been checked out to the workstation, the transaction can enter its *work* phase. The designer manipulates the design with the aid of design tools. Savepoints protect the transaction from loss of data due to local crashes. Changes made to files since the last savepoint are maintained in local change files. At savepoints, these are transmitted to the file server, where they are committed to the mirrored copies of the files, making the workstation and file server copies identical. It is possible to recover past a savepoint if the local logs have survived the crash, or are also saved on the file server.

Thus, a design transaction is punctuated with savepoints, insuring that changes so far will survive local failures. Activity at the workstation can continue even though the connection to the file server is broken (fortunately rare!). This is not advisable however, since it exposes the designer to serious loss of data in the event of a local crash.

A beneficial side-effect of the extra redundancy in storing mirrored copies on the file server is that browsing in-progress data is simplified. A relatively recent, savepoint consistent version of the design can be viewed without needing to access the data stored at the remote workstation.

When design work is completed, the transaction enters a *validation* phase. Verification programs are invoked to check that the modified design data is self-consistent. If the data is not valid, this does not cause the transaction to be aborted![5] The inconsistencies have to be located and corrected, (the transaction reenters the work phase), and validation must be retried. Self-consistency checking may actually be distributed throughout the lifetime of the transaction, and need not only occur at the very end of design activity. However, the system must insure that the design data has not been changed since it was last validated.

[5]This has nothing to do with optimistic concurrency control validation [KUNG81]!

Once shown to be valid, the transaction can enter a *completion* phase. The in-progress versions of the design files are made current, the old current versions become previous, etc., and the transaction terminates successfully. The new design file versions can now be checked out to other designers. If a designer decides to abort his transaction, then the global and local copies of files are destroyed, and the original versions are made available again for checkout. The version-based approach greatly simplifies undo processing.

From the above discussion, it is obvious that design transactions make a heavy demand on disk resources. However, redundancy is unavoidable if resiliency to crashes is to be obtained. Since the file server is dedicated to providing file services to the network of workstations, it can be equiped with a large number of disk devices.

4.2. Design Librarian

The Librarian is the component of design transaction management that coordinates all access to shared design data. It manages *designs*, which are named hierarchical collections of design files. A design is analogous to a file system directory, in which internal nodes of the design hierarchy are subdirectories, and leaf nodes are ordinary files. Subtrees of the hierarchy are the units of access, and are identified by their roots.

The Librarian is responsible for creating and manipulating the designs and for checking out design data to workstations. It is structured as a server process on the file server machine. Request for design file services are directed to it (see figure 4.1).

The process by which a design portion is checked out is the following. A designer identifies the portion of interest by giving the name of the root of the subtree. The Librarian traces a path from the root of the design to the root of the subtree. At each node along this path, control information is examined to insure that the

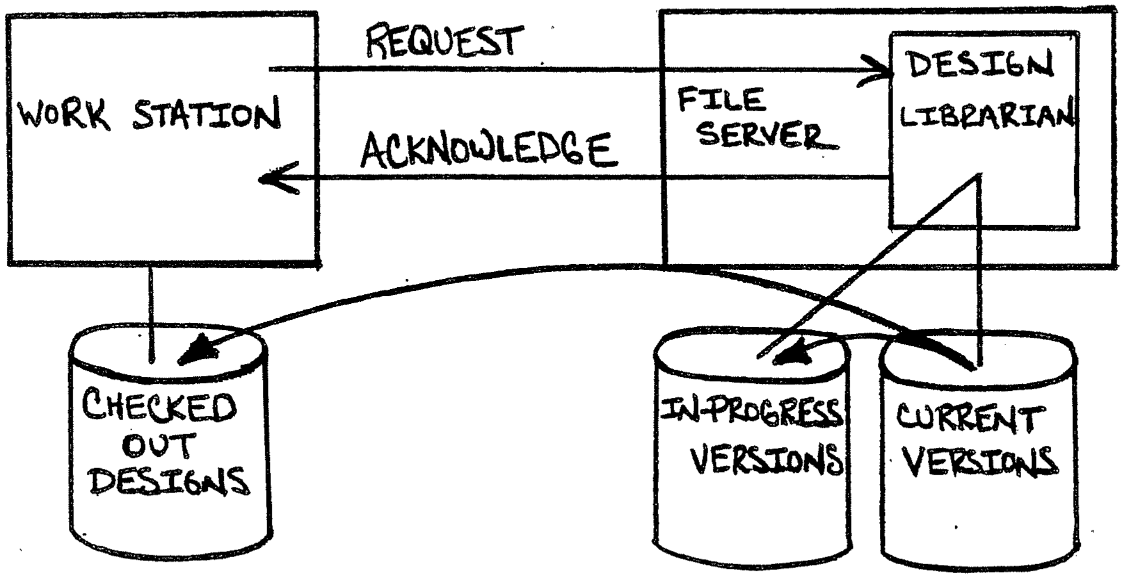

Figure 4.1 — Design Check-Out

data has not been checked out to another designer.

A hierarchical locking protocol is employed to arbitrate access to the design hierarchy. The protocol insures that no more than one designer has acquired a portion of the design for exclusive access with the intention of updating it, and thus creating a new version. The intention to acquire a subtree for exclusive access is recorded in every node along the path. If a node is already held for exclusive access by another designer, then an update request cannot be satisfied and is aborted. This insures that the requested subtree is not contained within a subtree already checked out. The designer must be able to acquire the root of the subtree with exclusive access. If another designer has acquired it with the intention of updating one of its subtrees, then the access cannot be granted. This insures that there is no checked out subtree contained within the subtree being acquired by the designer (see figure 4.2). Read requests never cause problems because all versions except

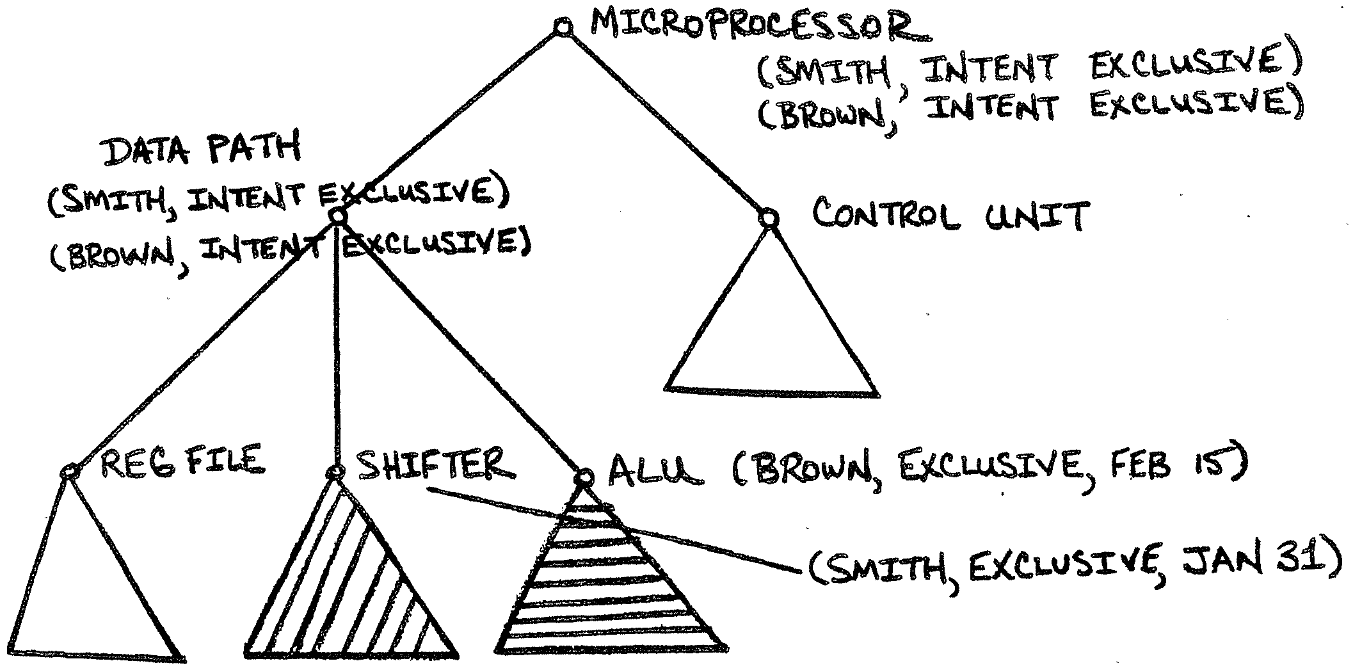
MICROPROCESSOR
(SMITH, INTENT EXCLUSIVE)
(BROWN, INTENT EXCLUSIVE)
DATA PATH
(SMITH, INTENT EXCLUSIVE)
(BROWN, INTENT EXCLUSIVE)
CONTROL UNIT
REG FILE
SHIFTER
ALU (BROWN, EXCLUSIVE, FEB 15)
(SMITH, EXCLUSIVE, JAN 31)

Figure 4.2 -- Controlled Sharing in a Design Hierarchy

the one actually in-progress are read-only. The hierarchical locking protocol for variable granularity locks has been described in [GRAY78].

If the data can be acquired, the subtree is copied to the workstation and another copy is made in the file server. The location of checked out and mirrored copies of files is recorded in the current copy of the design hierarchy, and is associated with individual nodes within the hierarchy.

When data is checked back into the repository after validation, the back-up copies are made *current* and the old current copies are made *previous*.

4.3. Recovery Manager

The recovery manager is responsible for insuring that as much data as possible survives a system crash. This is accomplished by keeping copies of the data in multiple places with different failure modes. Each checked out design file has five files associated with it: (1) the *local working file*, stored at the workstation, (2) the *local change file* stored at the workstation and holding a log of changes since the last savepoint, (3) the *global working file* stored in the file server, (4) the *global change file* which holds changes transmitted by the workstation but not yet saved, and (5) the *redo log* which contains all saved changes to the file since the in-progress version was first created by extraction from the current file version (see figure 4.3). The change files are differential files. The local and global change files record the differences between the local file and its global copy. The difference between the global working file and the current file is recorded in the the redo log.

The recovery manager supports *savepoints*. A save causes data and change file buffers to be forced to disk by the local buffer manager. The local change file is copied to the file server and is appended to the global change file. Note that only the changes are written back to the file server, not the complete file. A background process copies the local change file entries to the global change file, providing a *continuous save* capability. This guards against data lose in the event of a local hard

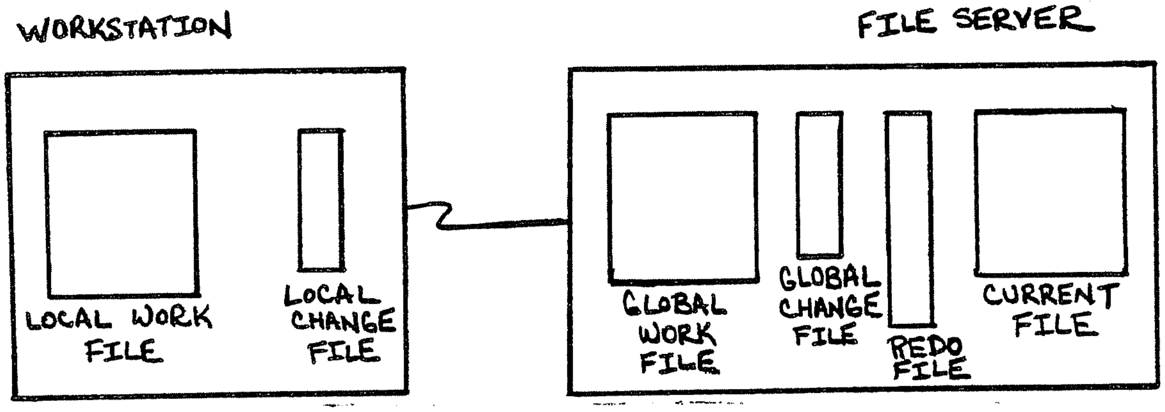

Figure 4.3 -- Files Associated with a Checked-out File

crash, and reduces the latency of a save. Space will be tight on the local disk, and can be reclaimed as local change file entries are copied to the file server. The global change file is then merged with the global working file to bring it up-to-date, and the global change file is appended to the redo log.

The merge operation can be implemented as an atomic action by performing the merge into a copy of the global working file. The original file is deleted when the merge completes. Save actions can be overlapped with continued work on the workstation, unless a blocking savepoint is requested.

An alternative strategy requires a modification of the file server's file system. When data pages are displaced from memory buffers to local disk, they can also be written through to the file server over the network. A shadow page approach [LORI77] is used to replace original pages without overwriting them. The original and updated versions of the file are described by page maps with unupdated pages in common. At a savepoint, the original version is replaced by the updated version. We have not selected this approach because it would require implementing a file system on the server that supports shadow pages.

In the event of a soft workstation crash, i.e., memory buffers are lost, a number of options are available. The last savepoint can be reconstructed by copying the file server copy back to the workstation. A more up-to-date version can be reconstructed by merging the local and global change files into the local copy. Some updates will be lost, since they were only recorded in the buffers, but others that were displaced to stable storage since the last save will be able to be committed. Note that the design file can be restored to a state beyond the last savepoint.

A hard workstation crash loses data on the workstation's disk. If we assume that both local changes and local data have been lost, recovery can proceed as above. The file can be restored to its last saved state by copying the global working file back to the workstation. If the merged global working file and global change file are copied back, then the file is restored to the last update known by the file server.

The file server can employ more conventional techniques to insure that its files are durable. Archival dumps of working and current files are taken frequently. The file merge and append operations are implemented to insure that they are idempotent in the event of a soft crash at the file server. Hard crashes are dealt with by restoring the global working files to their archived versions and reapplying the global redo logs. This brings the file back to its last saved state. The global redo file should be duplexed to insure recovery in the event of a hard crash in the file server.

4.4. Design Data Validation

The need for applications-level consistency is a new aspect of design transactions. [NOON82, EAST81] describe mechanisms for maintaining the consistency of design data, either by keeping track of which validation tools have been applied, or by modeling the transaction as a flow of data among applications and verification programs. Ways to specify the relationships between design data and design checkers are still under intensive study.

5. Design Transactions and Design Systems

A design system is more than a data management component. Facilities for user interface, design tools, and design integrity management are also necessary. In this section, we describe how design management integrates with the other components of the design environment.

A design system is structured into four different levels (see figure 5.1). The innermost level is concerned with storing design data on disk, and is called the *database component*. The database component provides for shared, reliable storage of design data. This could be implemented with a database system, but many of the facilities provided by modern database systems, such as high level query languages, are not strictly needed. The abundance of services provided by a general-purpose

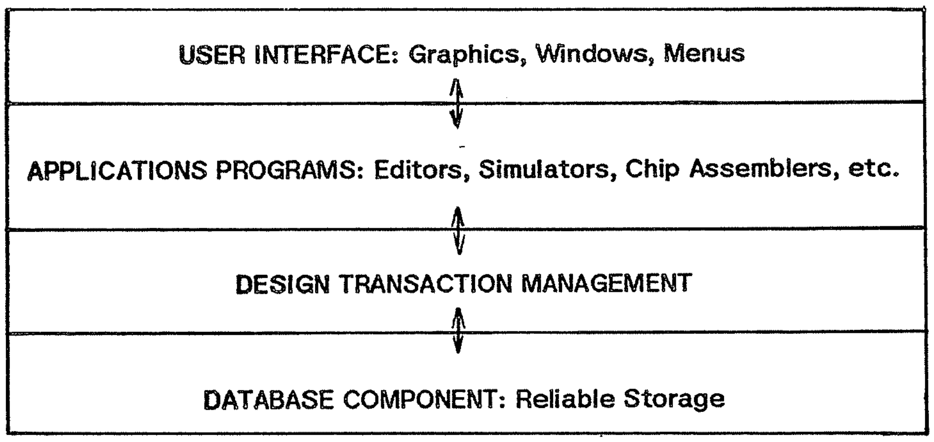
USER INTERFACE: Graphics, Windows, Menus
APPLICATIONS PROGRAMS: Editors, Simulators, Chip Assemblers, etc.
DESIGN TRANSACTION MANAGEMENT
DATABASE COMPONENT: Reliable Storage

Figure 5.1 -- Design System Architecture

database system may hinder performance, especially since most design data is accessed in bulk. The key facility is a file system that is resilient to system crashes and that provides high performance for sequential accesses.

The outermost component provides a package of services that communicate with users at workstations. We call this the *user interface* component. Included here are general packages for managing screens, windows, and menus. The existence of such software greatly simplifies the creation of new design tools.

Application programs reside between the user interface modules and the database component. In the design environment, these applications help to create, manipulate, view, and validate the correctness of design data. In a conventional system, a large portion of these programs are concerned with user interaction and data storage. These services are provided by the respective components of the design environment, simplifying the construction of new design tools.

The design transaction management component described in this paper resides between the applications and the database component. It is responsible for mediating the access to the design data, through the Design Librarian. It is also responsible for guaranteeing that design changes can survive system crashes, by providing a savepoint facility in conjunction with the resilient file system. Finally, it is responsible for maintaining the self-consistency of design data, by providing a design validation facility. This facility provides a framework in which design validation can occur; it does not include the particular validation tools that have to be used to check the correctness of a design. The latter are applications programs.

Just as interface and data storage services are viewed as being generally useful packages suitable for implementation as independent components, so to is design transaction management. Very few systems available today automatically or semi-automatically support the design validation process. This a key concept behind the notion of design transactions, and is an open topic for research.

6. Conclusions and Status

In this paper, we have described a new transaction processing environment that is significantly different from that in which the standard notions of consistency, atomicity, and durability have evolved. The experience gained in building these latter systems is not necessarily appropriate for the design environment. We believe that more effective systems can be built by implementing transaction management with new techniques.

We have described the architecture of a design transaction management system. It is part of an overall design environment being constructed at the University of Wisconsin-Madison. Currently, the database storage component has been implemented. The design of the transaction manager, as documented in this paper, is now complete, and implementation is underway. The first tool we intend to build is a chip assembly system for the manual hierarchical composition of design data across its representations, and the maintenance of consistency across these representations [KATZ82c]. The tool will be interfaced with shared data through the transaction manager.

7. References

[BROW81] Brown, M. R., R. Cattell, N. Suzuki, "The Cedar DBMS: A Preliminary Report," Proc. ACM SIGMOD Conference, Ann Arbor, Mi., (May 1981), pp. 205 -- 211.

[EAST81] Eastman, C. M., "Database Facilities for Engineering Design," Proc. IEEE, V 69, N 10, (October 1981), pp. 1249 -- 1263.

[GRAY78] Gray, J., "Notes on Data Base Operating Systems," IBM San Jose Research Report# RJ2188(30001), (February 1978).

[GRAY81] Gray, J., "The Transaction Concept: Virtues and Limitations," Proc. 7th Intl. Conf. on Very Large Databases, Cannes, France, (October 1981), pp. 144 -- 154.

[GUTT82] Guttman, A., M. Stonebraker, "Using a Relational Database Management System for Computer Aided Design Data," IEEE Database Engineering Newsletter, V 5, N 2, (June 1982), pp. 21 -- 28.

[HASK82a] Haskin, R. L., R. A. Lorie, "On Extending the Functions of a Relational Database System," Proc. ACM SIGMOD Conference, Orlando, Fl., (June 1982), pp. 207 -- 212.

[HASK82b] Haskin, R. L., R. A. Lorie, "Using a Relational Database System for Circuit Design," IEEE Database Engineering Newsletter, V 5, N 2, (June 1982), pp. 10 -- 14.

[KARS82] Karszt, J., H. Kuss, G. Lausen, "Optimistic Concurrency Control and Recovery in a Multi-Personal Computer System," ACM SIGSMALL Newsletter, V 8, N 4, (November 1982), pp. 12 -- 21.

[KATZ82a] Katz, R. H., "A Database Approach for Managing VLSI Design Data," Proc. 19th ACM/IEEE Design Automation Conference, Las Vegas, Nv., (June 1982).

[KATZ82b] Katz, R. H., T. J. Lehman, "Storage Structures for Versions and Alternatives," University of Wisconsin-Madison Computer Sciences Technical Report #479, (July 1982). Submitted to IEEE Transactions on Software Engineering.

[KATZ82c] Katz, R. H., S. Weiss, "Chip Assemblers: Concepts and Capabilities," University of Wisconsin-Madison Computer Sciences Technical Report #486, (November 1982). Submitted to 20th ACM/IEEE Design Automation Conference, Miami, Fl., (June 1983).

[KATZ82d] Katz, R. H., "DAVID: Design Aids for VLSI using Integrated Databases," IEEE Database Engineering Newsletter, V 5, N 2, (June 1982), pp. 29 -- 32.

[KUNG81] Kung, H. T., J. T. Robinson, "On Optimistic Methods for Concurrency Control," ACM Trans. on Database Systems, V 6, N 2, (June 1981).

[LORI77] Lorie, R. A., "Physical Integrity in a Large Segmented Database, " ACM Trans. on Database Systems, V 2, N 1, (March 1977), pp. 91 -- 104.

[LORI82] Lorie, R. A., W. Plouffe, "Complex Objects and Their Use in Design Transactions," IBM Research Report RJ 3706 (42922), (December 1982).

[NOON82] Noon, W. A., K. N. Robbins, M. T. Roberts, "A Design System Approach to Data Integrity," 19th ACM/IEEE Design Automation Conference, Las Vegas, Nv., (June 1981).

[REED79] Reed, D., "Implementing Atomic Actions on Decentralized Data," Proc. 7th ACM SIGOPS Symp. on Operating Systems Principles, 1979.