A MODEL OF CAD TRANSACTIONS

Francois Bancilhon[1], Won Kim[1] and
Henry F. Korth

Department of Computer Sciences
University of Texas at Austin
Austin, Texas, 78712

TR-85-06 April 1985

---

[1]Microelectronics and Computer Technology Corporation, 9430 Research Blvd., Austin, Texas.

A Model of CAD Transactions

François Bancilhon, Won Kim
Microelectronics and Computer Technology Corporation
9430 Research Blvd.
Austin, Texas 78759

Henry F. Korth
Dept. of Computer Sciences
University of Texas
Austin, Texas 78712-1188

ABSTRACT

A CAD environment requires a significantly different model of transaction from that developed for typical data-processing applications. A viable model of CAD transaction must allow a group of cooperating designers to arrive at a design without being forced to wait over a long duration. It must also allow a group of designers to collaborate on a design with another group by assigning subtasks. In this paper, we provide both an intuitive description and a formal development of a new model of CAD transactions.

1. Introduction

The model of transactions as popularized by Jim Gray [GRAY78] takes the view that a transaction is an atomic sequence of reads and writes against a database. In other words, when a transaction successfully finishes (commits), all its updates are permanently recorded in the database; and when the transaction fails, the effect of its updates up to the point of failure is completely erased from the database. To support this atomicity property of transactions, a database system must do two things. First, when a transaction updates a piece of data, it must be prevented from seeing changes made to data by other concurrently executing transactions. Second, when a transaction fails, due to a system crash, deadlock, or human intervention, the system must back out all updates the transaction has made, so as to return the database to the state it was in before the transaction started.

This model of transactions has served quite well in conventional data-processing applications, where transactions typically terminate within a few seconds. When a transaction is short-lived, it is not unreasonable to prevent access to data being updated by a transaction by forcing other transactions to wait, since the wait will be of short duration. It is also not unreasonable, if a transaction fails, to back out all its updates, since relatively little work will be lost.

Unfortunately, this model is not applicable to such engineering-oriented applications as CAD and software development. First, for performance reasons, the system configuration envisioned for design systems consists of a public system and a collection of private systems connected to the public system [HASK82, KATZ83, LORI83, KIM84]. The public system manages the public database of stable design data and design control data. A private system manages the private database of a designer on a design workstation. A CAD transaction, initiated on a private system, consists of

- checking out of design data from the public system and their insertion into the private database,
- reading and writing of both the private database and the public database, and
- checking in of updated design data to the public database.

Second, as the above paragraph implies, transactions in CAD applications span considerably longer durations than their counterparts in typical data-processing applications. Since CAD transactions are of long duration and involve conscious decisions of human designers, it is highly undesirable to force transactions to wait until other transactions with conflicting data access terminate, or to have the system back out all updates to the database if a transaction fails.

Early proposals attempted to define a model of transactions that reflects these design system requirements [KATZ83, LORI83]. These proposals define a CAD transaction as a sequence of conventional transactions, involving the public system and one private system. When a CAD transaction fails, the databases (public and private) are returned to the state they were in before the current short-duration transaction started, but not to the state before the long-duration transac-

tion. These models suffer from some important shortcomings.

- They fall far short of promoting a cooperative design environment in which a group of designers may exchange incomplete, yet stable, design objects. They imply that interaction among designers takes place only through explicit checkouts and checkins via the public system. For example, if designer A wishes to pass an incomplete object to designer B, he must first check the object into the public database, destroying an older copy. When designer B returns a more complete version of the object, he in turn must check it into the public database, destroying the copy designer A had checked in.

  Further, if designer A wishes designer B to update an object X, designer B will need, in general, other objects, say Y and Z, that designer A owns. As a result, designer B is forced explicitly to check out objects Y and Z, rather than simply issuing queries against the objects.

- They do not address properly the requirement for a *coordinated commit/abort* of the changes to both the public database and the private database that individual short-duration transactions have made.

  In other words, the models ignore the fact that a designer may issue queries and updates against not only the private database but also the public database, within a short-duration transaction. Further, they assume that such operations as checkout and checkin are local to a private database, when in fact, the effect of the operations is recorded in both databases.

The model of CAD transactions described in [KIM84] is an attempt to remove these deficiencies in the earlier models of CAD transactions. It supports the following notions:

- A CAD transaction has a *semi-public database,* into which it may place design objects it has updated. Once a transaction places an object in its semi-public database, other authorized transactions may check it out. A transaction that checks objects out of another transaction's semi-public database becomes a *child transaction* of that transaction. The

public system actually manages both the public database and semi-public databases of all CAD transactions. These notions of semi-public databases and transactions forming a hierarchy were introduced to support the exchange of incomplete design objects without explicitly involving the public system.

- A CAD transaction may access both the private and public databases. This means that when a short-duration transaction commits or aborts, changes to both databases must be synchronously (atomically) committed or aborted.

The model does not completely succeed in resolving the difficulties in the earlier proposals. In particular, it suffers from the following, rather serious drawbacks.

- The model does not establish properly the notion of *database consistency* that it satisfies. Although it appears that level 3 consistency is preserved across transaction hierarchies, only level 2 consistency appears to be satisfied among transactions within a transaction hierarchy.

  There are two reasons for this difficulty. One is the fact that the model allows a transaction to check in objects at any point during its execution, rather than only at the end of the transaction. Another is that the model resorts to long-duration locking for design objects that are checked out, but enforces locking for the duration of short-duration transactions for other data that are directly queried or updated.

- The notion that a CAD transaction is a *sequence* of short-duration transactions is not necessarily valid, as it implies a total ordering of short transactions. As we will show in the next section, a more flexible approach is to view a CAD transaction as a set of short-duration transactions, with some of the short transactions running in parallel.

- The model does not resolve fully the problem of destroying the old copies of objects being checked in on behalf of child transactions. When a transaction checks in an object to its parent, the original copy of the object residing in the semi-public database of the parent transaction is replaced by the copy the child checks in.

The new model we propose in this paper removes the first two of these difficulties. The third can be removed by an implementation of our model that links the checkout and checkin of objects to the creation and deletion of versions. However, this is part of the implementation issues, and we will postpone a detailed treatment of them to a sequel to this paper.

2. Intuitive Model of CAD Transactions

In this section we provide an intuitive description of our model of CAD transactions. We have arrived at the model from several observations about the way in which large design projects are carried out. We lump together design, engineering, and software development projects under the term *design projects*. Our contribution is in the development of a model that incorporates all the requirements that these observations lead to, in a way that admits a notion of database consistency.

The first observation is that a large design effort is typically partitioned into a number of projects. This induces a partitioning of the database into a number of mostly non-overlapping partitions, consisting of design data and some data about design data, for each project. There are classes of data, such as the database directory (catalog), to which different projects tend to require shared access.

We interpret this observation to mean that it will be acceptable to force *project transactions* that attempt to access another project's database partition to wait until that project finishes. This will, in general, be a long-duration wait. In other words, conventional concurrency control protocols (e.g., two-phase locking) will be acceptable for enforcing database consistency among projects.

Our second observation is that most of the workstations commercially available today support *windows* (or other comparable facilities) and that future workstations will continue to support them. A designer may create and manipulate multiple windows, executing multiple tasks concurrently.

Thus, a designer's transaction is better modeled as a *set*, rather than a *sequence* of short-duration transactions as has been proposed in [KATZ83, LORI83, KIM84]. It will, in general, form a hierarchy, where the child transactions of the transaction at a given node can be executed in parallel. Further, the sequence of transactions initiated from the same window will imply a user-defined ordering of transactions. An example designer's transaction is illustrated in Fig. 1, where each node of the graph is a short-duration transaction, and the set of edges defines the ordering of transactions.

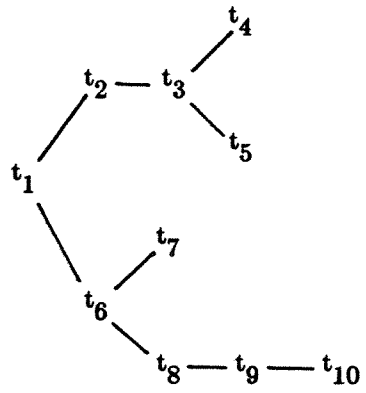


Fig. 1 An example designer's transaction

The third observation is that each project has a number of designers who further subdivide the project into a number of subtasks. As the designers work on a well-defined, fairly small subtask, there is a far greater need for shared access to the project's database, than that among projects.

This implies that a designer should not be forced to wait until other designers within the same project complete their subtasks. This leads to the notion of cooperating transactions. A *cooperating transaction* is a set of designer's transactions. As mentioned earlier, each designer's transaction consists of a set of conventional, short-duration transactions issued from the windows of the designer's workstation. As such, each designer's short-duration transaction needs to wait only until the current, conflicting short-duration transactions of other designers' transactions complete. Therefore, from the point of database consistency, it is immaterial how many designers

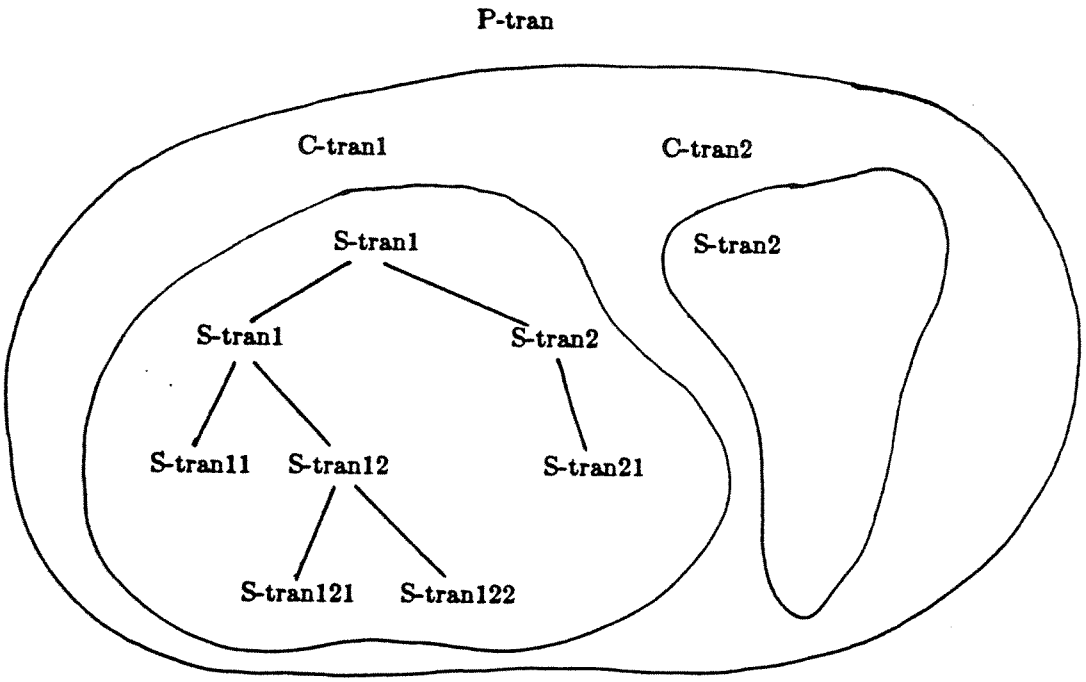
participate in the same cooperating transaction; all the short-duration transactions of all the designers' transactions are to be treated just as if they were issued by a single designer.

The fourth, and final, observation is that in a complex design project, it is often the case that some tasks are subcontracted to other (groups of) designers. The client specifies tasks to be completed, and grants the subcontractor limited access to the client's database. For example, a client may allow a subcontractor to access a file describing the interface (i.e., I/O pins and functional specification) of a circuit, and specify that the subcontractor provide the implementation details of the circuit.

This observation leads us to the notion of client/subcontractor transactions. A ***client/subcontractor transaction*** is a cooperating transaction which exists solely to work on behalf of another cooperating transaction. It is easy to envision a hierarchy of subcontractor transactions spawned by a single client, itself a cooperating transaction. An intermediate node on the hierarchy is a subcontractor to its ancestors, and at the same time a client to its descendants. From here on, we will use the term ***cooperating transaction*** to mean an entire hierarchy of client/subcontractor transactions. Where distinction is necessary, we will use the term ***client/subcontractor transaction*** to refer to a node of this hierarchy.

We can combine the above observations to derive the following intuitive model of CAD transactions. A design environment consists of a number of project transactions, each of which may consist of a set of cooperating transactions. Each cooperating transaction is a hierarchy of client/subcontractor transactions. Each client/subcontractor transaction is a set of designer's transactions, which in turn is a set of short-duration transactions. A short-duration transaction is initiated from a window of the designer's workstation. Fig. 2 illustrates our intuitive model.

This model contrasts with the nested-transaction model explored in [KIM84], in which each node of a transaction hierarchy is one designer's transaction, which in turn is a strict sequence of short-duration transactions.



legend: P-tran – project transaction
C-tran – cooperating transaction
S-tran – client/subcontractor transaction
(designer's transaction of Fig. 1)

Fig. 2 Intuitive model of CAD transactions

3. Formal Model of CAD Transactions

As we have seen, the traditional notion of transaction does not capture accurately the notion of a CAD designer's transaction. Not only do we need to generalize the definition of a transaction, but also we need to refine the notion of preservation of consistency in a database. The concept of *serializability*, which served well in data-processing applications, is too restrictive in a CAD environment. In this section, we define a formal model of nested transactions that includes the various transaction types discussed in Section 2. The model is independent of particular concurrency control algorithms, and, to the extent possible, of particular consistency constraints. Our primary purpose in defining this model is to provide a framework for the construction of alternative concurrency control schemes for the management of transactions in a CAD environment.

We will construct our model by defining notions of database operations and database consistency. Using these, we define a general, nested form of transaction. Instead of the standard concept of a *schedule* for a set of transactions, we use the concept of an *execution* of a transaction with nested subtransactions.

The nested model of transactions complicates some aspects of our discussion of correct (i.e., consistency preserving) executions. In these cases, we will represent nested transactions in an unnested form called the *closure* of a transaction. Using the notion of closure, we consider protocols that ensure correct executions.

3.1. Definitions

A *global database* in our model consists of the public database and the private databases of active transactions. We refer to a particular set of values taken on by the global database as a *state* of the database. The set of all possible states is denoted by S.

We use the global database to define our notion of global consistency. Although we do not require a specific form for the integrity constraints that define a consistent global database, we will assume that such a collection of constraints is specified at the time a global database is first created. Thus, we require that a global database satisfies a collection of integrity constraints.

Definition: An *integrity constraint* C is a predicate on S

$$C : S \rightarrow \{\text{TRUE, FALSE}\}$$

The subset of S consisting of states that satisfy constraint C is denoted by S_c.

$$S_c = \{ s \mid s \in S \text{ and } C(s) \}$$

Each transaction in our model is required to preserve some integrity constraint. These constraints form the basis of our concurrency control framework.

At the lowest level of a database system are the physical operations that the hardware allows on the actual data stored on disk. For the purposes of our formal model, we consider *database operations* to be atomic collections of physical operations, such as the reading and writing of the system catalog. We will not concern ourselves, in general, with the manner in which the

atomicity of database operations is ensured.

Definition: A *database operation* (or *operation*, for short) O is a mapping from S to S:

$$O : S \rightarrow S$$

Examples of operations are the following:

- read or write of a private database
- read or write of the public database
- read, write or update of indices for a database (public or private)

Our notion of transaction is a generalization of that of Gray [GRAY81]. We allow nesting of transactions to arbitrary depth [MOSS81]. Transactions in our model may themselves be parallel programs, subject to limitations we will discuss. The nesting and parallelism allow us to capture the notions of client/subcontractor transactions and cooperating transactions, as we will see in the next section.

In the traditional transaction model, the notion of correctness for an execution of a set of transactions is *serializability*. Serializability requires that an execution be equivalent to one that would have occurred without any concurrency in the system. In other words, users (transactions) must not see the effects of concurrent execution. We will allow *non-* serializable executions in our model. In other words, each designer's transaction may see the effect of transactions of other designers within the same project. We will rely on intelligent cooperation of designers within the same project for the preservation of integrity constraints (e.g. circuit interface specification) to ensure that the database remain consistent.

The idea of non-serializability in database systems has been controversial; see, for example, [GARC83]. However, we believe that non-serializability is acceptable in a design environment, since the primary concern is the correctness of the design rather than the sequence of steps that led to the design. The design environment differs from the banking-style transaction environment that motivated the traditional transaction model. In a banking environment, consistency involves

not only the final state, but also the sequence of steps through which the final state was reached.

Lynch [LYNC83] presents a model that, like ours, allows for non-serializability. Under her *multi-level atomicity,* a transaction may specify a set of "breakpoints". The equivalent of atomic execution is guaranteed only between breakpoints. Interleaving may occur among transactions at breakpoints. Our model may be viewed as a scheme for the implementation of a restricted form of multilevel atomicity.

We now define formally our notion of transaction:

Definition: A *transaction* is a 3-tuple (T,O,C) where

- T is a set of transactions or operations
- O is a partial order on T
- C is the integrity constraint preserved by the transaction

We require that for all transactions (t,o,c) in T, C implies c. In other words, nested transactions may have weaker constraints than their ancestors, but not vice-versa. Note that this definition of transaction is recursive. We will assume that the depth of nesting is finite. Thus, at the deepest levels of nesting, transactions consist solely of operations. A transaction represents a set of possible mappings from S to S. When the transaction is run, exactly one of these mappings will be chosen.

To simplify the presentation that follows, we assume that each transaction or operation has a globally unique name. In practice, this can be accomplished by using a hierarchical naming scheme in which each transaction or operation is prefixed by the names of its containing transactions. We use M to denote the set of all operation names.

It is possible to represent any nesting of transaction in an equivalent unnested form, consisting only of operations. For example, let t = (T,O,C) where T = $\{t_1, t_2, t_3\}$, O = $\{t_1 < t_2 < t_3\}$, and C is some consistency constraint. Let the transactions in T consist solely of operations: $t_1 = \{m_{11},m_{12}\}$, $t_2 = \{m_{21},m_{22}\}$, and $t_3 = \{m_{31},m_{32}\}$. The two-level transaction t is clearly

equivalent to a one-level transaction t' = (T',O',C), where T' = $\{m_{11},m_{12},m_{21},m_{22},m_{31},m_{32}\}$, and O' = $\{m_{11} < m_{12} < m_{21} < m_{22} < m_{31} < m_{32}\}$. We call this one-level form t' of t the *closure* of t. The nested form of a transaction is more natural than its closure for most applications. However, we will find the notion of closure useful in the development of the formal model.

Definition: The *closure* $(T,O,C)^*$ of a transaction (T,O,C) is the transaction (T^*,O^*,C) where

- $T^* = (M \cap T) \cup (\cup_X (t',o',c)^*)$ and $X = \{(t',o',c) \mid (t',o',c) \in T - M\}$
- O^* is the partial order induced on T^* defined as follows: Let a and b be operations in T^*. In O^*, $a < b$ if one of the following holds:
  1. a, b $\in$ T and $a < b$ in O
  2. There exist a pair of transactions (t_1,o_1,c_1) and (t_2,o_2,c_2) such that $a \in t_1^*$, $b \in t_2^*$ and $t_1 < t_2$ in O
  3. There exists a transaction (t_1,o_1,c_1) such that a, b $\in t_1^*$ and $a < b$ in $(t_1,o_1,c_1)^*$.

Thus, the closure of a transaction represents all possible partial orderings of operations at any level of nesting in the transaction. By a "possible ordering" we mean any ordering consistent with the orderings specified in the transaction definition. The result is a "flattening" of the transaction hierarchy. Closures, in general, allow several possible orderings of the operations. When the closed transaction is run, one of these orderings will be chosen. This leads us to define the notion of *execution*.

Definition: An *execution* e of a transaction t = (T,O,C) is a transaction (T^*,O',C) where O' is a total order such that O' implies O^*. That is, if a,b $\in T^*$, and a< b in O^*, then a< b in O'. We denote the set of all executions of (T,O,C) by E_t.

An execution of a transaction represents a linear (total) ordering of all operations of a transaction that is compatible with the partial orders defined at the various levels of nesting in the transaction. The effect of an execution e is a mapping from S to S defined as follows: Let m_1, m_2,

..., m_n be the ordered list of operations in an execution e. Then the mapping is the composition of the mappings represented by the operations in e:

$$e : m_1 \; o \; m_2 \; o \; \ldots \; o \; m_n$$

Definition: An execution e of (T,O,C) is *correct* if for all states $s \in S_C$, $e(s) \in S_C$.

The transaction definitions we expect most users of our model to define will permit incorrect executions. That is, we do not expect the orderings O to constrain E_t sufficiently to ensure preservation of integrity. Therefore, it will be the responsibility of the system to ensure that, given an integrity constraint C, only correct executions are permitted. This is analogous to the situation we encounter in the traditional transaction model: individual transactions preserve consistency, but the closure of a set of transactions (called a *schedule* in the traditional model) may not preserve consistency unless special action is taken by the system.

A protocol is a set of rules that restricts the set of admissible executions. Typically, a protocol is implemented by a set of rules that all "well-formed" transactions follow and by an algorithm executed by the system scheduler. We will use an abstract notion of protocol.

Definition: A *protocol* P is a mapping from the set E_t of executions of t to {TRUE, FALSE}. If e is an execution, then P(e) if and only if e is an execution *legal* under P. A protocol P is *correct* if for all executions $e \in E_t$, P(e) implies e is correct.

We do not require a protocol to capture all possible correct executions since such a requirement is not feasible in practical systems. The technical reason that we do not make this requirement has to do with a result of Papadimitriou [PAPA79]. Observe that a protocol is, in effect, an acceptability test for an execution. If we choose serializability as our correctness criterion, a protocol that captures the entire class of correct executions would be a serializability tester. In [PAPA79], it was shown that testing for serializability is an NP-complete problem. Thus, we will be satisfied with protocols that may leave out some correct executions. Such well-known techniques as the two-phase locking protocol [ESWA76], timestamp-ordering algorithm [BERN81], and

the tree protocol [SILB80] are all protocols under our definition.

In general, protocols need to be expressed in terms of executions. Ideally, we would like to express protocols directly in terms of the nested definition of transactions. We say that a protocol is *nested* if it can be expressed in terms of the partial orders of transactions contained within a nested transaction. We illustrate the notion of a nested protocol with a simple example. Consider a two-level nesting of transactions. Let F be a set of transactions of the form (T,O,C), where

- O is the empty order.
- T is a set of transactions of the form (t,o,C), where t consists solely of operations.

One nested protocol for F is the protocol that accepts all total orders of T, and for each (t,o,C) in T, the protocol accepts all total orders consistent with o. Observe that this protocol is a correct one.

Traditional concurrency control techniques tend to produce unnested protocols. As an example, consider once again the family F of transactions defined above. Let us assume that at least some of the transactions include write operations. Two-phase locking is known to be a correct protocol. Note, however, that lock requests need to be checked against locks held by all other transactions. Thus, we cannot construct a nested protocol for the usual form of the two-phase locking protocol.

We choose to define our notion of protocol in an unnested form because we wish to be able to apply traditional concurrency control techniques within our model of the CAD environment. We consider such a capability important since many CAD systems are built on top of existing database systems or are part of a distributed system involving standard database systems. We would like to construct nested protocols that are related closely to traditional techniques and that are sufficiently general to be of use in a practical system. We are investigating this issue currently.

The above-defined model allows us to describe a wide variety of concurrency control

schemes. Indeed, the degree of freedom allowed by the model may be too great for most users. Furthermore, a fully general implementation of the model may involve considerable overhead. In this paper, our goal is to find particular *instances* of the model that capture the kinds of concurrent activity and the level of consistency that we believe is necessary for a flexible CAD environment based on a database system. In the next subsection, we describe several basic types of transaction.

3.2. Basic Examples

We start with the simplest form of transaction: the one used in standard concurrency control theory. A transaction is a linearly ordered collection of operations. This form of transaction is represented in our model by (T,O,C) where

- T is a set of operations
- O is a total order on T
- C is the set of integrity constraints that we wish to enforce on the entire database.

Only one execution exists for this transaction, and it is trivial from the definition that this execution is correct.

Next, we describe the standard notion of concurrency for a set of standard (short-duration) transactions as defined above. One usually talks about a *schedule* for such a set of transactions. In our model, a schedule is simply a transaction consisting of nested transactions. Thus, we define a standard concurrent transaction to be of the form (T,O,C) where

- T is a set of standard transactions
- O is the empty order.
- C is the set of integrity constraints that we wish to enforce on the entire database.

There are a vast number of well-known correct protocols for standard concurrent transactions. In particular, we will make use of two-phase locking [ESWA76] and timestamp ordering [BERN81].

We extend the above definition of standard concurrent transactions to include the reading and writing of disk pages. Standard transactions include as operations database read and database write. They operate as if the actual database were being modified by each operation. However, these operations are usually implemented as transactions that read pages from the disk and write pages back to the disk. In order to represent this in our model, we add a third layer of nesting. A multi-level standard concurrent transaction is of the form (T,O,C) where

- T is a set of transactions of the form (t,o,C), where
  - t is a set of transactions of the form (t',o',c'), where
    - t' is a set of operations chosen from { **read-page, write-page** }
    - o' is a total order on t
    - c' is TRUE
  - o is a total order on t
  - C is the same C as that appearing in (T,O,C)
- O is the empty order
- C is the set of integrity constraints that we wish to enforce on the entire database.

One correct protocol for the above type of transaction is to use two-phase locking for level-2 transactions t and to use two-phase locking for level-3 transactions t' (independent of level 2). In other words, we may use a nested form of the two-phase locking protocol. It is fairly easy to see that the nested version of two-phase locking allows more executions than does the standard unnested form (i.e, a requirement that $(T,O,C)^*$ obey two-phase locking).

The forms of transaction we defined above show that our model is sufficiently general to include the standard model of transaction. The notion of serializability needs to be available in the CAD version of the model since there are occasions (such as catalog updates) where we require serializable actions on a CAD database.

4. Application of the Model to a CAD Environment

In this section we present an instance of the formal model suitable for a CAD environment. First, we will show how the model captures the various types of transaction identified for a CAD environment in Section 2. Then we define the correctness requirements for each type of transaction.

4.1. Transactions in a CAD Environment

At the root of our nested-transaction hierarchy is a *global,* or *root transaction.* Immediately below this transaction is a set of project transactions. These represent different groups of people, working on the same design. As noted in Section 2, each project is independent of the work of other projects, though they may access shared data. Each project therefore would maintain database consistency just as if no other concurrent project transactions exist in the system. Thus, a global transaction P is described by (T,O,C) where T is a set of *project* transactions, O is the empty order, and C is the database integrity constraint. The concurrency control protocol requires that the schedule for the set of project transactions be serializable (the protocol will be discussed later).

Each project transaction consists of a set of *cooperating transactions.* These transactions, as a whole, maintain database consistency but no order is specified among them. Transactions within a cooperating transaction (i.e., designer's transactions) are aware of one another's presence and cooperate to maintain the overall consistency of the database. Since (T,O,C) is a set of cooperating transactions,

- T is a set of transactions of the form (t,o,c)
- O is the empty order
- C is the set of integrity constraints that we wish to enforce on the entire database.

Each transaction (t,o,c) is a cooperating transaction. The concurrency control protocol at this level requires the atomicity of any subcontractor transactions (to be defined formally below) in each (t,o,c) in T.

In the simplest case in which cooperating transactions consist solely of database operations (i.e., there is only one designer's transaction and it has no subtransactions), all that the protocol requires is atomicity of database operations. In this case, a complete definition of a set (T,O,C) of cooperating transactions is as follows:

- T is a set of transactions of the form (t,o,c) where
  - t is a set of transactions of the form (t',o',c') where
    - t' is a set of operations
    - o' is some total order
    - c' is TRUE
  - o is some partial order
  - c is TRUE
- O is the empty order
- C is the set of integrity constraints that we wish to enforce on the entire database.

Observe that the deepest level of nesting represents the atomic operations provided to the designers. Each transaction at the second level of nesting represents an individual cooperating designer. At the top level of nesting, transaction T represents the work being performed by the group of designers within a project.

We generalize the above example to allow fully general designer's transactions and the participation of several designers in a cooperating transaction. To do so, we describe another form of collaboration that involves the assignment of a subtask to one group of designers by another. The assignment of a subtask to another transaction is analogous to a subroutine call in a programming language. The amount of information shared between the two designers is both well-defined and limited. In Section 2, we called this form of working relationship a *client/subcontractor* relationship. This relationship is similar to the one in the model of nested transactions that Moss proposed [MOSS81].

In general, each transaction within a cooperating transaction (i.e., a designer's transaction) is a hierarchy of clients/subcontractors. Each of these transactions can delegate subtasks to one or more subcontractors, thus becoming a client of those subcontractors. This situation is represented by (T_c,o,c), where T_c is a set of operations and subcontractors, o specifies when each subcontractor is initiated relative to the other members of T_c, and c is some local integrity constraint. The concurrency control protocol requires that every client sees every subcontractor as an atomic step executed immediately after the step that initiated it.

Let (T_c,O,C) be a transaction where

- T_c contains T_s, where T_s represents a subcontractor of (T_c,O,C) (i.e., (T_c,O,C) is a client of T_s). Transaction $T_s = (t',o',c')$
- O is some partial order.
- C is the set of integrity constraints that we wish to enforce on the entire database.

We say that T_s is a ***subcontractor*** of ***client*** T_c under protocol P if P(e) is true only if e is equivalent to a correct execution of $((T_c - T_s) \cup t', O'', C)$, where O'' contains:

- All elements $a < b$ of O such that $a,b \in T_c - T_s$
- All elements $a < b$ of o'
- All orderings of the form $a < b$ where $a \in T_c - T_s$ and $a < T_s$ appears in O, and $b \in t'$.
- All orderings of the form $a < b$ where $b \in T_c - T_s$ and $T_s < b$ appears in O, and $a \in t'$.

The ordering O'' above represents the original ordering for the client and the original ordering for the subcontractor. We add the constraint that any operation or transaction within the client that came before the subcontractor in O comes before all operations or transactions of the subcontractor in O''. Similarly, we require that any operation or transaction within the client that came after the subcontractor in O, comes after all operations or transactions of the subcontractor in O''.

In other words, a client/subcontractor relationship holds under a protocol if all executions

are *equivalent* to a "macro expansion" of the subcontractor's operations at some point within the client. The use of subcontractors accomplishes more than the use of subroutines in standard programming languages. Unlike a normal subroutine, the subcontractor transaction is able to run in parallel with its client.

Each client or subcontractor consists of a set of *short-duration transactions,* together with some order defined on them. Thus, the user divides his work into short-duration transactions (essentially for recovery purposes) and executes each of them either sequentially or in parallel. Therefore, a subcontractor is (t,o,c) where t is a set of short-duration transactions, o is a user-defined order, and c is the user's notion of consistency. The concurrency control protocol at this level requires that short-duration transactions be executed in a serializable fashion (this is to be expressed at some lower level).

A short-duration transaction consists of a sequence of database operations which include reads and updates. Thus, a short-duration transaction is (τ , o, c) where τ is a set of reads and updates, o is the user-defined sequential order, and c is the trivial integrity constraint that is always TRUE. Concurrency control at this level requires atomicity (which is equivalent to serializability).

And finally, a *database operation* consists of a set of system operations (catalog access, index reads and writes, page accesses, etc.) which are sequentially ordered. Thus we have (τ,o,c) where τ is a set of system operations, o is a sequential order, and c is trivial. Each system operation is atomic by definition.

4.2. Concurrency Control Requirements

From the discussion of the previous subsection, we conclude that a CAD environment consists of

- a set of concurrent project transactions, where a project transaction is
- a set of cooperating transactions, where a cooperating transaction is

- a hierarchy of clients/subcontractors, where a client/subcontractor transaction is
- a directed acyclic graph of short-duration transactions, where a short-duration transaction is
- a sequence of database operations, where a database operation is
- a sequence of system operations.

Below we describe the concurrency control requirement at each of these transaction levels.

- Serializability of concurrent project transactions can be enforced at the database operation level by a two-phase (potentially long-duration) locking algorithm.
- Cooperating transactions require atomicity of constituent short-duration transactions. A two-phase (short-duration) locking algorithm can enforce this.
- Subcontractor hierarchies require a multilevel concurrency control scheme, implemented at the database operation level. In Section 5, we will describe two algorithms, a virtual timestamp ordering algorithm and a two-phase checkout algorithm, for supporting the atomicity requirements of client/subcontractor hierarchies.
- Database operations must be executed in an atomic fashion. This must be enforced at the system operation level by guaranteeing serializability through a two-phase locking algorithm.

We now consider the semantics of transaction control commands **Begin_Transaction, End_Transaction, and Abort_Transaction** in our model. The semantics of these commands for the short-duration transactions that constitute a cooperating designer's transaction need no explanation. However, we emphasize that an abort induced by the system, due to a deadlock or system crash, will result in an abort of the current short-duration transaction, rather than the cooperating transaction to which the short transaction belongs.

The semantics of Abort_Transaction and End_Transaction for cooperating transactions and project transactions need new operational definitions. Abort of a cooperating transaction means abort of all its constituent short-duration transactions. Similarly, abort of a project transaction is

defined as abort of all its constituent cooperating transactions. Abort of a client transaction also means abort of all its subcontractor transactions. In order to ensure that aborts are executed properly (i.e., in a manner consistent with the consistency constraints), a recovery procedure must be defined. The options available under our model will be discussed in a sequel to this paper.

End_Transaction of a cooperating transaction results in the immediate commit of all updates to the database, if there are no short-duration transactions active within the cooperating transaction. Otherwise, commit is deferred until all active short transactions commit. Similarly, a project transaction immediately commits, at End_Transaction, if there are no active cooperating transactions within the project transaction and there are no subcontractor transactions for the project transaction. Otherwise, commit is deferred.

5. Protocols for a Client/Subcontractor Relationship

In this section we present two protocols that may be used to implement the client/subcontractor relationship in our model. In particular, we note that each of these protocols resolves the difficulty with database consistency in [KIM84]. The first of these, a timestamp algorithm, is suitable for situations in which there is a partial ordering in the client that constrains acceptable subcontractor executions. The second protocol pertains to an important special case in which the client's partial ordering places no constraints on the point at which the "subroutine call" to the subcontractor appears within the execution of the client.

5.1. A Virtual Timestamp Algorithm

The objective of virtual timestamping is to ensure that a subcontractor transaction appears atomic to its client and that it observes the partial order of its client. We resort to *virtual* timestamps, since system-clock-based timestamps in general do not provide the time granularity we need. The version of virtual timestamping we will present enforces a linear ordering on subcontractors of a client transaction. Its extension to partial ordering is straightforward.

When we begin a client/subcontractor transaction T, we assign to it *a start time* ST and *a duration* D. If that transaction consists of a sequence of short-duration transactions 1,2,...,n, we

will assign to each short-duration transaction i a start time st(i) and a duration d(i). Then the following equations have to be satisfied.

$$ST \leq st(1) < st(2) < ... < st(n) < ST+D \qquad (1)$$

$$st(i) + d(i) \leq st(i+1) \qquad (2)$$

When a short-duration transaction i of T spawns a subcontractor T', it assigns to it a start time ST' and a duration D' such that:

$$st(i) < ST' < ST' + D' < st(i+1) \qquad (3)$$

The above is repeated recursively for the depth of the client/subcontractor hierarchy.

Atomicity of subcontractors is guaranteed, if equations (1), (2), and (3) above are satisfied. Now we describe how the start times and durations of subcontractors in these equations may be assigned. Assume that transaction T has start time ST and duration D. Then, we assign start time and duration to its subcontractors by:

$$st(1) = ST \qquad d(1) = D/2$$

$$st(2) = st(1)+d(1) \qquad d(2) = d(1)/2$$

...

$$st(i) = st(i-1)+d(i-1) \qquad d(i) = d(i-1)/2$$

Let transaction i with start time st(i) and duration d(i) spawn subcontractor T'. We assign a start time ST' and a duration D' defined by:

$$ST' = st(i) + \epsilon \qquad D' = st(i+1) - ST'.$$

The above scheme satisfies equations (1), (2) and (3) and does not require an *a priori* knowledge of the number of subcontractors of any client.

We characterize a database operation by a pair (t#,st) where t# is the identifier of the immediate client of the subcontractor transaction that issues the operation, and st is the start time of the short-duration transaction that belongs to the subcontractor transaction. Then access

to data is granted if the data is stamped with a different transaction identifier, or, if already stamped with the same transaction identifier, it has an earlier start time. When access is granted, the data is stamped with the couple (t#,st). This algorithm guarantees that each object is processed by clients and subcontractors in the correct order. The details of the algorithm are analogous to standard timestamp techniques as described in [BERN81].

5.2. A Two-Phase Checkout Algorithm

For expository simplicity, we will only consider update-mode checkouts. The discussion can easily be generalized to include read-mode checkouts.

Definition: We say that a client or subcontractor transaction has associated with it a client space, a private space, and a subcontractor space. The *private space* of a transaction is not shared with any other transaction. A transaction can read and update data in its private space. The *subcontractor space* is where a transaction places private data that subtransactions may check out. A transaction has a unique subcontractor space. The *client space* of a transaction is the subcontractor space of its client, if there is one; otherwise, it is the public database. As a cooperating transaction is in general a hierarchy of client/subcontractor transactions, more than one transaction may share the same client space.

Definition: A *checkout* is the moving of data by a transaction from its client space to its private space. The converse of a checkout is a *checkin,* whereby a transaction moves data from its private space to its client space. A *checkout enable* is the moving of data by a transaction from its private space to its subcontractor space. A *checkout disable* is the moving of data from a transaction's subcontractor space back to its private space.

Definition: We say that a client/subcontractor transaction is observes the *two-phase checkout protocol,* if it checks out data from its client space during one phase and checks them back in during the next phase, such that once it checks in any data, it cannot check out any more data.

We now show that the above protocol ensures the atomicity property required of subcontractor transactions. The discussion is analogous to that described in [ESWA76] for two-phase locking. Let T_1 be a transaction, and let T_2 be a subcontractor of T_1. When T_1 wishes to assign a subtask to T_2, it checkout-enables the necessary data into its subcontractor space (i.e., the client space of T_2). T_2 checks out the data, moving it from its client space to its private space. We emphasize that our two-phase checkout protocol does not impose any restrictions on the order in which T_1 may checkout-enable data or in which T_2 checks them out. In other words, T_2 may dynamically checkout data it needs, possibly over a long duration; likewise, T_1 may dynamically checkout enable data, as T_2's needs become clear.

Once T_2 does not need any more data, it may enter the second phase of the two-phase checkout protocol and starts to check data back into its client space.

As long as T_2 observes the two-phase protocol, the net effect of its execution is just as if T_2 was executed atomically between its last checkout and its first checkin, because none of its input data was changed by T_1 until this last checkout and none of its output data was seen by T_1 before that first checkin.

The semi-public database protocol presented in [KIM84] is similar to our protocol. However, it does not enforce two-phased checkout/checkin: it allows subcontractors to check in and check out data any time. We also note that while both the two-phase checkout protocol and the virtual timestamp algorithm of the previous section ensure atomicity of subcontractor transactions, the former provides the client with a better control over its subcontractors, since the client can specify explicitly the data that the subcontractors can use.

6. Summary

In this paper, we provided an intuitive description and a formal development of a general model of CAD transactions. The model is designed to support a design environment in which a group of cooperating designers can complete a design without being forced to wait over a long duration, and in which a group of designers can collaborate on a design with another group by

assigning subtasks.

Our model maps each of the projects comprising a design project into a set of cooperating transactions. A cooperating transaction is a hierarchy of client/subcontractor transactions, each node of which in turn consists of a set of cooperating designers' transactions. A cooperating designer's transaction is a set of conventional short-duration transactions, initiated from a window of the designer's workstation. The client/subcontractor relationship supports the notion that all actions of a subcontractor are logically executed atomically immediately after its initiation by the client.

Due largely to space limitations, in this paper we have not dealt with implementation issues, including user commands for supporting our model, long-duration two-phase locking between project transactions, recovery within a cooperating transaction and between a client and a subcontractor transaction, and tying the checkout and checkin of objects to the creation and replacement of versions of objects. We will report the results of our work on these in a sequel to this paper.

References

[BERN81] Bernstein, P.A, and N. Goodman "Concurrency Control in Distributed Database Systems," ACM Computing Surveys, 13:2 (June 1981), pp. 185-221.

[ESWA76] Eswaran, K.P., J.N. Gray, R.A. Lorie, T.L. Traiger "The Notion of Consistency and Predicate Locks in a Database System," Communications of the ACM, 19:11 (November 1976), pp. 624-633.

[GARC83] Garcia-Molina, H "Using Semantic Knowledge for Transaction Processing in a Distributed Database," ACM Transactions on Database Systems 8:2, pp. 186-213

[GRAY78] Gray, J. "Notes on Data Base Operating Systems," IBM Research Report: RJ2188, IBM Research, Calif., February 1978.

[GRAY81] Gray, J.N. "The Transaction Concept: Virtues and Limitations," 7th VLDB Conf. pp. 144-154 (1981).

[HASK82] Haskin, R. and R. Lorie. "On Extending the Functions of a Relational Database System," in Proc. ACM SIGMOD Conf., June 1982, pp. 207-212.

[KATZ83] Katz, R. and S. Weiss, "Transaction Management for Design Databases," working paper, 1983.

[KIM84] Kim, W., R. Lorie, D. McNabb, and W. Plouffe. "A Transaction Mechanism for Engineering Design Databases," in Proc. Intl. Conf. on Very Large Data Bases, August 1984.

[LORI83] Lorie, R. and W. Plouffe. "Complex Objects and Their Use in Design Transactions," in Proc. Databases for Engineering Applications, Database Week 1983 (ACM), May 1983, pp. 115-121.

[LYNC83] Lynch, N.A., "Multilevel Atomicity -- A New Correctness Criterion for Database Concurrency Control," ACM Transactions on Database Systems, 8:4 (Dec 1984), pp. 484-502.

[MOSS81] Moss, J.E. "Nested Transactions: An Approach to Reliable Distributed Computing," Ph.D. dissertation, Dept. of Electrical Engineering and Computer Science, MIT, April 1981.

[PAPA79] Papadimitriou, C. "The Serializability of Concurrent Updates," Journal of the ACM, 26:4 (Oct 1979) pp. 631-653.

[SILB80] Silberschatz, A. and Z. Kedem "Consistency in Hierarchical Databases," Journal of the ACM, 27:1 (Jan 1980), pp. 72-80.