A MULTI-LEVEL TRANSACTION MODEL FOR ENGINEERING APPLICATIONS

Rainer Unland
Gunter Schlageter

University of Hagen, Department of Computer Science
P.O. Box 940, D-5800 Hagen 1
West-Germany

Abstract

Before database systems are suitable to support new application areas, like for example CAD/CAM or software engineering, they have to consider the special requirements of these environments. Main characteristics are long duration of the design process, group-oriented division of areas of responsibility, and cooperative work between and within these groups. Due to both the complexity, the group-orientation, and the long duration of design processes and the enormous number of objects involved, a multi-level hierarchical transaction concept seems to be absolutely necessary. In this paper we will introduce such a concept. The suggested transaction model has several levels which, among others, allow to naturally reproduce the hierarchy within a design process: On the one hand a design team has to be protected from the outside world. This is done with the help of a so called database transaction. This transaction is very similar to a conventional transaction. On the other hand mechanisms are required which not only allow to express the hierarchy within a project but also support a controlled teamwork on common objects. For these purposes group transactions and user transactions are introduced. Group transactions are especially tailored to the cooperative kind of work of a group. The real development process, however, takes place in user transactions which are installed as children of group transactions. Both kinds of transactions do avoid the rigid synchronisation techniques of conventional database systems. Instead, they offer some flexibility in working on and with data essentially by offering an extensive communication and information system. A prototype of this model was implemented on a VAX on top of the ORACLE relational database system.

1. Introduction

The conventional model of transactions as it was presented by /Gray79/ is based on the ACID principle /HaRe83/ which, in short, means that a transaction is an atomic unit of reads and writes against a database. To support atomicity and isolated execution a transaction is not allowed to see changes performed by any parallel transaction. Additionally, the all or nothing principle must be obeyed. Either all changes are reflected in the database or none. As long as transactions process within a few seconds this model is suitable. Because transactions are shortliving little work will be lost in case of a backup. Additionally, transaction have to wait only for a very short time if they are blocked due to a conflict with a parallel transaction.

Non standard applications, like e.g. CAD/CAM or software development, behave totally different. Typically, transactions are interactive and of long duration. Most of the work performed by a designer is only expressed within this transaction and documented nowhere else. Therefore, of course, long waits or rollback in case of a transaction or system failure are unacceptable. Another important point is that design projects are usually performed by groups of experts who want to work in a cooperative fashion. These requirements led to some proposals of new transaction models which satisfy the above mentioned demands. We lump together design, engineering, and software development projects under the term design projects.

In /HaLo81/ the CHECKOUT/CHECKIN mechanism is proposed. The main idea of this approach is that objects are no longer modified directly in the public database (update in place) but that a copy of every object needed is CHECKed OUT into a private database. These private copies can be modified in an arbitrary way. Only after all modifications have been performed the modified objects are restored into the database by CHECKIN. Long-living locks are imposed on the objects checked out from the public database. Other transactions can read these objects in the state before the checkout.

The CHECKOUT/CHECKIN model offers the following advantages:
- the object can be read in its former state in parallel to an user's activity.
- the modification of an object is performed in a private space, perfectly invisible to other users.
- the only critical phases with respect to the concurrency control of the whole transaction are the operations CHECKOUT and CHECKIN.
This idea was extended in /BakK85/. They distinguish between a public and several private databases in which objects are checked out from the public database. Every private database is associated with a conversational transaction. It provides shelter for a number of conventional transactions running on the data of the private database. The reason for the introduction of conventional transactions is to define a finer granularity for recovery. The unit of consistency is the conversational transaction, while the unit of recovery is the conventional transaction.

A similar model was presented in /KaWe83/ or /Kat82/. For recovery reasons they don't use conventional transaction but introduce savepoints. These proposals already take the long duration and interactivity of design processes into consideration. Therefore, they are fully appropriate when users work in an isolated manner on objects of the public database. However, they do not sufficiently support those (many) environments where users do work cooperatively on a common set of objects. /KLMP84/ remove this deficiency by the introduction of a hierarchy of transactions.

Since /BaAK85/ refine this proposal we will only discuss their model here. A design environment consists of a number of independent project transactions, which are protected against each other like conventional transactions. A project transaction consists of a number of cooperative transactions. Each cooperative transaction is responsible for a (complex) subtask. To fulfill this job a cooperative transaction can be subdivided into several client/subcontractor transactions (C/S transaction). Each C/S transaction deals with a well-defined part of the original subtask. If this part is still very complex a C/S transaction is allowed to form a hierarchy of C/S transactions. Finally, each C/S transaction is divided into several designer's transactions each of which is associated with exactly one designer. A designer fulfills his task by starting a number of conventional short-duration transactions. The way transactions are associated with each other is very similar to the model of nested transactions presented in /Moss81/ or /Moss86/. Although cooperative work is supported to a certain degree the transaction model still has weaknesses:

Usually a complex design process is subdivided into a (great) number of small subtasks which are executed by one or a couple of designers. But these small subtasks are still long duration subtasks. To modify a chip will take days or even weeks but not seconds. That means that a designer's transaction will work on a (maybe) small number of objects but for a longer period of time without wanting these objects to be released and thereby visible to other designers (even of the same design team). Therefore, from our point of view, this kind of splitting might cause some problems for the user especially because objects might be available to other designers between such short transactions. As one reason for providing this concept the need for an introduction of a formal model of consistency was mentioned. Without any doubt the proposed model provides an elegant solution, but, from our point of view, only on the formal level. The pure splitting of long duration transactions into a number of short duration transactions doesn't really reflect the way designers want to work. The notion of transaction and the notion of consistency are tightly coupled. Therefore, we think that designers' transactions should still be long term transactions which should be treated as such.

Another drawback of this approach is that it doesn't support an appropriate recovery mechanism. For example, it is allowed to back up a C/S transaction. As a consequence all already committed short duration transactions of the project transaction are backed up, too. Since these transactions might already have released some updates (even in the public database) a rollback propagation might occur.

2. New requirements for the transaction model

One of the major problems of new database applications is the provision of an appropriate environment for the complex and lengthy design process. Within such a process a number of users work for weeks or months on a comparatively large, common set of objects. The work which has to be performed can be divided into individual units which only require a subset of the objects.

The set of users is divided into groups. Each group covers an area of responsibility. To solve problems groups are further subdivided into subgroups, etc., until, at the end of the line, one or a couple of designers are being engaged in the solution of a special problem. Each user works on a well-defined part of the problem within the environment. The members of the group work cooperatively, rather than competitively, they coordinate their work. In particular, the users know to what degree their work overlaps and whether they may proceed in parallel without endangering the consistency of the common data or not.

An environment of this kind, therefore, requires not only synchronisation of the users' transactions - as is usual in contemporary systems - but, moreover, it has to support cooperation within groups of users.

The groups altogether form a tree-like hierarchy. Groups which are located beneath another group but on the same direct path to the root are called subgroups and define specialisations of the work of the superior group. Subgroups are tightly coupled with the (unique) superior group. On the other hand, groups of different branches of the hierarchy are usually involved in different problems. Therefore, these groups are only loosely coupled. Of course, cooperative work between tightly coupled groups has to be supported in a more extended way than cooperation between loosely coupled groups.

The system should enable the users to work with the flexibility they have today, which means, that the way of reasoning must be how this flexibility can be offered but with sufficient safety and extensive support. Especially, it should be able to influence the setting and releasing of locks to cooperate and communicate in a more flexible way with other members of the group. Of course, if the users make use of these possibilities, the transaction management cannot completely guarantee consistency within a cooperative transaction in the sense of serializability of transactions.

In the next sections a transaction model will be introduced
* which allows to reproduce the above mentioned complex hierarchies within large design divisions and
* which is capable to support the different kinds of cooperation between differently coupled groups.

Other essential concepts of non standard databases like e.g. versions and alternatives or a flexible synchronisation mechanism will not or only rudimentarily be discussed. The interested reader is referred to the following (exemplary) literature:

version concept: /DiLo86/, /KaCB86/, /KiSW86/
synchronisation: /GaKi88/, /UnSc88/

3. The multi-level transaction model

The transaction model has several layers (see figure 1): Firstly, the users are to be shielded from the outside. This is done by the database transaction (DBT). This type of transaction realizes the one and only gateway of the design team to the public database. Protected by the DBT the design process can take place in a more convenient and suitable atmosphere. For this purpose an arbitrary number of (hierarchically organized) group transactions (GT) can be established which serve the individual groups as private databases. The real work on objects, however, is done in user transactions (UT) which are always leaves of the group hierarchy tree. Of course, user transactions as well as group transactions allow a controlled cooperation on common objects within the group and (in a more restricted way) between different groups. This paper is a refinement of the two-level transaction model presented in /KiSUW85a/.

The unique direct path from a transaction T to the public database will be called the direct path of T in the remainder.

The following holds:

1. The set of all transactions forms a strict tree-like hierarchy. The database transaction acts as the root and the user transactions represent the leaves. All inner nodes are group transactions. Of course, during the construction of the tree a group transaction may also be a leaf. But since a group transaction is not allowed to manipulate objects (see (3)) it makes no sense to create a group transaction without any child transaction. The tree is built up from the root to the leaves and taken down in reverse order.

2. For each transaction T an object pool is established. The object-pool is a temporary database in which all objects necessary at that level are checked in. It exists only as long as T exists. It represents the working environment for the group. DBT and all GTs only serve as data pools. UTs, additionally, allow to operate on objects. An object pool can logically be subdivided into several subpools each of which includes the objects of exactly one child transaction. This view allows to easily assign objects to child transactions. The only way how transactions can acquire objects is by the CHECK-OUT command.

3. Each transaction is associated with exactly one user group whereby the term group includes single user groups.

4. GTs and UTs operate in parallel to each other, they are well-formed and - unless the user intervenes - two-phase. But they allow users to influence the concurrency control.

5. Transactions are object-oriented in the sense that they realize the principle of information hiding. Only members of the group which is associated with the transaction are allowed to operate on objects of the object pool. Non-members can't operate on these objects. They have to communicate with the transaction by the sending of messages. The transaction can either reject or comply with the request. If the requested objects are not available in the object pool the transaction can send a message to its superior transaction in order to get the missing objects. Therefore, the requesting user is unaware from which pools the objects are checked out.

6. By default every user is allowed to see every object at any time which is in the public database or in any object pool on the direct path even if it is checked out by another transaction (dirty read). But the object won't be copied into the user's object pool. Instead, it remains in its original place. The user has to realize that this kind of read is a dirty read.
Outline of a design process

Let a design team be the set of all users who are involved in a design process. For this team, as the root of their design environment, the database transaction (DBT) with its common object pool has to be installed. Under shelter of the DBT an arbitrary number of group transactions (GT) can be established. Each group transaction provides the object pool for the level it shall represent. The DBT as well as all GTs only serve their groups as a private data pool. No update-operations on objects are directly performed within such transactions. Modifications are completely performed in user transactions (UT).

In the following an (informal) outline of a design process will be presented. The syntax used is exemplary and shall help the reader to realize the semantics of the commands (I I comprises alternatives, [ ] describes optional parts of a command).

In our model every subject can uniquely be identified either by a system defined identifier or by a user defined name. For reasons of simplicity we will only use the identifier in the remainder although the use of the name is always possible, too.

Definition of groups

As already mentioned a design team is hierarchically subdivided into a number of groups which work with an arbitrary number of transactions. The mapping between the transactions and the user groups is a function in which the transactions represent the domain and the groups the range. Transactions are associated with exactly one group. Groups can be related to none up to an arbitrary number of transactions. Therefore, firstly user groups have to be defined:

CREATE USER GROUP <group_name> WITH MEMBERS "user_id_list".

User_id_list can include a special user identification as well as a group identification which means that the group as a whole is part of the new defined group.

The structure of a group can be modified at any time - even during the run-time of an associated transaction - by:

CHANGE USER GROUP <user_id> [ADD | REMOVE]
"user_id_list".

By DELETE USER GROUP <user_id> a user group can be deleted. This command is only allowed if all transactions associated with the group have terminated.

Creation of a transaction

In the following we will sometimes use the terms view of a designer or accessible objects. By this we always mean the default view or the default access right to an object. No attention is paid to the fact that this view or access right can be restricted by the access control component.

Any designer can start a new transaction in his name or in the name of a group he is member of. A new GT or UT is always a child transaction (CT for short) of an already existing transaction (the father transaction (FT for short)). Therefore, the transaction tree is built up from the root to the leaves (top-down). Let T be a transaction in the hierarchy. Each transaction on the direct path from T is called subordinate to T while each transaction on a lower level which can be reached on a path starting at T is called superior to T. All superior transactions of T form a path, the direct path, while all subordinate transactions form a (sub)tree with T as its root (called subtree of T in the remainder).

The following command allows to start a new transaction:

CREATE_TRANS <trans_name> [FOR <group_id>]

[AS_CHILD_OF <trans_id>] AS DBT

Each transaction is associated with exactly one group which will be called associated group in the remainder. For reasons of simplicity we will call members of the associated group of a transaction T members of T. The associated group includes all members of that group that is declared in the FOR part. As already mentioned, the transaction model is in a sense object-oriented. Only members of the associated group have access to the interior of a transaction T.

To assure that all members of T have also access to all subtransactions of T they are implicitly added to the associated group of any child transaction of T. Since this definition is recursive each member of T is implicitly also member of all transactions in the subtree of T. Of course, it is possible to explicitly remove any user from any associated group.

Let T be a transaction a designer D is currently working with. The most comprehensive transaction of D with regard to T (called top transaction of D for T) is that transaction T which is located on the direct path of T and which fulfills the following conditions:

* D is member of T
* D isn't member of any transaction on the direct path of T.

In other words, the top transaction of D for T is that transaction on the direct path of T which is located nearest to the DBT (or the DBT itself). Of course, the top transaction of D may change if T changes.

If the FOR part of the CREATE_TRANS command is omitted only the user who has delivered the command represents the explicit part of the associated group.

If the AS_CHILD_OF part is omitted the new transaction represents a database transaction (DBT). In this case the AS part is redundant.

<trans_id> must not identify an UT since it is not allowed to create a transaction as a subtransaction of an UT.

CREATE_TRANS not only starts a new transaction but also creates the object pool. Additionally, a new logical subpool is added in the object-pool of the FT (if the new transaction is not the DBT). The name of this subpool is identical to the name of the CT. The subpool serves as an input area for the CT. Not only all objects checked out by the CT are registered here
but also those which are considered for the CT but which were not explicitly checked out by the CT yet.

Thus, the object pool is logically subdivided into a number of input areas, one for each child transaction, and a general area (called common pool). The common pool includes all objects which are still available. That are all objects which are not assigned to any CT yet as well as objects which are already locked by at least one CT, but only in a shared lock mode. Of course, objects with shared lock mode may occur in several subpools.

The following commands can be delivered by any member of a transaction T to add or remove objects to or from a subpool:

ADD <object_id_list> TO <subpool_id>
REMOVE <object_id_list> FROM <subpool_id>

As already mentioned the subpool name is identical to the transaction name of the CT.

The REMOVE command is only allowed if the appropriate objects are not checked out by the CD yet.

A subpool only serves as output area for a CT. Modified objects which shall be restored into the FT are checked in into the common pool of the FT.

Every transaction T can only be active on behalf of one member of the associated group at a time. The user who has control of T can perform any command associated with T until he explicitly suspends the transaction by the SUSPEND command:

SUSPEND <trans_id> suspends the transaction <trans_id> currently active for the user.

CONTINUE_TRANS <trans_id> reactivates the specified transaction.

A user may run several transactions at the same time. This enables him to work on different objects simultaneously, and to use transactions alternatively.

Different types of access

During a design process a lot of objects are needed for different reasons. The most important objects are those which are the basis of the design process. They have to be accessible during the complete process or, at least, for a considerably long time. Another group of objects are those which are only needed during a relatively short period. Examples for this type are objects which are used for the preparation of a decision or which were considered in a selection process but which were excluded. Mostly, these objects have only to be read; often a dirty read is sufficient. A third group are those objects which are accessed in a rather conventional way, for example administrative data.

In order to support these different kinds of demands three types of access are offered:

(1) Check out / Check in

This mechanism is appropriate for all operations which have to work for a very short time on a consistent set of objects. It makes use of the conventional lock protocol.

(2) Dirty read

A dirty read allows to read any object at any time even if it is locked exclusively. Therefore, this type of access is appropriate for each read which doesn't require a fully consistent view of the objects read (e.g. browsing).

(3) Conventional access

This kind of access is appropriate for all operations which have to work for a very short time on a consistent set of objects. It makes use of the conventional lock protocol.

Check out / check in

This mechanism is the one and only by which objects can be added to any object pool of any transaction. By a check out a long-living lock is imposed on the object which can be released by the appropriate check in at the earliest. In the following we will describe the syntax of the command. We won't consider any access limitations. This will be made up in the section about views below.

CHECK_OUT <object_id> [FROM <trans_id> WITH lock_mode]

BASE

CHECK_IN <object_id> [IN <trans_id>]

With the help of a check out command a member of a transaction T checks out an object from an object pool (or the public database) and copies it into the object pool of any other transaction. As already mentioned the object is at least logically included into each object pool of any transaction on the direct path between <trans_id> and T. A check out always transfers an object from one level to a deeper level (from a superior to a subordinate transaction) while a check in transfers it to a higher level (from a subordinate to a superior transaction).

We can distinguish between two types of check out: a leaving one and a coming in. Let T be a transaction a designer D is currently working with. The check out is coming in if T wants to get an object O from any superior transaction. In this case the FROM part can be specified while the TO part has not to be used since the destination transaction is T itself. O is copied from <trans_id> to the common pool of T. If the FROM part is omitted then an object is requested from which the location is unknown. If O is part of the current access view of D the request will be granted if O is not locked in an incompatible mode. Otherwise, two possibilities exist. If O is located in an object-pool of any other branch the request is rejected. In the other case, that is, if O is located on the direct path of T the request will be granted if O is not locked in an incompatible mode. Otherwise, the request moves up, whereby each transaction on the path has to agree. If any transaction does not comply with the request it is rejected. To reduce overhead it is possible to lay down default behaviour (e.g. reject/agree in any case).

By a leaving check out an object of the object pool of T is copied into the object pool of any subordinate transaction. Since now the initial transaction is fixed the FROM part has not to be used.

A check out coming in always copies an object of a higher level into the object pool of T while a leaving check out copies an object of the object pool of T to a deeper level transaction.
Each object is transmitted from up (e.g. public database) to down (e.g. UT). In this process the object O is copied in any object pool on the way forming at least a continuous path (exclusive lock) or tree (shared lock). Since, therefore, a lot of copies might exist the question arises which transaction has which rights on O. The answer is that the rights on O depend on the type of lock. An exclusive lock only allows one transaction to modify O or to move O further: the most subordinate transaction which possesses O. Therefore the right to move or modify an object moves down with the object. Since the object can only be moved to one child transaction a path is formed. In case of a shared lock an arbitrary number of transactions are allowed to use the object. In this case every transaction which houses the object in its object pool is allowed to move it to as many CTs as necessary. Now, a tree is formed.

A transaction which is allowed to move (and to modify) an object O is called owner of O.

Since in our model transactions are independent a 'long distance' check out shouldn't be possible without the permission of every transaction Ti involved. To satisfy this requirement it has to be fulfilled that the user who delivers the checkout command is member of every Ti. In case of an leaving checkout this is, by default, always fulfilled. In the other case the designer has to be member of the starting transaction T* (specified in the FROM part) and every transaction on the direct path up to the demanding transaction T.

The check in command is very similar to the check out. It is always upward. The user checks in an object from the transaction T he is currently working with into the object pool of the father transaction (default value if the TO part is omitted) or the transaction specified (trans_id). Of course, a check in also returns the lock and is therefore only allowed if all subordinate transactions have already returned the object to T.

Dirty read

Each designer D is allowed (by default) to read every accessible object at any time in an unprotected mode. If D is currently an active user of transaction T then the accessible objects of D with regard to T are all those objects which are part of any object pool on the direct path or which belong to the public database. Objects in different branches of the tree are not accessible. A dirty read only allows to read the object. It is not copied into the object pool of T. The user has to be aware of the fact, that he might see objects in an inconsistent state.

The access view of a designer

According to his function in the design team each designer has an individual view on the set of all objects accessible for him (called access view). In this context the right to access an object means to have the permission to check out the object in any mode currently available or to check it in.

The access view depends on the transaction T the designer D is currently working with. If T is a user transaction then the view of a designer is established by the object pool of T and the appropriate subpool of the father transaction. Additionally, a designer has access to the object pool of every transaction on the direct path up to the top transaction of D for T. As an active user of a transaction T he doesn't have direct access to object pools of transactions which don't belong to the direct path of T even if he is member of these transactions.

If T is a group transaction (or the DBT) then D additionally has access to the subtree of T (by default).

The public database is generally accessible for everybody.

Therefore, the upward part of the access view of D always forms a path (and includes the public database), the downward part forms a tree.

Savepoints

Although we don't want to consider recovery mechanisms in more detail a short discussion seems to be necessary.

In our model we support the concept of savepoints (for further details see KSUW88b). Savepoints serve two different aims:

* they describe a checkpoint at which the system can restart in case of a system crash.

* they constitute a state to which a user can voluntarily reset the transaction (or object modifications).

We can distinguish between two different kinds of savepoints: strong savepoints and weak savepoints.
A strong savepoint sets up an irrevocable state. Neither the system nor the user are allowed to reset a transaction beyond this point. The only way to attain to any further state is to start a compensation operation (transaction).

Weak savepoints, however, allow a voluntary reset to a state beyond them. They only serve the operation (transaction) to any further date is to start a compensation transaction beyond this point. The only way to attain to any further date is to start a compensation

To support the cooperation a transfer of objects between transactions is supported which belong to different branches of the transaction tree. Generally, only a transaction which is owner of an object is allowed to transfer it. Additionally, the current state of the object has to be protected by a strong savepoint. By this rollback propagation is prevented.

Transferring objects between transactions

To support the cooperation a transfer of objects between transactions is supported which belong to different branches of the transaction tree. Generally, only a transaction which is owner of an object is allowed to transfer it. Additionally, the current state of the object has to be protected by a strong savepoint. By this rollback propagation is prevented.

Lending of objects

For a number of reasons it might be desirable to let a user transaction UT1 work on an object O for a certain period of time, although this object is exclusively locked by a transaction T2. As long as UT1 is member of the subtree of T2 such a conduct does not cause any problems. But, in case this is not fulfilled (e.g. T2 is a user transaction, too) a new mechanism has to be introduced. For this purpose we suggest the concept of lending of objects. With the lending of an object O to a user transaction UT1, all rights are assigned to the borrower UT1, except the right to check in the object into the object pool of any other transaction. As soon as UT1 has finished its work, it returns the control over O to the owner of the lock, i.e. to T2. This concept allows an alternating work of UT1 and T2 on O, with either UT1 or T2 working at one time.

The lender T2 uses the operation

GRANT <object_id> TO <trans_id>

Grant makes the object O to be lent available to the transaction <trans_id> (here: UT1) in its present state. But, O is not directly copied into the object pool of UT1. Instead, it is copied in the appropriate subpool of UT1's father transaction. UT1 can copy the borrowed object into its object pool by the CHECKOUT command.

When UT1 has finished its work, it releases the modified object by

RETURN <object_id>

for the original transaction. Again, O is not directly copied into the object pool of T2 but into the appropriate subpool of T2's FT. T2 can fetch back the object, by a check out into its object pool.

Transfer of objects

If the owner transaction T1 of an object O is not interested in getting the object back later (e.g. another transaction continues its working unit), it can transfer it to another transaction T2. The receiver transaction subsequently may work with the object at will, especially, it may check it in.

The transfer is realized by:

PASS <object_id> TO <trans_id>

The PASS command is not allowed in any case. Instead, two conditions have to be satisfied. Let D be the designer who wants to pass O from T1 to T2. Let TR be the root of the smallest subtree which contains T1 as well as T2. To pass O from T1 to T2 means to remove O from every subpool on the direct path between T1 and TR and to add it to every subpool on the direct path between TR and T2. This is only allowed if

* T1 is part of the access view of D with regard to T2
* every transaction on the direct path between TR and T2 permits the insertion of O in its object pool.

Communication system

Let D be a designer who is currently working with transaction T1. If an object O is owned by a transaction T2 in a different branch of the transaction tree it is, by default, not accessible for T1 since it doesn't belong to the access view of D with regard to T1. If D nevertheless wants to access O he has the possibility to ask a member of T2 whether he is in the position to pass the object to T1. In the same way D can ask for a dirty read. For this purposes a communication system is installed which allows every user to communicate with every other user or transaction. By this requests like the one above can be handled. Every member of a transaction is able to comply with or reject any request automatically or to make an individual decision.

Information system

In order to explore all possible avenues of cooperative work inside and between groups the user has to be in a position to get all information he needs. For this purposes a comprehensive information system is available. Among others this system enables the user to get the following informations:

* a list of all groups he is member of
* a list of all transactions which are associated with him or with a special group
* a list of the members of a special group
* a list of all objects which are being included in a given pool
* an overview of the present transaction tree
* the information which transaction currently owns an object

Information board

Every object O which is checked out for modification or which was created by any transaction is supplied with an information board. Every owner O can stick information e.g. about the actual state of O or about recommended subsequent operations on the board. Therefore, this board is an important component which supports the users in their effort to work on the object in a consistent way.

EOT processing

Any transaction can be finished by the END_TRANS <trans_id> command. When the group or designer has reached its aim, the result (as well
as any intermediate state the user wants to make visible to others) are written into the object pool of the FT (for the public database in case of the DBT).

Of course, this command is only allowed if all subordinate transactions have already finished their work.

The following principle is applicable to objects borrowed or checked out:

Only the objects which are rewritten explicitly (RETURN), survive the end of the transaction. END,TRANS will be rejected, since the objects lent could not be written into the FT.

Every object the transaction has borrowed of other transaction and which has not been returned yet, is returned at END_TRANS. In this case the performed modifications are void, and the object is returned in its original state.

Concurrency control aspects

Concurrency control aspects are an essential part of a transaction model. Nevertheless, we don't want to handle them here. The interested reader is referred to /UnSc88/. In this paper

* a number of necessary lock modes are introduced
* the effect of a lock is split into the effect for the owner of a lock and into the effect for a transaction which requires the locked object.

The integration of both considerations lead to a general lock mechanism which allows to adapt concurrency control to the special needs of most applications. From our point of view it is essential to offer a variable concurrency control mechanism which can easily be adapted to all kinds of demands.

4. CONCLUSIONS

This paper introduced a transaction concept tailored to the requirements of design environments, in particular the support of cooperative work. The transaction model allows to naturally reproduce a design process. A database transaction acts as the basis for the design process. It shields the design team from the outside and, therefore, offers the possibilities for a convenient and suitable atmosphere in which the designers can work in a cooperative way.

A first prototype of the transaction model was implemented on top of the ORACLE relational database system. It is our intention to integrate this concept into the (non standard) database kernel system DASDBS which was developed and implemented at the university of Darmstadt /PSSW87/.

Literature

/Gray78/ Gray, J.: Notes on Data Base Operating Systems. from 'Operating Systems, R. Bayer, R. Graham, G. Seegmüller (Editors), Springer Verlag, 1979
/Moss86/ Moss, J.E.B.: An Introduction to Nested Transactions, COINS Technical Report 86-41, University of Massachusetts at Amherst, 1986