A TRANSACTION MECHANISM FOR ENGINEERING DESIGN DATABASES

Won Kim*, Raymond Lorie, Dan McNabb, Wil Plouffe

IBM Research Laboratory
5600 Cottle Road
San Jose, Calif. 95193

Abstract

One primary difference between transactions in an engineering design environment and those in conventional business applications is that an engineering transaction typically lasts a much longer time. Existing proposals for supporting the long-lived engineering transactions are all based on the public/private database architecture, in which a transaction checks out design objects from the public database, modifies them, and checks them into the public database for use by other transactions. However, the design environment which these proposals model is a very rigid one which does not allow a team of designers to complete a complex design involving numerous design objects by passing incomplete objects back and forth among them in a controlled manner. In this paper we present a model of engineering transactions which attempts to resolve this shortcoming as well as satisfying the constraints imposed by the engineering design environment. The model augments existing models by refining the notion of checkout environment which a transaction sees and coupling it with the notion of nested transactions. The model is then extended to a practical mechanism for supporting a complex engineering design environment by imposing the view that a long-lived engineering transaction is really a sequence of conventional short-lived transactions.

1. Introduction

Conventional general-purpose database management systems have been designed primarily to support transaction-oriented business applications in which transactions typically terminate within a few seconds. It is not surprising that such systems cannot support the engineering design applications where transactions may last weeks or months, spanning system crashes and user sessions. When a transaction is short lived, it can be used as the unit of both recovery and consistency [GRAY78]. Locks may be dynamically acquired and held until the end of a transaction, and a transaction can be backed out in case of a deadlock. In an engineering environment, a transaction can still serve as the unit of consistency; however, it should not be used as the unit of recovery [HASK82].

* Author's new address is Microelectronics Computer Technology Corporation, 9430 Research Blvd., Austin, Texas 78759

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

It is generally accepted that the engineering design environment requires a public database system which manages the public database and a number of private database systems running on engineering workstations to interact with the public system [HASK82, LORI83, KATZ83]. The existing models of engineering transactions have been motivated by the need to overcome the shortcomings of conventional models of transactions for the engineering design environment and to support this public/private database system architecture. These models force a rather rigid design environment in which a designer checks out design objects from the public database system as needed, updates the objects using his private database system, and checks into the public system the completed objects.

However, these are inadequate for the complex design environment where a large number of designers, possibly representing an organizational hierarchy (division, department, function, project, and so on) must complete a complex design involving numerous design objects by passing incomplete objects back and forth among them in a controlled manner. A primary advantage of allowing a designer to check out an object from another designer is that a designer may check out a partial design and complete the design on behalf of the initial designer or make use of it to design his own objects. This situation may arise because the designer has modified a checked-out object to a state where parceling out further work on the object to a number of other designers is desirable or necessary. Suppose that a designer who is working on a family of gates has at some point completed the design of an AND gate but not the others in the family. Because the entire set of gates has not been designed, the designer cannot check in his design of the AND gate to the public database for use by other designers. At this point, it will be useful if other designers can directly check out the AND gate for use in their own designs, or, say, the incomplete OR gate to complete the design and return it to the initial designer.

In this paper we will describe a model of engineering transactions which augments existing models by first refining the notion of checkout environment which a transaction sees and coupling it with the notion of nested transactions proposed in [DAVI78, REED78, MOSS82]. The model will then be extended to a practical mechanism for supporting a complex engineering design environment by imposing the view that a long-lived engineering transaction should really be a sequence of conventional short-lived transactions. This model not only overcomes the inadequacies in existing proposal for engineering transactions but also supports a

design environment involving a large number of designers and design objects. We note also that our model provides a rather natural framework for the management of versions and alternatives of design objects, since the hierarchy of nested transactions closely corresponds to the hierarchy of versions and alternatives derived from objects at higher levels of abstraction. This aspect, however, is outside the scope of the paper and will not be further discussed.

The remainder of the paper is organized as follows. Section 2 provides a discussion of our transaction model, starting with an intuitive overview and leading up to a complete (though informal) description. The semantics of the model are further discussed in Section 3. Section 4 describes a number of features which transform the model into a practical mechanism. A number of major implementation issues are considered in Section 5. Section 6 compares our model with other relevant transaction models found in the literature.

2. Model of Nested Transactions for Engineering Design Databases

2.1. Intuitive Overview of the Model

Existing models of engineering transactions support the same set of commands that conventional transactions support: BEGIN_TRANSACTION (BT), END_TRANSACTION (ET), ABORT (A), SAVE (S), and UNSAVE (U). An engineering transaction is initiated with a BEGIN_TRANSACTION and terminated by an ABORT or END_TRANSACTION.

Once a transaction BEGINs, it CHECKs OUT (receives copies of) design objects from the public database system. The transaction issues queries and updates against the objects, which have been inserted into its private database. It can SAVE changes any time, and UNSAVE to an arbitrary SAVE point to back out the changes. When the transaction ENDs, the modified design objects are CHECKed INto the public database system.

To allow a transaction to check out objects from other transactions which are not ready to commit them to the public database, we introduce the notion of **semi-public databases**. A semi-public database is associated with each engineering transaction, and is the repository of design objects which the transaction considers in some sense correct and as such other transactions may check out. In other words, the **checkout environment** of a transaction includes the public database and the semi-public databases of other transactions. (It will be shown in Section 5 that the public database system will actually manage both the public database and the semi-public databases.)

We introduce two new commands to support the notion of semi-public databases: DOWNWARD COMMIT and UPWARD COMMIT (or CHECKIN). A transaction DOWNWARD COMMITs a set of objects to its semi-public database for checkout by other transactions. The transactions which check out objects from a transaction's semi-public database become its children transactions. When a transaction UPWARD COMMITs a set of objects, it transfers (checks in) the objects to the semi-public database of its parent transaction.

A transaction may issue DOWNWARD or UPWARD COMMITs at any time; that is, a DOWNWARD or UPWARD COMMIT does not mark the termination of the transaction. When a transaction terminates (ENDs or ABORTs), all objects in its semi-public databases are automatically UPWARD COMMITted to the parent's semi-public database. If the transaction has no parent, its objects are UPWARD COMMITted to the public database.

Just as a transaction may not UNSAVE changes which it has committed to the public database, objects committed to the semi-public databases cannot be UNSAVEd. If changes must be made to the objects committed to the semi-public database, the transaction must check them out again, update them, and put them back to the semi-public database.

Figs. 1 and 2 below illustrate the concepts discussed thus far. In Fig. 1, T1 checks out objects O1, O2, and O3 and updates O1 and O3 to a consistent state. T1 DOWNWARD COMMITs (DC) the objects. Transaction T2/1 can check out O1 and/or O3 which have been committed by T1. (T2 is the identifier of the transaction and /1 indicates its parent.) T1 further checks out O4, O5... and T2/1 goes on to modify O1 and check out O6 and O7. The checkout environment of T2/1 is the public database, the semi-public database and of course its own semi-public database.

Once T2/1 decides that O1 has been updated correctly, it may UPWARD COMMIT (UC) O1 to the semi-public database of T1, as shown in Fig. 2. T1 can now check out O1 again and see the changes made by T2/1. T2/1 can proceed to check out O6, O7 and update them. If T2/1 terminates, all the objects which have not already been UPWARD COMMITted (O6, O7) are automatically UPWARD COMMITted to T1.

Now suppose T1 in Fig. 1 must UNSAVE the changes it has made to the beginning of the transaction, after DOWNWARD COMMITting O1, O3. When T1 UNSAVEs, updates to O2 will be undone; however, updates to O1 and O3 will remain. This may be viewed as if O1 and O3 have been completely removed from the private database of the transaction.

Similarly, suppose in Fig. 2 that T1 must be UNSAVEd, say to the beginning of the transaction, after T2/1 UPWARD COMMITs O1. Again, as O1 has been placed in T1's semi-public database, changes to O1 will not be affected by the UNSAVE.

2.2. Precise Description of the Model

The transaction model overviewed in the previous subsection may be generalized in two directions in a straightforward manner. First, a dependent transaction can itself DOWNWARD COMMIT objects for checkout by other dependent transactions. This will give rise to a hierarchy of nested transactions. Second, a transaction can not only check out objects from the public database and the semi-public database of its parent, but also the semi-public databases of any transactions in its chain of ancestors. Each non-leaf node of a transaction hierarchy then controls the checkout of objects in its semi-public database by its descendant transactions.

We can now completely describe our transaction model.

1. A transaction may check out objects from its checkout environment, which includes its own semi-public database, the semi-public databases of all transactions in its chain of ancestors, and the public database.

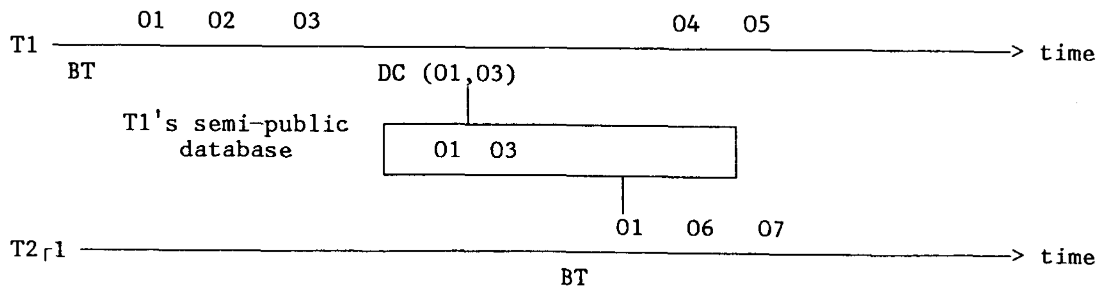

Fig. 1

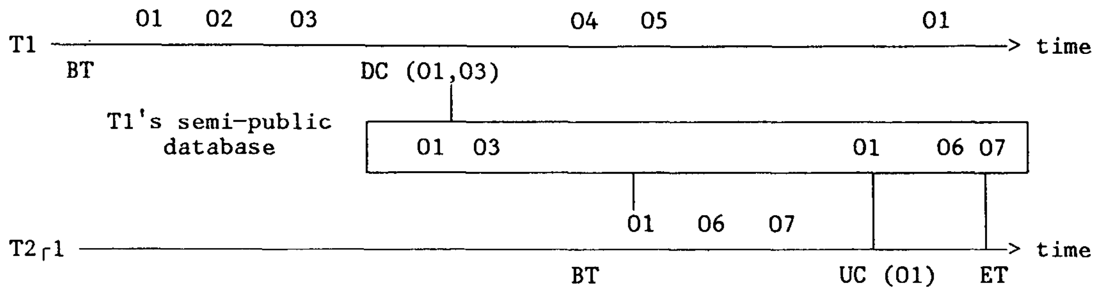

Fig. 2

Any transaction may check out any objects from the public database, A transaction may check out from the semi-public databases of transactions on its chain of ancestors on the same transaction hierarchy. A transaction which is the root of a transaction hierarchy may check out from any transaction on another transaction hierarchy, giving rise to a new parent-child relationship between the two transactions. A transaction may not check out from anywhere else.

We take the view that a direct checkout of an object from a transaction at a higher level of the transaction hierarchy than the parent is equivalent to a succession of checkouts from the transaction down the transaction hierarchy. This means, in particular, that the public database is the virtual root of all transaction hierarchies, and as such a direct checkout from the public database is regarded as a succession of checkouts down a transaction hierarchy.

2. When a transaction checks out an object, it receives a copy of the object from its checkout environment. The object becomes part of the private database of the transaction. This means that if the transaction UNSAVEs to a save point preceding the checkout, the object is implicitly dropped from the private database of the transaction and the lock on the object returns to the transaction from which the object was checked out.

3. Objects are committed to the semi-public database of a transaction when the transaction DOWNWARD COMMITs them or its children transactions UPWARD COMMITs them. Objects stored in the semi-public database of a transaction are not affected by UNSAVEs. The transaction may change them only by checking them out, updating and committing them to the semi-public database again.

We note here that although the semantics of SAVE and UNSAVE are correct in the present context, the discussion of Section 4.3 will lead us to drop these commands from our transaction mechanism.

3.1. When a transaction DOWNWARD COMMITs objects to its semi-public database, they become available for checkout by other transactions. Once the transaction DOWNWARD COMMITs an object, it must in theory compete with other transactions to check out the object again.

3.2. When a transaction UPWARD COMMITs objects, they enter the semi-public database of the parent transaction and become part of the parent's checkout environment. The objects become invisible to the transaction whichon UPWARD COMMITted them. If the parent already has an older version of the same objects, they are replaced by the new version being UPWARD COMMITted.

A transaction UPWARD COMMITs objects to its immediate parent, regardless of where in the chain of ancestors it had initially checked them out. In particular, a transaction UPWARD COMMITs an object to its immediate parent an object, even if it had directly checked it out of the public database. The parent is then responsible for UPWARD COMMITting the object to its parent, and so on, up to the semi-public or public database from which the object was initially checked out.

We note that an UPWARD COMMIT may cause the transaction hierarchy to be altered. Once a transaction Ti UPWARD COMMITs all the objects it checked out of its chain of ancestors, and no transactions have checked out any objects from Ti, Ti becomes an independent transaction and drops out of the current transaction hierarchy. However, as long as the transaction retains any object it checked out of any transaction on its chain of ancestors, the transaction hierarchy is not altered. This means, for example, that even if a transaction UPWARD COMMITs all the objects it checked out of its parent, it remains dependent on the parent if it

had checked out objects from an ancestor of the parent.

4. When a transaction terminates, all the objects in its semi-public database become part of the checkout environment of its parent transaction. If the transaction ENDs, all the objects which have not been committed to its semi-public database are UPWARD COMMITted to the parent. If the transaction ABORTs, all uncommitted objects are simply dropped. (In Section 4.3, we will further discuss the semantics of ABORT and in fact drop it from the list of meaningful commands in a long-lived transaction.)

5. If a transaction terminates before all of its dependent transactions have terminated, the dependent transactions become dependent on the parent of the terminating transaction. In particular, if the root transaction terminates, the public database becomes the virtual root transaction, and all UPWARD COMMITs are directed to it.

3. Further Discussions of the Semantics of the Model

3.1 Transaction Consistency

For expository simplicity, we have so far assumed that all objects are checked out in write (W) mode. We can easily refine the checkout modes to W, R and RW, discussed in [LORI83].

An R-mode checkout is equivalent to the read lock in the conventional share/exclusive lock scheme. It does not conflict with R-mode checkouts by other transactions; however, it conflicts with a W-mode checkout.

An RW-mode checkout conflicts with neither R nor W. That is, a transaction may check out an object in RW mode, even if other transactions may have checked it out in R or W mode; and that some transactions may have checked out an object in RW mode has no bearing on whether other transactions may check out the object in R or W mode. An RW-mode checkout is merely a request to receive a copy of an object from the public database or semi-public database of any transaction. The system is not concerned with the possibility that a transaction may use some of the information in that object to produce modifications to other objects. Therefore, a transaction may check out any object in RW mode, without being constrained to its checkout environment. Put in a different way, no parent-child relationship is introduced as a result of an RW-mode checkout.

As discussed in [GRAY78], a transaction may upgrade a lock to a more exclusive lock; however, it may not downgrade a lock to a less exclusive lock. For example, an R lock may be converted to a W lock, while a W lock may not be downgraded to an R lock. This is known as *lock-mode conversion*. On our a hierarchy of nested transactions, a transaction may convert an R-mode checkout of an object into a W-mode checkout, if no other transaction has already checked out the object in R or W mode.

Transaction T is said to observe the *consistency lock protocol* if it observes the following rules [GRAY78]:

1. T sets a W lock on any data it updates,
2. T sets an R lock on any data it reads, and
3. T holds all locks until the end of transaction.

[GRAY78] defines three levels of consistency. Level-one consistency only avoids lost-update and deadlock during transaction backout. Level-two consistency avoids dirty read as well. And level-three consistency further guarantees repeatable reads. Level-three consistency is obtained when the consistency lock protocol is observed.

Ignoring RW-mode checkouts, our model guarantees only level-two consistency between transactions within a transaction hierarchy. The reason is that read is not repeatable within a transaction hierarchy. Suppose a transaction Tj checks out an object Oi from its parent Ti, updates it, and UPWARD COMMITs it. Then another transaction Tk checks out object Oi from Ti and UPWARD COMMITs it to Ti after updating it. If Tj checks out the object from Ti again, it will see a different object.

However, it is easy to see that our model guarantees level-three consistency between different transaction hierarchies. Further, if we require that a transaction UPWARD COMMIT all objects when it terminates, the model guarantees level-three consistency even between transactions within the same transaction hierarchy. Rules 1 and 2 are enforced when objects are checked out. Rule 3 is enforced by propagating locks from dependent transactions through their chains of ancestors. The root of the transaction hierarchy ends up with locks on all the objects and holds them until it terminates. When the root of the transaction hierarchy terminates, all the objects are UPWARD COMMITted to the public database and all the locks are released.

3.2 Permanence of the Semi-Public Databases

A critical question of far-reaching consequence is whether we allow the UNSAVE to back out the objects which have been committed to a semi-public database. If the UNSAVE causes a backout of the objects in the semi-public database, it may cause backouts of entire subtrees of transactions, since other transactions may have checked out the objects. This approach may be too drastic and that it will significantly complicate our transaction model and its implementation. Instead, we have adopted the alternate view that when a transaction allows other transactions to check out its objects, it is because the transaction (i.e., the designer who is running the transaction) has decided that the objects are stable and correct. This view led us to introduce the notion of DOWNWARD and UPWARD COMMIT which will move out of the path of the UNSAVE those objects that a transaction has deemed stable enough to allow checkouts by other transactions.

If an object which has been UPWARD- or DOWNWARD-COMMITted must be undone, the transaction which has the W access to the object in its checkout environment must explicitly check out the object and modify it. Although this approach makes wholesale backout of committed objects rather cumbersome, it also offers some clear advantages. Once a dependent transaction checks out a committed object, it need not be concerned about the possibility that the object may be UNSAVEd by the parent transaction. Remember that UNSAVE may be forced upon an uncommitted object when the private database system recovers from crashes, particularly hard crashes that destroy its disk contents.

3.3 Constraints on the Checkout Environment

We will now justify our decision to limit a transaction's checkout environment to the semi-public databases of transactions on its

chain of ancestors. Consider for example a transaction hierarchy in which T1 is the root, T2/1 and T3/1 are immediate children of T1, and T4/2 is a child of T2/1. T4/2 cannot check out objects from T3/1; T3/1 must UPWARD COMMIT them to T1, before T4/2 can check them out (from T1).

Suppose this environment constraint is not imposed. Then T4/2 may check out an object O3 from T3/1 and O2 from T2/1, update them, and check them back into T3/1 and T2/1, respectively. The trouble is that T4/2 may have updated the objects such that they are now semantically dependent on each other. As such, after they have been checked in, any changes to one may require corresponding changes in the other. However, since O2 has been checked into T2/1 and O3 into T3/1, neither transaction has direct control over this semantically interdependent set of objects.

In fact, this is also the reason we have taken the view that an object checked out by a transaction Tk from the public database or the semi-public database of an ancestor transaction Ti should be checked in through a succession of UPWARD COMMITs up the chain of ancestors of Tk, rather than directly back to the public database or the semi-public database of Ti.

We have maintained that a transaction may check out objects only from its chain of ancestors. If a transaction Ti must check out an object from transaction Tj which is not on its chain of ancestors, the object would have to be first UPWARD COMMITted to transaction Tk, the first ancestor common to Ti and Tj, and Ti must check out the object from Tk.

If the transaction hierarchy becomes deeply nested, this process can become cumbersome. One way to improve this situation would be to allow the transaction hierarchy to be restructured. In the current example, Ti will become a child of Tj.

The algorithm for restructuring a hierarchy is quite simple. Suppose Ti is to check out an object from Tj, a transaction which is on the same transaction hierarchy with Ti but which is not an ancestor of Ti. Let Tk be the first common ancestor to both Ti and Tj. If Ti checks out the object from Tj, the subtree of transactions rooted at Ts, a transaction which is the highest ancestor of Ti and an immediate child of Tk, becomes a subtree of Tj. This is illustrated in Fig. 3. We are currently investigating a feasible mechanism to support this feature.

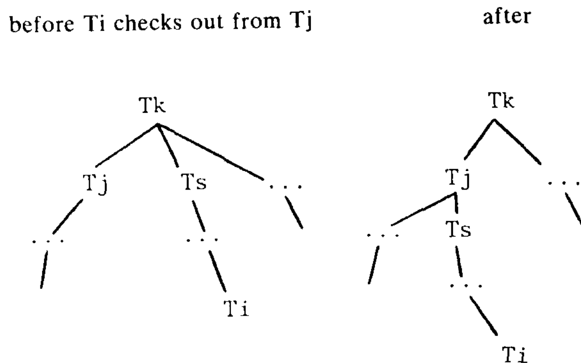

Fig. 3 Restructuring a transaction hierarchy

4. Extending the Model to a Transaction Mechanism

4.1 Undoing and Selective Authorization of CHECKOUTs

We have shown that DOWNWARD COMMIT makes available some objects to dependent transactions on the transaction hierarchy. We need finer control over which dependent transactions get to check out DOWNWARD-COMMITted objects and UPWARD COMMIT them. We now introduce three additional commands for this purpose; CANCEL, ALLOW, and DISALLOW. (As these features are somewhat orthogonal to a discussion of our transaction model, we will not attempt to provide a full account of these commands in this paper.)

A dependent transaction may CANCEL an object it checked out. The command is used in effect to undo a checkout. However, this does not cause the parent transaction to implicitly take away the checkout privilege from the dependent transaction. When a dependent transaction CANCELs an object, any record of the checkout is erased from the system's control structures, and the parent-child relationship between the transactions is broken if the checkout being CANCELed is the only reason for the parent-child relationship. A CANCELed object of course cannot be UPWARD COMMITted, since it was in effect never checked out.

Further, the object being CANCELed is dropped from the transaction's private database, or semi-public database (if it is in the semi-public database, but not in the private database).

When a transaction DOWNWARD COMMITs an object, it implicitly grants the RW checkout privilege to all transactions. However, it must use the ALLOW and DISALLOW commands to grant and revoke R and W privileges to specific transactions. (Of course, dependent transactions must first have been started, so that their identifiers are known.) ALLOW and DISALLOW are essentially equivalent to the GRANT and REVOKE commands supported in System R to grant and revoke read and write privileges on tables to selected users [BLAS81].

A transaction T may ALLOW specific dependent transactions. Or it may allow all but some transactions by simultaneously ALLOWing PUBLIC and DISALLOWing the particular transactions.

Analogous to GRANT and REVOKE, the system in a sense regards PUBLIC as a special single transaction. If a transaction T ALLOWs PUBLIC, a dependent transaction Ti is implicitly ALLOWed (unless T explicitly DISALLOWed Ti). If T now explicitly ALLOWs Ti and then DISALLOWs PUBLIC, Ti still retains its ALLOW.

A DISALLOW in R mode disallows both R and W-mode checkouts. A W-mode DISALLOW disallows W-mode checkouts while implicitly allowing R-mode checkouts. If T had previously ALLOWed Ti, the DISALLOW negates the ALLOW and CANCELs all checkouts on object O. If the newly DISALLOWed mode is R, both R and W checkouts are CANCELed. If the DISALLOW is in W mode, W checkout is CANCELed. The objects implicitly CANCELed by a DISALLOW then cannot be UPWARD COMMITted. T's DISALLOW of object O, however, does not CANCEL any checkouts from the semi-public databases of T's descendants which have DOWNWARD COMMITted versions of O.

4.2 Supporting Multiple Transactions on a Workstation

We now introduce two final commands: SUSPEND and RESUME. The SUSPEND command allows a designer to suspend a transaction to use his workstation and private database system for some other purposes; for example, to initiate another transaction. After a while, he may RESUME the transaction at a point where it was SUSPENDed.

4.3 Superimposing Short Transactions on a Long Transaction

The model of long-lived transactions developed in the previous sections is sufficiently powerful for supporting an environment where the long transactions only check out objects from their checkout environments, and manipulate them in their private databases, and check them back in the public and semi-public databases. However, we believe that long transactions must also be able to interact with the public system in conventional ways. That is, they must be able to issue queries, data manipulation statements (update, insert, delete), data definition statements (create and drop tables, indexes, views) and data control statements (authorization) against the database the public system manages.

An engineering design system must provide support for managing two distinct types of database; the design database of design objects, and the conventional database for design administration. The design objects are usually collections of related records, for example, complex objects [LORI83], and are given system-generated unique identifiers. The direct query/manipulation capability will be essential to query and update not only the design administration database, but also such system control data as system catalogs and even the checkout/checkin control structures. The feature will also be used to define design objects, besides defining and creating access paths to the non-design database.

To provide this capability, we have taken the view that each long-transaction within a transaction hierarchy is actually a sequence of conventional, short-lived transactions. Within these short transactions, queries, data manipulation statements, data definition statements, and data control statements may be issued against the transaction's private database and the public database. Engineering transaction commands we have discussed will also be issued as part of these short transactions against the transaction's checkout environment. This is illustrated in Fig. 4, where T is a long transaction, and end__t means end of a short transaction.

All changes to the public, private, and semi-public databases which result from a short transaction will be committed together, when the short transaction commits. If the short transaction aborts, the effect of all the long-transaction commands and queries and updates issued within the short transaction will be backed out to the end of the preceding short transaction. As a short transaction involves two systems, the public system and a private system, its commit and abort require use of a coordinated two-phase commit protocol [GRAY78]. And a long transaction can be backed up only as far as to the beginning of the current short transaction, in case of intentional ABORT of the short transaction or crashes of the private/public database systems.

Since all long-transaction commands, even the DOWNWARD and UPWARD COMMIT, are now 'recoverable' (can be backed out), when a long transaction DOWNWARD or UPWARD COMMITs an object, logically the object does not enter its semi-public database until the short transaction commits. In particular, even when an object is DOWNWARD COMMITted and ALLOWed, a dependent transaction cannot check it out until the short transaction commits the DOWNWARD COMMIT and ALLOW commands.

Further, in view of the fact that the effect of each short-transaction must become 'permanent' when it commits, the SAVE/UNSAVE commands as well as ABORT we discussed in the context of long transactions can only be effective within a short transaction. That is, an UNSAVE or ABORT can back up the changes that have taken place only up to the beginning of the current short transaction. Thus our approach to superposing short transactions on a long transaction effectively eliminates the SAVE, UNSAVE, and ABORT as long-transaction commands.

4.4 User Interface

Below, we summarize the user interface for all the long-transaction commands we have discussed thus far.

1. DOWNWARD COMMIT has one parameter, the list of objects being committed.

2. UPWARD COMMIT has one parameter, the list of objects being committed.

3. CHECKOUT has three parameters: identifier of the object to be checked out, checkout mode, and identifier of the transaction from which the object is to be checked out. If the identifier of the target transaction is not specified, the target defaults to the (semi-)public database which has the most recent version of the object.

4. BEGIN__TRANSACTION has one optional parameter, the transaction identifier. If the user does not supply the transaction id, the system generates an identifier which will be unique across the entire network of private database systems.

5. SUSPEND has no parameter. The system automatically ends the current short transaction.

6. RESUME requires the transaction identifier of the transaction to be resumed.

7. The ALLOW command has three parameters: identifier of the object being allowed for checkouts, the access privilege being given to other transaction, and optionally, the identifier of the transaction which may check out the object. If transaction id is not specified, it defaults to PUBLIC.

8. The DISALLOW command has three parameters: identifier of the object whose checkout is being disallowed, the checkout mode being disallowed, and identifier of the transaction being disallowed.

9. The CANCEL command has only one parameter, the list of the identifiers of the objects being dropped.

5. Implementation Considerations

At IBM Research, San Jose, we are currently prototyping an Engineering Design Database System by extending the functions of System R [BLAS81, CHAM81]. The nested long-lived transaction mechanism described in this paper is being incorporated into this prototype system.

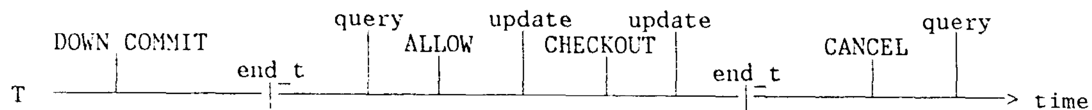

Fig. 4 Short transactions superimposed on a long transaction

The system will have a public database system which manages the public database and a number of private database systems connected to the public system in a star configuration. The host itself may be a network of computers on which a distributed database system runs. The private database systems may run in one of a few different environments. It may run as a single-user system on an engineering workstation with a private disk drive. Or it may be one of a number of database managers that run on a medium-scale computer with multiple I/O channels and disk drives under a virtual memory operating system.

The workstations will communicate with one another through the central, public system. This means that the public system will control the checkout and checkin of all objects, in both the globally consistent public database and the semi-consistent, semi-public databases of the nested transactions. The public system will maintain the control structures to keep track of the checkout environment which any transaction can see at any time.

All the long-transaction commands, as well as direct queries and updates against the public and semi-public databases, must be sent to the public system for processing. A CHECKOUT request will cause the public system to make a copy of the requested object, send it to the requesting private system, and update the checkout control structures. A DOWNWARD and UPWARD COMMIT command will cause the public system to insert the object being committed into the semi-public database, and update the checkout control structures. The CANCEL, ALLOW, DISALLOW, BEGIN_TRANSACTION, END_TRANSACTION all cause the public system to update the control structures.

These control structures must persist across system crashes and user sessions. This means that we will organize the control structures as relations and manage them using the concurrency control and recovery mechanisms of System R, which will serve as the public database system.

It should perhaps be mentioned that when an object is checked out by a transaction, the object is not visible to another transaction executing within the same private system. This separation of private database space between transactions is particularly important when different transactions running on the same private system check out the same object in different modes; for example, one transaction in W mode and another in RW mode.

Before concluding this section, we must point out that our transaction mechanism incurs only a minor additional cost, in terms of both system performance and system implementation, over exisiting proposals. Implementation of the checkout and checkin commands and manipulation of the persistent control structure which support them represent the major implementation and performance overhead in all models of long-lived engineering transactions. Our mechanism requires only a slight complication in the algorithms for manipulating the checkout/checkin control structures, primarily to support the ALLOW, DISALLOW, and CANCEL commands.

6. Comparisons with Other Proposals

The models for engineering transactions found in the literature [HASK82, LORI83, KATZ83] do not support the notion of nested transactions. By introducing the notion of semi-public databases and coupling it with the notion of nested transactions, we are providing a mechanism for supporting a complex design environment, by allowing designers (transactions) to check out and work on partial designs in a controlled manner from other transactions.

Our model of nested transactions shares some characteristics with conventional models of nested transactions [DAVI78, REED78, MOSS82]. In these proposals, a transaction can spawn subtransactions, which can execute in parallel. The subtransactions can in turn spawn their own subtransactions. Each subtransaction inherits locks from its chain of ancestors, and is logically serialized with respect to other concurrent subtransactions in the hierarchy of transactions. When a subtransaction commits, its locks and recovery mechanisms are passed onto its immediate parent transaction, and its updates are made visible to the parent. Commitment of a subtransaction is conditional; that is, updates committed by a subtransaction may be backed out by any transaction in its chain of ancestors.

However, our model differs from these others in some important ways. These differences result from the long-lived nature of engineering transactions.

First, the DOWNWARD COMMIT in our model allows a transaction to commit its updates and to dynamically build up the checkout environment for dependent transactions. This notion is completely absent in other models of nested transactions.

Second, the updates committed by an UPWARD or DOWNWARD COMMIT are not conditional, in that the transaction which has the updated objects in its checkout environment must explicitly check out the updated object and undo the updates.

Third, our model grants specific access privileges to specific transactions, rather than relying solely on the lock mechanism to resolve the problem of who gets which type of access to an object.

Fourth, our model incorporates the view that a long transaction is a sequence of individually committable, conventional short transactions.

Fifth, the UPWARD COMMIT in our model allows a dependent transaction to commit its updates to its parent transaction at any time. Other models, as the conventional model of non-nested transactions require all updates to be committed only at the end of a transaction.

For completeness, we note that Gray proposed an entirely different model of nested transactions [GRAY81]. Gray gives an exam-

ple in which a transaction consists of a number of subtransactions. These subtransactions are executed in sequence. Each subtransaction issues an unconditional commit. This is exactly the way in which we view any one transaction within a hierarchy of transactions as being a sequence of short transactions.

However, in Gray's model, when a subtransaction commits, all its updates become immediately visible to other transactions and subtransactions. In the context of an engineering design environment, this means that when a subtransaction commits, its updates are committed to the public database, rather than to the parent transaction. But this is exactly the existing model of engineering transactions without the nesting of transactions.

Summary

In this paper we described a model of engineering transactions which combines and generalizes key concepts found in both the existing models of engineering transactions and nested transactions.

Our model of engineering transactions is based on three key concepts; the notion of nested transactions, the notion of semi-public databases associated with the nested transactions, and the view that a long-lived engineering transaction is a sequence of individually committed, conventional short-lived transactions.

The database logically consists of the public database and a collection of semi-public databases. The public database is the repository of objects from which all transactions check out and into which they check in completed objects. A semi-public database is the halfway house into which a transaction commits objects which are not ready to be checked into the public database but which are consistent enough for checkouts by other transactions.

When a transaction Tj checks out objects from another transaction Ti, it becomes a child of Ti. Further, a transaction Tk may check out objects from Tj and become a child of Tj and a grandchild of Ti. In this way a hierarchy of transactions, called nested transactions, is established. A separate semi-public database is associated with each transaction on the transaction hierarchy. A transaction may check objects out of the public database and the semi-public databases associated with the transactions which are its ancestors on the transaction hierarchy.

Each long transaction on a transaction hierarchy is a sequence of conventional short transactions. Within each short transaction, checkout and checkin requests are issued, as well as normal query, data manipulation, data definition, and data control statements against those objects which are not units of checkout and checkin.

Our model provides considerable flexibility in allowing a team of designers to complete a complex design involving numerous design objects by passing incomplete objects back and forth among them in a controlled manner. Further, since the hierarchy of nested transactions closely corresponds to the hierarchy of versions and alternatives derived from objects at higher levels of abstraction, we expect that this model can also serve as the framework for the management of versions and alternatives of design objects.

References

[BLAS81] Blasgen, M.W. et al. "System R: An Architectural Overview," IBM Systems Journal, vol. 20, no. 1, Feb. 1981, pp. 41-62.

[CHAM81] Chamberlin, D.D. et al. "A History and Evaluation of System R," Commun. ACM, vol. 24, no. 10, October 1981, pp. 632-646.

[DAVI78] Davies, C.T. "Data Processing Spheres of Control," IBM Systems Journal, vol. 17, no. 2, 1978, pp. 179-198.

GRAY78 Gray, J. Notes on Data Base Operating Systems, IBM Research Report: RJ2188, Feb. 1978.

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

HASK82 Haskin, R.L., and Lorie, R.A. "On Extending the Functions of a Relational Database System," *ACM SIGMOD Conf.*, 207-212 (1982).

KATZ83 Katz, R.H., and Weiss, S. "Transaction Management for Design Databases," Working Paper (1983).

LORI83 Lorie, R., and Plouffe, W. "Complex Objects and Their Use in Design Transactions," *Database Week - Databases for Engineering Design,* (1983).

[MOSS82] Moss, J.E. Nested Transactions: An Approach to Reliable Distributed Computing, Ph.D. disseration, MIT Laboratory for Computer Science, Technical Report: MIT/LCS/TR-260, 1981.

[REED78] Reed, D. Naming and Synchronization in a Decentralized Computer System, Ph.D. dissertation, MIT Laboratory for Computer Science, Technical Report: MIT/LCS/TR-205, 1978.

[SQL82] SQL/Data System Concepts and Facilities, IBM Corp. Form GH24-5013-1 File No. S370/4300-50, Feb. 1982.