A Cooperative Transaction Model Handling Multiple Correctness Levels

Haiyan XU
Department of Electronics, Fukuoka Institute of Technology, Fukuoka, Japan

Abstract: In database systems, in order to ensure that the simultaneous operations of users do not interact in undesirable ways, the concept of transaction management is introduced. In design database systems, however, since Computer-Aided Design (CAD) processes are very complicated, a transaction cannot be simply treated as a signal execution of a program as done in conventional database systems. Furthermore, since it is required to store incomplete data as well as the data which only passed a subset of tests in design database systems, multiple levels of correctness must be permitted. In this paper, we introduce a transaction model which makes the properties of transactions in design databases clear: transactions of a design project should be defined according to the correspondence between the multiple correctness levels of the project and the designer group hierarchy of the project. By defining a view for the design data satisfying one correctness level, we show that a reasonable unit of a transaction is a set of operations which can be regarded as a primitive operation in some view. We also provide mechanisms for transaction management and show that they can be used to decrease the burden of designers on the transaction management.

1 Introduction

The process of Computer-Aided Design (CAD), such as VLSI circuit design and software engineering, requires handling of large amounts of data. In principle, a database management system can help to manage these data by offering data sharing, preserving data integrity and allowing convenient access. Conventional database systems, however, were principally designed for business applications and have been found to be inadequate for addressing the requirements of these new application domains [Bern87]. This paper deals with one of the major problems of new database applications, the transaction management in cooperative design database systems.

Three major characteristics of CAD activities are as follows:

1) Design processes are complicated.
2) There are multiple correctness levels.
3) Designers cooperate hierarchically.

Each of these characteristics introduces difficulties when constrained by traditional transaction properties. Most conventional database systems allow numerous applications run simultaneously. It is necessary to ensure that these applications do not interact in undesirable ways. The concept of transaction management is intended to solve such problems, where a transaction is a single execution unit of a program. Since conventional database systems impose a set of integrity constraints on all the data in the database, its transaction model was defined for one correctness level. In design database systems, however, due to the complexity of design processes, first it is necessary to treat an execution of one or more programs as one transaction. These programs may be CAD tools, or queries expressed in some query language of design database systems. Secondly, since it is required to store incomplete data as well as the data which only passed a subset of tests in design database systems, multiple levels of correctness must be permitted and its transaction model must be defined for multiple correctness levels, too. Finally, since a large design project is generally carried out by a hierarchy of designer groups, it is necessary to discuss the problem of how to define the transactions of each group properly.

One representative model of CAD transactions, tries to solve these difficulties using both transaction hierarchy and database hierarchy [UnlS89, KSUW85, KaCB86]. As shown in Figure 1, only completely tested design data can be stored in the public database, which are shared with designers. When a group of designers wishes to complete a design cooperatively, the group has to define a group transaction for the design. For each transaction, a temporary database is
automatically generated to represent the working space for
the designers involved. According to a hierarchy of designer
groups, a transaction hierarchy is created. The operations
of a transaction are carried out on the temporary database
of the transaction and all data necessary for the transaction
must be copied into it. Before committing a transaction, con-
versely, it is necessary to copy its result to the database of
its parent transaction, since the temporary database of each
transaction exists only as long as the transaction exists.

![Diagram]

Figure 1: Method of conventional design databases

In such a way, this approach makes incomplete results
of a designer visible to the other designers in the same group.
However, it does not give designers any rule to define trans-
actions properly and designers can lock arbitrary numbers of
objects by a transaction.

For a design project, it is necessary to manage design
data satisfying various evaluation standards (or correctness
levels) and give designers rules to define transactions prop-
erly. If the transaction is defined too small, a task may be
separated into several ones and the other operations
may be interleaved among them. On the other hand, if the transac-
tion is defined too large, clearly some reasonable cooperative
work may be prohibited, which may result in a less degree
of concurrency.

In this paper, we make the properties of transactions
in design databases clear: a transaction should be defined
to consist of a set of operations which can be regarded as a
primitive operation in some correctness level. Since each de-
signer group of a large design project is in charge of the task
in certain correctness level of the project, designers should
define their transactions according to their tasks. We de-
fine a transaction model which can represent various kinds
of cooperative works flexibly and use view functions to give
an intuitive image of transactions. We also show that design
database systems can check the propriety of the transactions
based on the plan made out by designers. Design database
systems can also prompt designers to define or to commit
some transactions. By such a way, the burden of designers
on transaction management can be decreased.

The paper is organized as follows: In Section 2, basic
properties of design data and design operations are discussed.
In Section 3, we provide a method to define transactions
properly and give an intuitive image of transactions. How to
manage transactions in cooperative design database systems
is discussed in Section 4.

2 Preliminaries

Design database consists of a set of objects, together with
relationships on them. Objects are used to identify useful
aggregates of design data without committing to an imple-
mentation in term of records, tuples or files. Since the objects
can be classified by their functions, interfaces and imple-
mentations [BatK86], we use a tuple of (Component-name,
Version-name) to denote an object. The objects with an
identical interface and function are assigned to an identical
Component-name, and the term 'component' is used to refer
to such a set of objects. Furthermore, since design processes
are carried out stepwise, even incomplete results are required
to be stored in design databases. In this paper, we assume
that objects contain data and their design status.

Relationships record correspondence between objects or
components. We will consider the following two typical ones
in this paper:

1. History relationships (between objects):
   \[ R_H = \{ ((C_i, v_k), (C_j, v_l)) \mid (C_j, v_l) \text{ was derived from} \ (C_i, v_k) \} \]

2. Aggregate relationships (between components):
   \[ R_A = \{ (C_i, C_j) \mid C_j \text{ is a subcomponent of } C_i \} \]

If \((C_i, v_k), (C_j, v_l) \in R_H\), there is a history relationship
from \((C_i, v_k)\) to \((C_j, v_l)\). If \((C_i, C_j) \in R_A\), there is an aggre-
gate relationship from \(C_i\) to \(C_j\). Each \((C_i, C_j) \in R_A\) means
that \((C_j, v_l)\) may be used as a subcomponent of \((C_i, v_k)\), for
any pair of objects \((C_i, v_k)\) and \((C_j, v_l)\). Whether \((C_j, v_l)\) is
used as a subcomponent of \((C_i, v_k)\), however, will be deter-
mined by designers according to the result of tests or simu-
lations.

Design data also exist several representation levels. For
example, a VLSI design is specified in various ways such
as functional description, gate, transistor, and layout levels.
However, when objects in the highest level are modified, all the ones depended on them should be changed, too. For simplicity, we will treat a set of dependent objects in different representation levels as a single object.

Since both history and aggregate relationships are binary relationships, they can be represented by directed graphs. History relationships between objects can be expressed by the graphs whose vertices represent objects and edges represent history relationships. Aggregate relationships between components, on the other hand, can be expressed by the graphs whose vertices represent components and edges represent aggregate relationships. Figure 2 shows a part of a design database, where,

1. \((CHIP, v_1), (ALU, v_1), (RAM, v_1), (Q-SHIFT, v_1), \ldots, (RESTC, v_1)\) are objects,
2. \((CHIP, v_2), (CHIP, v_3)\) and \((CHIP, v_4)\) are objects of the component \(CHIP\) which have an identical interface and different implementations,
3. \(((CHIP, v_1), (CHIP, v_2)), ((RAM, v_1), (RAM, v_1)), \ldots\) are history relationships between objects, while \((CHIP, ALU), (CHIP, RAM), \ldots\) are aggregate relationships between components.

Figure 2: Structure of design data

For the graphs of aggregate relationships between components, since only the strongly connected subgraph which form a hierarchy and has one root can be tested or simulated together, in the following discussions we will call such a set of components as a component hierarchy. Let \(P\) be a component hierarchy, a set of objects is called a configuration of \(P\), if it contains one and only one object of every \(C_i \in P\). Let \(F\) be a configuration of \(P\), object \((C_i, v_k) \in F\) is called the root object of the configuration \(F\) if \(A(C_j, v_l) \in F\) for any \((C_j, C_i) \in R_A\). In the same way, object \((C_i, v_k) \in F\) is called a leaf object of the configuration \(F\) if \(A(C_j, v_l) \in F\) for any \((C_i, C_j) \in R_A\). For example, in Figure 2, \(P_1 = \{CHIP, ALU, RAM, CONTROL\}\) is a component hierarchy and \(F_1 = \{(CHIP, v_1), (ALU, v_2), (RAM, v_2), (CONTROL, v_2)\}\) is a configuration of \(P_1\), where \((CHIP, v_1)\) is the root object of \(F_1\) and \((ALU, v_2), (RAM, v_2), (CONTROL, v_2)\) are the leaf objects of \(F_1\). \(\{CHIP, ALU, RAM, DESTC, Q-SHIFT, RAM-SHIFT\}\), however, is not a component hierarchy.

The operations in CAD environments, on the other hand, according to their influence to objects, can be classified into the following six basic ones:

1. \(W(C_i, v_j)\): modification of an existing object \((C_i, v_j)\).
2. \(D((C_i, v_j), (C_k, v_l))\): modification of an existing object \((C_i, v_j)\), where the result is stored as a new version \((C_k, v_l)\).
3. \(R(C_i, v_j)\): reading of an existing object \((C_i, v_j)\).
4. \(X((C_i, v_j), \ldots, (C_k, v_l))\): executing of existing objects \((C_i, v_j), \ldots, (C_k, v_l)\) together.
5. \(B((C_i, v_j))\): browsing of an existing object \((C_i, v_j)\).
6. \(C((C_i, v_j), (C_k, v_l))\): copying of an existing object \((C_i, v_j)\) into \((C_k, v_l)\) for the purpose of independent manipulation.

The compatibility of the above operations is showed in Figure 3. The rows correspond to the operations being requested, and the columns correspond to the operations already being carried out. The entries are \(Y\) (yes, they can be carried out simultaneously), and \(N\) (no, they cannot be performed at the same time). For example, suppose several write operations are carried out on an identical object at the same time, the result of the write operation which finished earlier will be overwritten by the later ones. For this reason, the compatibility of the write operations is \(N\). Since browsing and copy do not require cooperation management, in the following discussions we will focus on the first four ones.

<table>
<thead>
<tr>
<th></th>
<th>Write</th>
<th>Derive</th>
<th>Read</th>
<th>Execute</th>
<th>Browse</th>
<th>Copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write</td>
<td>(N)</td>
<td>(N)</td>
<td>(N)</td>
<td>(N)</td>
<td>(Y)</td>
<td>(Y)</td>
</tr>
<tr>
<td>Derive</td>
<td>(Y)</td>
<td>(Y)</td>
<td>(Y)</td>
<td>(Y)</td>
<td>(Y)</td>
<td>(Y)</td>
</tr>
<tr>
<td>Read</td>
<td>(Y)</td>
<td>(Y)</td>
<td>(Y)</td>
<td>(Y)</td>
<td>(Y)</td>
<td>(Y)</td>
</tr>
<tr>
<td>Execute</td>
<td>(Y)</td>
<td>(Y)</td>
<td>(Y)</td>
<td>(Y)</td>
<td>(Y)</td>
<td>(Y)</td>
</tr>
<tr>
<td>Browse</td>
<td>(Y)</td>
<td>(Y)</td>
<td>(Y)</td>
<td>(Y)</td>
<td>(Y)</td>
<td>(Y)</td>
</tr>
<tr>
<td>Copy</td>
<td>(Y)</td>
<td>(Y)</td>
<td>(Y)</td>
<td>(Y)</td>
<td>(Y)</td>
<td>(Y)</td>
</tr>
</tbody>
</table>

Figure 3: Lock type compatibility matrix

3 Transaction Model

In this section, we first summarize features of cooperative works for design applications. Then we formally define what
we mean by a transaction and give an intuitive image of such a transaction.

Let us use an example to indicate the features of cooperative design works.

**Example 3.1** Suppose designers A, B, C, D, E, F plan to design a computer. In order to fulfill the design in time, the design is classified into a component hierarchy (Figure 4 (a)) and each designer is in charge of the design of a part of components: Designer A is in charge of the design of components CU, CPU and COMPUTER, designer B is in charge of the design of ALU, designer C is in charge of the design of RAM, designer D is in charge of the design of MEMO, designer E is in charge of the design of IOC, IOD1, designer F is in charge of the design of IOD2.

The results created by designers, however, need to be tested together. For this reason, designers will form a hierarchy (Figure 4 (b)) and each group works to fulfill certain evaluation standard: Each leaf group will create objects which can pass its test separately; Each nonleaf group will test the objects created by its child groups together.

![Figure 4: An example of cooperative work](image)

This example shows that a large design project is typically partitioned into a hierarchy of designer groups. Each designer group is in charge of the design of a part of components under certain evaluation standard. That is, for a hierarchy of designer groups there is a hierarchy of evaluation standards and each designer group works to fulfill certain evaluation standard. In general, design objects need to pass several kinds of tests or simulations. For example, a VLSI circuit may be tested by static logic test, logic simulation, timing simulation and analog simulation. Furthermore, even if several objects pass a test separately, it does not necessarily mean that the configuration constructed by them can pass the same kind of test. For this reason, the evaluation standard is defined in the following way:

**Definition 3.1** An evaluation standard $S$ is characterized by $S = \{ (E_j, P_j) \mid E_j$ is an evaluating method, $P_j$ is a component hierarchy $\}$. Evaluation standard $S$ means that configurations of $P_j$ will be evaluated by the method $E_j$ ($j = 1, 2, \ldots$). Components of $P_j$, furthermore, are called the components evaluated by $S$.

![Figure 5: Hierarchies of a cooperative project](image)

In this paper, we will consider the cooperative works in the following kinds of cooperative projects:

**Assumption 1:**

1. There is a tree hierarchy of designer groups (Figure 5 (a)). The designer groups in the leaves of the tree contain a single designer and designers of a nonleaf group should be included in the union of the designers of its child groups.

2. There is also a tree hierarchy of evaluation standards (Figure 5 (b)). For each nonroot evaluation standard $S_i = \{ (E_{ij}, P_{ij}) \}$, the components evaluated by $S_i$ will also be evaluated by its parent evaluation standard $S_k$.

3. The two hierarchies, furthermore, are not independent:

   For each nonleaf group $G_i$, there is a corresponding nonroot evaluation standard $S_i$, which expresses the design objectives of $G_i$'s child groups. The root evaluation standard represents the final design objective of the project.

That is, multiple evaluation standards of a project is fulfilled by a hierarchy of designer groups stepwise. In Example 3.1, if designers test the design by one kind of evaluating method, the hierarchies of evaluation standards and designer groups are indicated in Figure 6. Figure 7 shows another case that the design is tested by two kinds of evaluating methods. That is, Assumption 1 does not restrict the cooperative works among designers, it only expresses the property of cooperative works explicitly; designers can define their required designer group hierarchy and evaluation standard hierarchy.
flexibly.

Figure 6: The hierarchies of designer groups and evaluation standards of Example 3.1

As shown in these figures, the correspondences between the hierarchies of designer groups and evaluation standards define the cooperative works among designers. On the other words, the access right of each designer group will be restricted by the evaluation standard of the group: each group should be able to evaluate the objects created by its descendant groups; each group should be able to control the cooperative works among its child groups. For this reason, we give another assumption of cooperative projects.

Assumption 2:

(1) If a designer has right to write or derive some objects, his results should be evaluated by the parent group of the leaf group to which the designer belongs.

(2) If designer A has right to write or derive some objects created by another designer B, designer A must belong to the parent group of the leaf group of designer B.

(3) Designers of a nonleaf group may have right to read or execute the objects which are created by the group and satisfy the evaluation standard of the group.

Due to the property of the evaluation standard hierarchy, (1) of Assumption 2 results in that the objects created by a designer can be evaluated by all of the ancestor groups of the leaf group to which the designer belongs. Furthermore, (2) and (3) of Assumption 2 require that if two or more than two designers can access an identical object, it is necessary to define the evaluation standard of the cooperative work among these designers.

In the case of Figure 6, for instance, designers A and C may have right to write or derive all of objects of components COMPUTER, CPU, RAM, ALU, CU, which have satisfied the evaluation standard of group \{A, C\}. Designer B, however, will not have right to write or derive all of them. Furthermore, designer A may also have right to read or execute objects of components IOC, IOD1, IOD2, MEMO, which have satisfied the evaluation standard of group \{A, F\}. Designer C, however, will not have right to read or execute such objects.

Since each designer group in general only accesses a subset of all the objects, it is required to provide suitable working environments for the group. View functions introduced by us [XU90] can be used to define required working environments of designers.

Definition 3.2 Let \(G_i\) be an nonleaf designer group and \(S_i\) be the evaluation standard of \(G_i\). The view of group \(G_i\) defined on the conceptual schema consists of the following kinds of objects and relationships:

(1) objects which are created by the group \(G_i\) and satisfy \(S_i\),

(2) history relationships between the objects of (1) and the following ones:

(a) history relationships between objects mapped from the transitive history relationships on the conceptual schema, where intermediate objects are not included in the view.

(b) history relationships between configurations mapped from the history relationships in the view between objects of the two different configurations, which satisfy \(S_i\) and belong to an identical component hierarchy.

(3) aggregate relationships \(((C_i, v_k), (C_j, v_l))\) between objects mapped from the aggregate relationship \((C_i, C_j)\) between components on the conceptual schema, where \((C_i, v_k)\) and \((C_j, v_l)\) belong to a same configuration that satisfies \(S_i\), or \((C_i, v_k)\) is the leaf object of a configuration that satisfies \(S_i\) and \((C_j, v_l)\) is the root object of another configuration that satisfies \(S_i\).

That is, objects which can be accessed by one specific group are selected into the view of the group; at the same time the direct or indirect relationships between these objects are provided. Views for a cooperative project consists of one and only one view of each nonleaf designer group \(G_i\) according to its evaluation standard \(S_i\), and an additional view for the project according to the final evaluation stan-
The hierarchy of design groups

Figure 7: Another kind of hierarchies of Example 3.1

Example 3.2 Let \((C_1, v_1), (C_2, v_1)\) and \((C_3, v_1)\) be three objects which construct a configuration \(F_1\). Suppose designer group \(G_1\) plans to refine \(F_1\) into a new configuration, group \(G_1\) is composed of designer \(A, B, C\), and each designer is in charge of the design of a component. In such a case, designers form a hierarchy with root \(\{A, B, C\}\) and its children: \(\{A\}, \{B\}, \{C\}\). The task of each designer is to create an object which can pass its simulation separately. Group \(G_1\) will make use of such objects to create a configuration of the component hierarchy \(\{G, C_2, G\}\), which can pass the same kind of simulation together.

Let the conceptual schema in Figure 8 show objects which are created in the succeeding design processes. Suppose objects \((C_1, v_1), (C_2, v_2), (C_3, v_1), (C_2, v_3), (C_2, v_4), (C_3, v_1), (C_3, v_2), (C_3, v_4)\) satisfy the evaluation standard of group \(G_1\). Suppose \(F_1\) and the configuration constructed by \((C_1, v_3), (C_2, v_4), (C_3, v_4)\) satisfy the evaluation standard of the project. The objects and relationships of the view hierarchy are shown in Figure 8.

As shown in Figure 8, objects which can be accessed by the group \(G_1\) or the project are selected into their views. Furthermore, although a designer group in general only accesses a part of objects, in order to grasp history relationships among them completely, transitive history relationships are also provided. For instance, in the view of group \(G_1\), the history relationship \(((C_1, v_1), (C_1, v_3))\) is the transitively generated one. In the view of the project, not only \(((C_1, v_1), (C_1, v_3)), ((C_2, v_1), (C_2, v_4)), ((C_3, v_1), (C_3, v_4))\) are transitive history relationships, but also a history relationship between two configurations is provided. Aggregate relationships between components, on the other hand, are mapped into the ones between objects.

Figure 8: The view hierarchy of Example 3.2

Because designers of a group may access objects in the view of the group simultaneously, it is necessary to ensure that works of these designers do not interact in undesirable ways. On a view, a set of operations carried out on the conceptual schema is regarded as a primitive operation. For example, in Figure 8 on the view of group \(G_1\), the operations to modify \((C_1, v_1)\) into \((C_1, v_2)\), to modify \((C_1, v_2)\) into \((C_1, v_3)\), and \((C_1, v_3)\) into \((C_1, v_4)\) are regarded as a primitive operation to derive \((C_1, v_3)\) from...
On the view of the project, the operations to derive a new configuration from \( F_i \) are regarded as a primitive operation. For this reason, it is necessary to treat with such a set of operations as a transaction.

**Definition 3.3** A transaction \( T \) of cooperative design database systems is characterized by

\[
< G, \text{view}^p, \{(K_j, U_j)\}, O >
\]

where

1. \( G \) is a group of designers,
2. \( \text{view}^p \) is the view of \( G \)’s parent group, from which \( T \) is regarded as a primitive operation,
3. \( \{(K_j, U_j)\} \) \((j = 1, 2, \ldots)\) represents a primitive operation regarded in \( \text{view}^p \), where operator \( K_j \) is one of operations \( W, D, R, X \), and operand \( U_j \) is a configuration which satisfies the evaluation standard of \( \text{view}^p \). If group \( G \) wants to carry out a merger operation, \( j > 1 \), at most one \( K_j = W \) and the other \( K_j = D \). Otherwise \( j = 1 \),
4. \( O \) is a set of practice operations carried out by the group \( G \), which can be regarded as a primitive operation in \( \text{view}^p \): to merge several \( U_j \) \((j > 1)\) into a single one, or to perform \( K_1 \) \((W, D, R, X)\) on \( U_1 \).

That is, a transaction in design database systems consists of a set of operations carried out by designers of a group, which can be regarded as a primitive operation to fulfill a certain evaluation standard. A designer group may run several transactions at the same time to work on different objects simultaneously. However, the parameters \( G \) and \( \text{view}^p \) of a transaction are not independent, since each designer group need to fulfill the evaluation standard of its parent group. Furthermore, the parameters \( \{(K_j, U_j)\} \) and \( \text{view}^p \) of a transaction are also not independent, since a transaction should be regarded as a primitive operation in the evaluation standard of its parent group.

In the above, we consider each transaction separately. Since objects in the view of group \( G \) also belong to the view of \( G \)’s child groups, a primitive operation for the view of \( G \) in general consists of a set of primitive operations for the views of \( G \)’s child groups. For example, as shown in Figure 8, the group transaction to derive a new configuration from \( F_i \) is carried out by a set of private transactions of designers \( A, B \) or \( C \). That is, transactions in general form a hierarchy.

**Definition 3.4** A transaction hierarchy of a cooperative project consists of a set of transactions, each transaction \( T_i \) is characterized by

\[
< G_i, \text{view}^p_i, \{(K_{ij}, U_{ij})\}, (P_i, O_i) >
\]

where

1. \( G_i \) is a leaf group, then \( O_i \) is a set of practice operations: \( O_i = \{o_{ik} \mid o_{ik} \text{ is carried out by the designer of } G_i, \text{ and the objects which } o_{ik} \text{ accesses belong to } U_i \cup R_i \}; \)
2. \( t \) is a nonleaf group and \( T_k = < G_k, \text{view}^p_k, \{(K_{kt}, U_{kt})\}, (P_k, O_k) > \) is a child transaction of \( T_i \), then \( G_k \) is a child group of \( G_i \), \( \text{view}^p_k \) is the view of \( G_i \), and \( U_{kt} \subseteq U_i \cup R_i \) \((t = 1, 2, \ldots)\); Furthermore, suppose \( W > D > R > X \), if \( U_{kt} \subseteq U_{ij} \), then \( K_{ij} \geq K_{kt} \);
3. \( o_{ik} \) belongs to \( T_i \), then \( o_{ik} \) reads or executes objects belonged to \( U_i \cup U_{ij} \cup R_i \), and such objects do not exist in the view of the child group of \( G_i \) to which the designer carried out \( o_{ik} \) belongs.

That is, according to a hierarchy of views, transactions also form a hierarchy: There is a priority relation \( W > D \), since operation \( W((C_i, v_j)) \) can be realized by \( D((C_i, v_j), (C_i, v_{j+1})) \), deleting \((C_i, v_j)\) and renaming \((C_i, v_{j+1})\) to \((C_i, v_j)\). The other priority relations can be defined in the same way. The condition that if \( U_{kt} \) is included in some \( U_{ij} \) then \( K_{ij} \geq K_{kt} \) requires that a transaction car-

For example, in the case of Example 3.1, private transactions of designer \( A \), \( B \) or \( C \) consist of practice operations to create, modify, refine, or read objects of components \( \text{COMPUTER, CPU, RAM, ALU, CU} \). Transactions of group \( \{A, C\} \) consist of private transactions of designer \( A \), \( B \), or \( C \) or the practice operations carried out by designers \( A \) or \( C \) which read or execute objects of components \( \text{IOC, IOD1, IOD2 or MEMO} \).
4 Transaction Management

In this section, we first summarize the requirements of locking mechanism in design applications, and then discuss problems of transaction management: How to assist designers to define transactions properly and how to decrease designers' burden on transaction management at the same time.

4.1 Locking Mechanism

To control transactions, in conventional database systems strict two-phase locking protocol is usually used, which requires that every transaction unlocks data items at its commit point [BeHG85]. The protocol has the property that any correction of transactions obeying the protocol cannot have a legal, nonserializable schedule. By using the protocol, furthermore, cascading rollback can be prevented. In order to apply strict two-phase protocol to design database systems, however, it is also necessary to lock the results of $D$ operations by $W$ lock, otherwise a correction of transactions obeying the protocol may not have a legal, serializable schedule. A simple example shows such a requirement.

Example 4.1 Suppose two transactions execute in the following way:

Transaction $T_1$

$t_1$: $D((C_i, v_1), (C_i, v_2))$

$t_2$: $D((C_i, v_3), (C_i, v_4))$

...$

If the result of $T_1$ $(C_i, v_3)$ or the result of $T_2$ $(C_i, v_3)$ is not locked by $W$ lock, $T_1$ and $T_2$ can be executed in the above way, i.e., $T_2$ is interleaved among $T_1$.

Another problem in cooperative design database systems is: If a designer group $G_i$ plans to utilize some objects to fulfill its task, it is necessary to lock these objects at that time. Otherwise, those objects may be modified by other groups, by which the plan of group $G_i$ will be interfered with. That is, in design database systems it is necessary to make designers possible to reserve objects, which are planned to be used. In order to reserve objects used by a transaction $T_i = < G_i, \{(K_{ij}, U_{ij})\}, \text{view}_i, (P_i, O_i), R_i >$, $K_{ij}$ should be refined into the one which assigns the maximum type of the operations performed on each objects of configuration $U_{ij}$ clearly. A transaction only can begin to work after all objects required by the transaction are locked.

In order to avoid unnecessary delay of designers, browsing (reading of locked objects) or copying (copying of locked objects) is allowed at any time. Thus, designers have a high degree of flexibility. Of course, the prize is the system can no longer guarantee consistency if designers make use of this flexibility.

In order to simplify the management of the compatibility of locks of transaction hierarchies, we use the following lock mechanism: A child transaction $T_i$ of $T_j$ can only lock the two kinds of objects: (1) the objects which have locked by $T_j$ with a more strong kind of lock or a same kind of lock, suppose $W > D > R > X$; or (2) the objects which are not available from $\text{view}_i$. Due to the definition of the cooperative projects, design database systems are only necessary to check the compatibility of locks of the root transaction of a transaction hierarchy with the ones of other transaction hierarchies, and the compatibility of locks of sibling transactions in a same transaction hierarchy.

4.2 Transaction Management

According to this model, a transaction typically runs in the following way (the syntax used in the following is exemplary and shall help the reader to realize the semantics of the commands):

Definition of a cooperative project

In general, before beginning a large design project, it is necessary to determine the task of each designer group. That is, the designer group hierarchy and evaluation standard of each group should be determined. Utilizing such information, its view hierarchy can be defined and design database systems can check its propriety. In general, only the leader of a project will have right to define the view hierarchy of the project. It may be difficult for designers to define the view hierarchy completely at the beginning of a project. The view of a group, however, are not necessary to be defined until the objects satisfy the evaluation standard of the group are created. That is, the view hierarchy can be defined incrementally according to the progress of the design processes.

Creation of a transaction hierarchy

Any designer can start a new transaction in his name or in the name of the group of which he is a member. A new transaction is always a child transaction of an already existing transaction. That is, the transaction tree is built up from the root to the leaves.
The following command allows to start a new transaction:

```
BEGIN TRANS <trans-name> :=< Gi, viewf, ((Kij, Uij)), P, >
```

Since the initial value of R is φ, designers do not need to assign it explicitly. If the H (parent transaction) is omitted, the transaction represents a root transaction. After a transaction is defined, design database systems can check the propriety of the transaction: whether the transaction of a designer group tries to fulfill the evaluation standard of its parent group (that is, to check the correspondence between the parameters Gi and viewf); whether the transaction can be regarded as a primitive operation on the view of the evaluation standard (that is, to check the correspondence between the parameters viewf and {(Kij, Uij)}). If the transaction is defined as a nonroot transaction, furthermore, design database systems will check the propriety of the transaction hierarchy: whether the transaction performs a part of operations of its parent transaction (that is, to check whether Definitions 3.4 is satisfied). The system only accepts the commands which satisfy Definitions 3.3 and 3.4.

The command not only starts a new transaction, but also locks objects of configuration Uij according to Kij (j = 1, 2, ...). Since a transaction tree is built up from the root to the leaves, it is only necessary to manage the compatibility of locks of the following transactions:

1. the compatibility of locks of the root transaction of a transaction hierarchy with the ones of other transaction hierarchies;
2. the compatibility of locks of sibling transactions in an identical transaction hierarchy.

### Beginning a practice operation

Each practice operation belongs to a transaction. However, it may be trouble for designers to assign the transaction to which practice operations belong. In our transaction model, design database systems can decrease such burden on designers by the following mechanism:

#### Mechanism 1

**input:** A set of transactions LT; a practice operation o;

**output:** A set of transactions CT to which o may belong.

**method:**

```
begin
CT := φ;
for each Tj :=< Gj, viewf, ((Kij, Uij)), (Pi, Oj), Rj > ∈ LT do
```

```
if (Gi is a leaf group of the designer who performs o) ∧ (the objects which o operates ∈ Uk U Rj)
then CT := CT U {Tj};
```

```
if (the designer who performs o is a member of Gi) ∧ (the objects which o operates ∈ Uk U Rj)
∧ (o is a read or execute operation)
∧ (the objects does not exist in the view of Gi’s child group to which the designer belongs)
then CT := CT U {Tj};
```

**end**

That is, the mechanism lists the transactions to which the practice operation may belong. If CT = φ, design database systems will prompt the designer to define the transaction, to which the practice operation belongs. If |CT| > 1, the transaction of the practice operation may not be determined uniquely by design database systems, since a designer may derive several configurations from an existing configuration at same time. In such a case, it is required designers to assign it. Otherwise the transaction of a practice operation can be automatically determined by the system.

### Terminating a transaction or a practice operation

Any transaction can be finished by the END-TRANS <trans-name> command, if all descendant transactions have already finished their work. Designers, however, may be difficult to commit a transaction as soon as the transaction finished its work. Using the information defined at the beginning of the transaction, design database systems can prompt designers to commit a transaction. As defined in Section 3, a transaction Ti :=< Gi, viewf, ((Kij, Uij)), (Pi, Oj), Rj > is regarded as a primitive operation in viewf. For this reason, whether Rj (the result of Ti) is available from viewf is the key point to check whether Ti should be committed or not.

#### Mechanism 2

**input:** A transaction Tk (or a practice operation ok) which has been committed;

**output:** The transaction which can be committed.

**method:**

```
begin
LT := {Tk};
```

```
if (Gi is a leaf group of the designer who performs ok) ∧ (the objects which ok operates ∈ Uk U Rj)
then CT := CT U {Tk};
```

```
if (the designer who performs ok is a member of Gi) ∧ (the objects which ok operates ∈ Uk U Rj)
∧ (ok is a read or execute operation)
∧ (the objects does not exist in the view of Gi’s child group to which the designer belongs)
then CT := CT U {Tk};
```

**end**

That is, the mechanism lists the transactions to which the practice operation may belong. If CT = φ, design database systems will prompt the designer to define the transaction, to which the practice operation belongs. If |CT| > 1, the transaction of the practice operation may not be determined uniquely by design database systems, since a designer may derive several configurations from an existing configuration at same time. In such a case, it is required designers to assign it. Otherwise the transaction of a practice operation can be automatically determined by the system.
then prompt designers of $G_i$ that $T_i$ can be committed;
end

Of course, designers may find that a transaction cannot fulfill its planned task. In such a case, the transaction is necessary to be interrupted by designers themselves. For the transaction $T_i$, where $j = 1$ and $K_i = 'R'$ or 'X', furthermore, it is also required designers to commit it explicitly.

5 Conclusion

This paper made the properties of transactions in cooperative design database systems clear. Using these properties, designers can define or commit transactions properly. That is, works of groups of designers will not interact in undesirable ways, and will not lose a concurrency at the same time. We also have shown that if designers define the properties of transactions at the beginning, design database systems can check the property of the transactions, can prompt designers to define or to commit transactions and can determine the transactions of practice operations automatically in most cases. For this reason, the burden imposed on designers of transaction management can be decreased.

This paper is meant to be a framework and a starting point for an overall solution, and not a detailed solution to all problems of interest in this context. Some important problems for further research are the modification of the definition of transactions or view hierarchies dynamically. Now, if designers want to refine their view hierarchy, it is necessary to interrupt the transactions based on such views first. If designers want to refine the definition of a transaction, it is required to be carried out by two steps: interrupting the transaction and redefining a new transaction.

Acknowledgments

The author is very much grateful to Prof. Yahiko Kambayashi of Kyoto University for giving her proper guidance as well as valuable comments on this paper.

References


