

Specification and Coordination of Long-Running Design Activities for CAD Environments¹

Haengrae Cho and Songchun Moon

Department of Information and Communications Engineering
Korea Advanced Institute of Science and Technology
207-43, Cheongryang, Dongdaemun, Seoul 130-012, Republic of Korea

Fax: +82-2-960-6743

E-mail: hrcho@sicom.kaist.ac.kr

ABSTRACT

This paper presents a new transaction management scheme, LOT, for coordinating design activities in CAD Environments. Unlike the previous approaches for coordinating design activities, in which each designer faces complexity of a design activity, LOT encapsulates this complexity from designers. This is achieved by two notions: (1) *transaction template* that guides the designer not to produce incorrect design results due to misunderstanding of complex design activities, and (2) *interleaving specification* that enables the designer to cooperate with his/her group members in a consistent way. LOT also contains a new concurrency control scheme, called *semantic concurrency control*, which takes advantage of interleaving specification. In LOT, therefore, any designer does not need to know the details of the design activity, and also does not need to concern about the moment of release for his/her intermediate results.

Keywords: intelligent user interface, transaction template, interleaving specification, semantic concurrency control

1. Journal of Microprocessing and Microprogramming 40(7) 1994.

1. INTRODUCTION

In computer aided design (CAD) environments, a design activity represents an abstract design function required to achieve a specific design goal, which is performed by a designer. VLSI design projects, in particular, are usually too large and too complex to be completed by a single designer. It is a common practice that a VLSI design project is accomplished by a number of design groups, and each designer cooperates with his/her group members to perform his/her design activities [1,2]. In VLSI design projects, therefore, a flexible and efficient mechanism for coordinating the design activities is the key to the success of the projects.

In a CAD database system, design activities are represented as transactions and coordinating the design activities is modeled by a concurrency control scheme for the transactions [2]. While the notion of a transaction provides a good starting point for modeling design activities, it has a serious limitation in coordinating the design activities. This is because a transaction in design applications is usually long-lived and requires a tight cooperation with transactions performed by other group members. When a transaction is long-lived, it is highly undesirable to force transactions to wait until other transactions with conflicting data access terminate. Furthermore, undoing all the execution results of the transaction is intolerable because a designer performs all his/her design activities in the transaction. Cooperation between transactions requires that several transactions in a group can exchange knowledge in order to be able to continue their works. This implies that two or more transactions working on shared objects may not be serializable. Therefore, the conventional transaction management schemes must be enhanced to coordinate the design activities.

A number of transaction models that extend the classical notion of a transaction have been proposed to coordinate the design activities [3-5]. The common approach used in those models is that user interactions are incorporated as a *part* of the transaction management. For example, in the group-oriented CAD transaction model [3], a designer is allowed to release design data at any time, and thus the design data are available to the other designers' activities in the same group. The cooperating CAD transaction model [4], under the same approach, enables a designer to define and create a short-duration transaction (SDT) which performs a part of a designer's task; this implies that a designer's task is modeled as a set of SDTs, and the design data created by an SDT is allowed to be accessed by other designers' SDTs in the same group. The notification-based transaction model [5], under the same

approach, alerts a designer of any conflict, for instance, an attempt to lock an object that has already been locked in an exclusive mode; this kind of conflict is resolved by negotiation between the two designers under the assumption that they are able to interact to resolve the conflict.

The main problem of incorporating user interaction in the transaction management in this manner is that human beings are *error-prone*, which could result in inconsistent data. Furthermore, the structure of a design activity is usually too complex for a designer to understand the details of the design activity. This is illustrated by Example 1.

Example 1 (Problems of Previous Transaction Models): Suppose a transaction T of a simple design activity that consists of five design tools. Figure 1 shows the graphical abstraction of T .

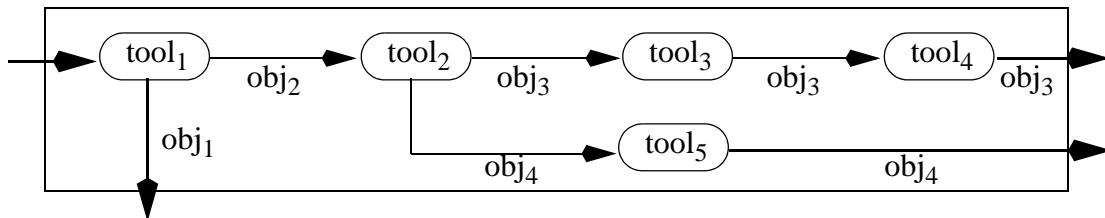


Figure 1 Graphical abstraction of an example transaction

The output objects of T are obj_1 , obj_3 , and obj_4 produced by $tool_1$, $tool_4$, and $tool_5$, respectively. When a transaction in the same group tries to access obj_1 and obj_3 , T allows the access of obj_1 after $tool_1$ is terminated because obj_1 is assumed not to be updated in T . The access of obj_3 , however, is prohibited until $tool_4$ is terminated. This requires that designers must know the details of the design activity, i.e., the input and output objects of each design tool, and output-to-input relationships among design tools. When a number of design tools are involved in a particular design activity and their interrelationships are complex, it is difficult for a designer to understand the details of the design activity and also to decide whether intermediate results can be released or not. ■

While there are some approaches to hide the complexity of a design activity, for example, EDA [6], Cadweld [7], and Ulysses [8], they concentrate only on modeling single design activity using an intelligent user interface. In other words, they do not address the issues associated with multiple design activities like concurrency control. Therefore, a transaction management scheme that incorporates the notion of a transaction into the intelligent user interface is a strong requirement for coordinating the design activities.

In this paper, we propose a new transaction management scheme called LOT, which supports design transparency in coordinating the design activities. Unlike the previous approaches for coordinating design activities, in which designers directly face the complexity of design activities,

LOT hides the complexity and thus enables designers to perform their design activities without acquiring detailed knowledge on them. This is achieved by the notion of *transaction template* that identifies design tools for the design activity and their invocation sequence, and by the notion of *interleaving specification* that represents the set of possible interleaving points where other transactions are allowed to access the intermediate results of the transaction.

LOT also uses *run-time information* of a transaction to support transactions running on different types of applications. Run-time information consists of group information and work structure. Group information represents the organizational structure of a design project. Work structure specifies how the members of a design group cooperate with each other to achieve the design goals. With run-time information, LOT can exploit the semantics of the execution environments of a transaction, and therefore a flexible transaction management according to the applications is possible.

The rest of the paper is organized as follows: The next section introduces the transaction model of LOT. In Section 3, we describe a methodology for specifying design activities with the notion of transaction template. The notion of interleaving specification is presented in Section 4. This is followed by a section that describes the semantic concurrency control which takes advantage of interleaving specification. In Section 6, we discuss the correctness of the semantic concurrency control. Finally, conclusions and further research areas appear in Section 7.

2. TRANSACTION MODEL

In LOT, a transaction is constructed *hierarchically* to represent the hierarchical structure of a design process. The transaction hierarchy implies that a transaction representing a design process consists of several subtransactions, each of which may be a hierarchy. The transaction hierarchy, therefore, can be represented as a transaction tree, in which nodes and edges represent transactions and parent-child relationships in transactions, respectively. Figure 2 shows a typical transaction hierarchy in a CAD environment.

A CAD environment consists of a number of *process transactions*, each of which represents a distinct design process. A process transaction is decomposed into a set of *group transactions*, each of which indicates a part of the design process and corresponds to the coherent unit of design activities.

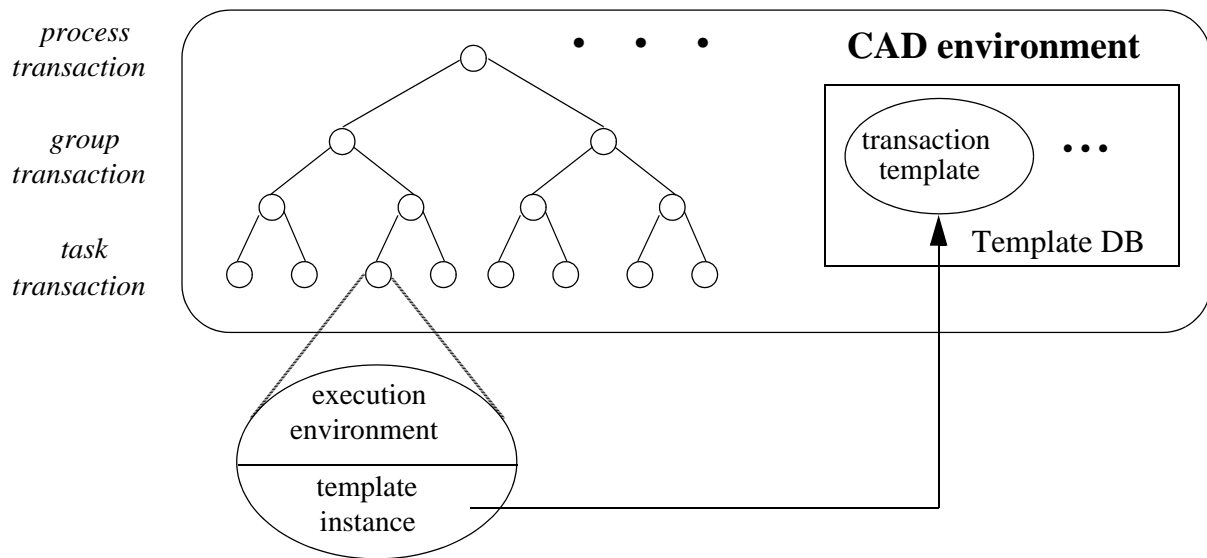


Figure 2 A transaction hierarchy in a CAD environment

A group transaction has several cooperative member transactions. A member transaction represents a subgroup or an atomic design task, each of which is a cooperative component of a design group. When a member transaction represents a design task, we regard this transaction as a *task transaction*. A task transaction corresponds to a leaf node in the transaction hierarchy.

In the transaction hierarchy, non-leaf transactions decide design methodologies that define design styles, design strategies, implementation techniques, and so on. According to the design methodology, the organization of subgroups and the role of each subgroup can be matured. The actual design activities are performed by leaf transactions.

Design tasks in a CAD environment have usually many common sequences of design tool invocations. In this sense, a design task can be predefined as a transaction template by an *expert* designer, and then a novice designer can continue his work just by instantiating the transaction template. When a transaction template is instantiated, a run-time image of the transaction template and its execution environments are created. In LOT, the template database stores every transaction template.

Our notion of a transaction hierarchy is different from that of conventional nested transaction model [9], in the sense that a transaction of the transaction hierarchy is a *cooperative* unit rather than an isolated one. This implies that several transactions may exchange their intermediate results in order to be able to continue their work.

3. TRANSACTION TEMPLATE

A transaction template defines the behavioral composition of one or more cooperating design tools in the design task. Each transaction template specifies two aspects of the behavioral composition: *identifications* of the design tools and an *execution plan* for the design task. The followings describe each of the aspects in more detail.

- The transaction template identifies the design tools and their interrelationships. Each design tool is characterized with a number of generic features, such as the input it requires, the output it produces, and what constraints should be satisfied. Interrelationships consist of type relationships and functional relationships. *Type relationships* specify that the corresponding design tools must support compatible input and output object formats. *Functional relationships* specify that one design tool must produce correct output objects required in the other design tool, and make certain conditions true after its execution. With type and functional relationship, the transaction template captures the behavioral dependencies between design tools.
- The transaction template defines an execution plan for the design task. The plan consists of sequencing information of design tools and the information flow between design tools. Since the transaction template may specify a complex procedural sequence of design tools, LOT supports a number of language constructs. They include sequential statement (separated by ;), conditional statement (*if-then-else*), iterative statement (*while-do*), and parallel statement (*parbegin-parend*). A parallel statement causes all design tools within *parbegin* and *parend* to execute in parallel.

In order to demonstrate how a transaction template is constructed, we select a design task for the standard cell design in VLSI design environments. This is illustrated in Example 2.

Example 2 (Transaction Template for Standard Cell Design): Suppose a typical design flow for standard cell design in Figure 3(a). Figure 3(b) shows a transaction template, **CellDesign**, for the design flow. **CellDesign** has three output objects and consists of seven design tools.

By assigning the same name to the objects that are specified in related design tools, **CellDesign** represents the functional relationships of design tools. For example, the functional relationship of **logic_entry** and **logic_simulator** is represented by **logic_description_list**. To support the type relationships between design tools, the objects of the same name but specified in different design tools must have compatible object formats.

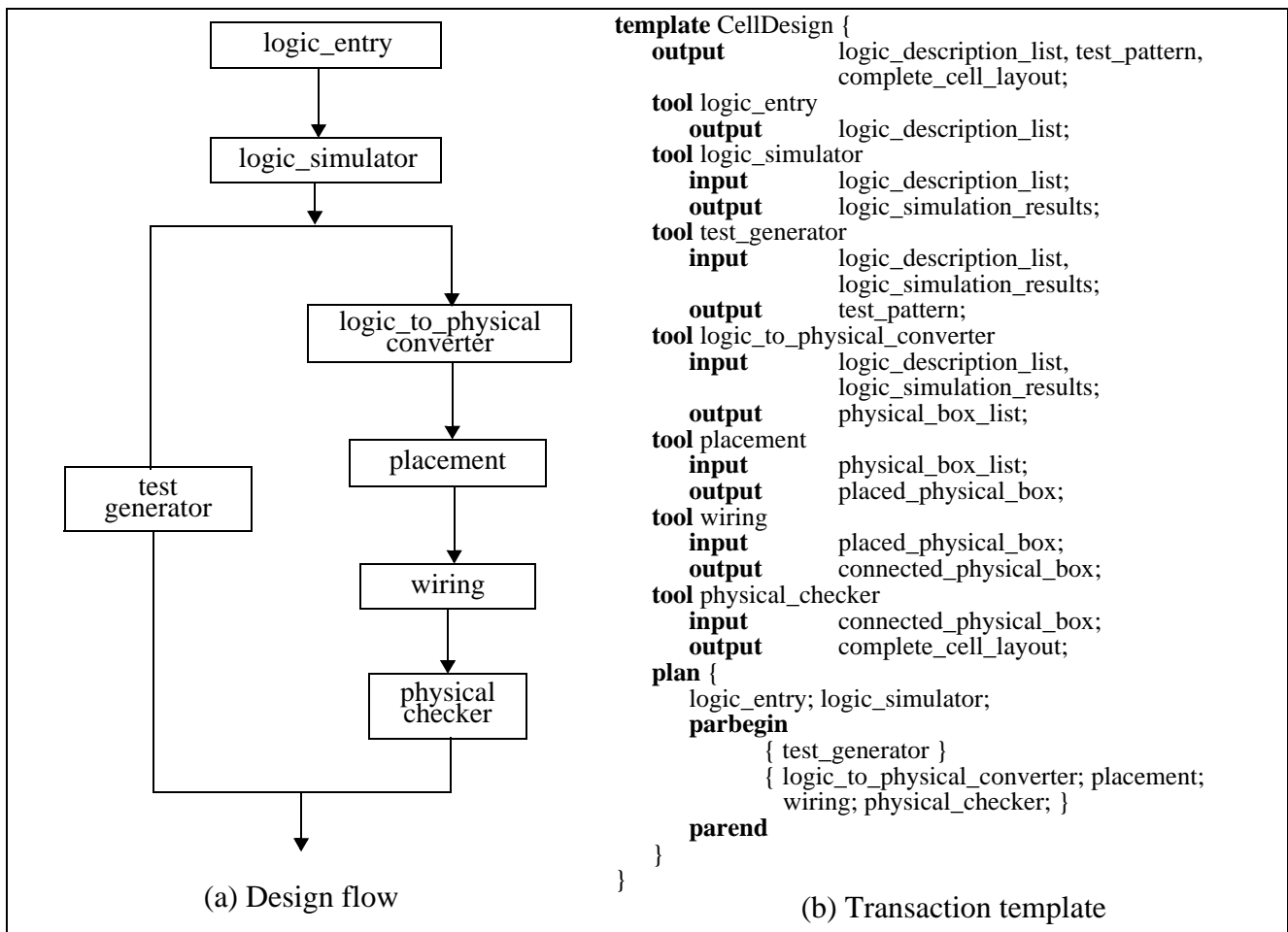


Figure 3 A transaction template for standard cell design

For example, the output object of `logic_entry` and the input object of `logic_simulator` must have compatible object formats.

`CellDesign` also includes the planning of design tool invocations. The plan of `CellDesign` shows that `logic_entry` and `logic_simulator` are executed first in that sequence. Following the design tools, two design activities are executed concurrently. The first activity corresponds to the generation of test data, executed by `test_generator`. The second activity corresponds to the production of a physical cell, executed by four design tools, `logic_to_physical_converter`, `placement`, `wiring`, and `physical_checker`, in that sequence. ■

After a transaction template is selected to perform a design task, an instance of the transaction template is created. The instance is the run-time image of the transaction template. Once the instance is created, the execution routine loops through a plan of the transaction template. Because the transaction template includes only the specification of the design task, run-time bindings should be performed between specification and actual arguments. From actual arguments, conditional statements can be resolved and actual names of input objects and output objects can be obtained. Example 3 illustrates the procedure of template instantiation.

Example 3 (Template Instantiation): Suppose that we intend to instantiate `CellDesign`. Because there are no conditional statements and input objects in `CellDesign`, only the actual output objects are determined. We can instantiate `CellDesign` as follows:

`CellDesign (logic, test, layout)`

`Logic` is the actual object identifier of `logic_description_list`. Similarly, `test` corresponds to `test_pattern`, and `layout` corresponds to `complete_cell_layout`.

An input object of a design tool can be either an input object of the template or an output object of the preceding design tool. Since `CellDesign` has no input objects, every input object of the design tools is generated from the preceding design tools. If an output object of a design tool matches that of the template, the object should be bound to the actual argument of the template. For example, `test_pattern`, which is the output of `test_generator`, is bound to `test`. ■

4. INTERLEAVING SPECIFICATION

In this section, we describe a new correctness criterion for concurrent design activities. It weakens the usual notion of serializability by permitting controlled interleaving among transactions. The basic idea is to release objects at any phase of transaction execution based on the transaction template. If an object is released then other transactions can access the object. The interleaving specification of LOT, therefore, consists of the following two aspects: *what* objects can be released and *when* the objects can be released. Now, we define the notion of interleaving point that represents the former aspect of an interleaving specification.

Definition 1 (Interleaving Point): An *interleaving point* of a transaction consists of three attributes $\langle T, OBJ, MODE \rangle$ where T is a transaction identifier and OBJ is an object identifier. $MODE$ can be R for read operation and W for write operation. If T creates an interleaving point $\langle T, OBJ, R \rangle$ then other transactions can read OBJ . Those transactions, however, cannot update OBJ until an interleaving point $\langle T, OBJ, W \rangle$ is created. □

The creation time of an interleaving point can be determined *automatically* according to the access pattern of the transaction template. In order to specify the access pattern, we define the notion of object dependency.

Definition 2 (Object Dependency): In a transaction template, an object OBJ_n is *dependent* on another object OBJ_l if one of the following conditions is satisfied;

- (1) when OBJ_1 is equal to OBJ_n (reflexive property), or
- (2) when there is a design tool of which input objects include OBJ_1 and output objects include OBJ_n , or
- (3) when there is a sequence of design objects, $OBJ_1 OBJ_2 \dots OBJ_{n-1} OBJ_n$ such that OBJ_{i+1} is dependent on OBJ_i , $1 \leq i < n$.

If OBJ_n is dependent on OBJ_1 and there are no other objects that are dependent on OBJ_n except itself, then OBJ_n is *finally dependent object* of OBJ_1 . \square

Now, we introduce the notion of interleaving specification that represents what objects can be released and when the objects can be released.

Definition 3 (Interleaving Specification): An *interleaving specification* of a transaction, T , consists of two attributes $\langle IP, COND \rangle$ where IP is an interleaving point defined on T and $COND$ is a condition representing the creation time of IP . According to the type of T (task transaction or group transaction), $COND$ is represented differently. When T is a task transaction, $COND$ specifies the followings for each type of IP :

- (1) $\langle T, OBJ, R \rangle$: a design tool that updates OBJ at last. When such design tool does not exist (that is, OBJ is not updated in T), $COND$ is null.
- (2) $\langle T, OBJ, W \rangle$: design tools that produce finally dependent objects of OBJ .

On the other hand, if T is a group transaction, $COND$ specifies the followings:

- (1) $\langle T, OBJ, R \rangle$: a set of interleaving points $\langle T_i, OBJ, R \rangle$ where T_i is a member transaction of T that updates OBJ . When such T_i does not exist (that is, OBJ is not updated in member transactions), $COND$ is null.
- (2) $\langle T, OBJ, W \rangle$: a set of interleaving points $\langle T_i, OBJ, W \rangle$ where T_i is a member transaction of T that reads or updates OBJ .

For a task transaction, an IP is created when every design tool specified in $COND$ is terminated. For a group transaction, an IP is created when every interleaving point specified in $COND$ is created. \square

The implications of an interleaving specification are as follows. First, a condition for the interleaving specification of a group transaction is represented by *interleaving points* rather than design tools. This enables information hiding between a group transaction and its member transactions. Only a task transaction should detect the termination of target design tools. A group transaction just needs to consider whether the interleaving points of its member transactions are created. Second, when T is

a task transaction, a condition for $\langle T, OBJ, R \rangle$ specifies *one* design tool. This is because we assume that, in a given task transaction, all updates of an object are serial. When concurrent updates of the object are required, we recommend that each updates should be specified as a separate transaction template. On the other hand, when T is a group transaction, a condition for $\langle T, OBJ, R \rangle$ specifies *multiple* interleaving points. This is because member transactions of T can update OBJ concurrently, and thus T cannot predict the last update member transaction. Finally, when T is a task transaction, a condition for $\langle T, OBJ, W \rangle$ specifies *multiple* design tools. This is because an object can be read by a number of concurrent design tools, each of which is independent. By prohibiting concurrent updates but allowing concurrent reads, we can support intra-transaction parallelism without any concurrency control overhead. Example 4 illustrates interleaving specifications of a group transaction and task transactions.

Example 4 (Interleaving Specification): Suppose a simple group transaction, T_{cell} , represented at Figure 4(a). T_{cell} has two member transactions, CellDesign and CellMask. The transaction template of CellMask is represented in Figure 4(b). Suppose again that CellDesign and CellMask are instantiated as follows:

CellDesign(logic, test, layout), CellMask(layout, test, cell).

Table 1 shows the interleaving specifications of CellDesign, CellMask, and T_{cell} . In Table 1(a), $\langle \text{CellDesign}, \text{logic}, R \rangle$ is created just after terminating logic_entry, because logic is not updated further. The final dependent objects of logic are test and layout. They are produced by test_generator and physical_checker, respectively. This implies that $\langle \text{CellDesign}, \text{logic}, W \rangle$ should not be created until both test_generator and physical_checker are terminated. In Table 1(b), the creation time of $\langle \text{CellMask}, \text{layout}, R \rangle$ is null, because CellMask does not update layout. This implies that when CellMask is running, other transactions can read layout. In Table 1(c), $\langle T_{cell}, \text{layout}, R \rangle$ is created just after creating $\langle \text{CellDesign}, \text{layout}, R \rangle$ because $\langle \text{CellMask}, \text{layout}, R \rangle$ is null. $\langle T_{cell}, \text{layout}, W \rangle$ cannot be created until CellDesign and CellMask create interleaving points for layout with W mode. This is because, for every member transaction, interleaving points for layout with W mode are not null. The same is true for test. ■

Even though two transactions are in a group, there may be several types of work structure between them. For example, if there are few interactions between transactions, each transaction can be regarded as an isolated unit like a conventional transaction. However, if two transactions interact very frequently, then more sophisticated transaction management schemes are required. Furthermore, some transactions try to access objects without any delay, and they can live with inconsistencies in the objects.

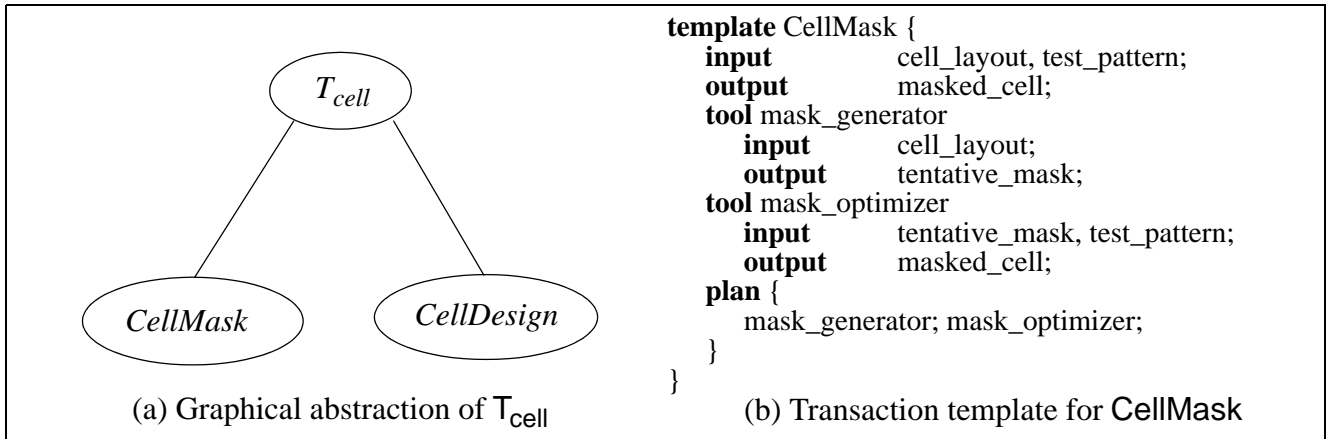


Figure 4 An example of a group transaction and its member transaction

Object	Interleaving Point	Creation Time
logic	<CellDesign, logic, R>	logic_entry
	<CellDesign, logic, W>	test_generator physical_checker
test	<CellDesign, test, R/W>	test_generator
layout	<CellDesign, layout, R/W>	physical_checker

(a) Interleaving specification of CellDesign

Object	Interleaving Point	Creation Time
layout	<CellMask, layout, R>	null
	<CellMask, layout, W>	mask_optimizer
test	<CellMask, test, R>	null
	<CellMask, test, W>	mask_optimizer
cell	<CellMask, cell, R/W>	mask_optimizer

(b) Interleaving specification of CellMask

Object	Interleaving Point	Creation Time
logic	<T _{cell} , logic, R>	<CellDesign, logic, R>
	<T _{cell} , logic, W>	<CellDesign, logic, W>
layout	<T _{cell} , layout, R>	<CellDesign, layout, R>
	<T _{cell} , layout, W>	<CellDesign, layout, W>, <CellMask, layout, W>
test	<T _{cell} , test, R>	<CellDesign, test, R>
	<T _{cell} , test, W>	<CellDesign, test, W>, <CellMask, test, W>
cell	<T _{cell} , cell, R>	<CellMask, cell, R>
	<T _{cell} , cell, W>	<CellMask, cell, W>

(c) Interleaving specification of T_{cell}

Table 1 Interleaving specifications of example transactions²

We divide the work structure into three classes: isolated, cooperative, and inconsistent. In *isolated* work structure, there are few interactions between transactions. This implies that few objects

2. A notation of <CellDesign, test, R/W> represents <CellDesign, test, R> and <CellDesign, test, W>. The same is true for <CellDesign, Layout, R/W> and <CellMask, cell, R/W>

are shared between transactions. Therefore, a transaction in isolated work structure may be regarded as an isolated unit. In *cooperative* work structure, transactions cooperate each other to achieve the common goals. Transactions in cooperative work structure often share a number of objects. To achieve effective performance, a transaction in cooperative work structure should not be regarded as an isolated unit. In *inconsistent* work structure, each transaction does not concern the consistency of objects, but just want to see the current states of the objects. Transactions in this work structure can be executed without any overhead on the transaction management.

In isolated work structure, every interleaving point of a transaction is created when the transaction commits. Furthermore, the transaction should access only the committed objects. This notion is the same as that of conventional transactions. In cooperative work structure, every interleaving point of the transaction is created according to the interleaving specification. Transactions are allowed to access uncommitted objects if their interleaving points have been created. In inconsistent work structure, the interleaving points are insignificant. When a transaction with inconsistent work structure tries to access an object, the transaction can be executed immediately regardless of the interleaving points. The effects of work structure on interleaving specifications are illustrated in Example 5.

Example 5 (Effects of Work Structure): Suppose T_{Cell} transaction described in Example 4. If **CellMask** executes with isolated work structure, it should not read **layout** until **CellDesign** commits. However, if **CellMask** executes with cooperative work structure, it can read **layout** when $\langle \text{CellDesign}, \text{layout}, R \rangle$ is created. Since $\langle \text{CellDesign}, \text{layout}, R \rangle$ is created while **CellDesign** executes, **CellMask** does not need to wait until **CellDesign** commits. This results in allowing more concurrent executions of transactions with cooperative work structure than those with isolated work structure. Now, suppose a new transaction with inconsistent work structure, **StateLog**, which has a role to browse the current state of objects. **StateLog** must be able to access objects without unnecessary delay and it can live with inconsistencies in the objects. In this case, **StateLog** can execute immediately regardless of the interleaving points. ■

5. SEMANTIC CONCURRENCY CONTROL

In this section, we propose a new concurrency control scheme, called semantic concurrency control scheme (SCC) that takes advantage of interleaving specifications. The basic idea of SCC is that a transaction can access an object when an interleaving point of the object is created. The transaction need not wait until the owner transaction of the object terminates. This enables transactions to be executed more concurrently than conventional concurrency control schemes.

Since SCC runs on transaction hierarchies, we introduce some terminologies on a transaction hierarchy. $\mathbf{PATH}(T_1, T_n)$ is a sequence of transactions, $T_1 T_2 \dots T_{n-1} T_n$, where T_i is a member transaction of T_{i+1} , $1 \leq i < n$. If the sequence does not exist, $\mathbf{PATH}(T_1, T_n)$ is defined as null. $\mathbf{ANCESTOR}(T_i)$ is a set of transactions T_j , where $\mathbf{PATH}(T_i, T_j)$ is not null. $\mathbf{LCA}(T_i, T_j)$ means a least common ancestor transaction of T_i and T_j . That is, $\mathbf{LCA}(T_i, T_j)$ is a first intersecting transaction of $\mathbf{PATH}(T_i, T_{root})$ and $\mathbf{PATH}(T_j, T_{root})$, where T_{root} is a root transaction in the transaction hierarchy. When T_i and T_j are in different transaction hierarchies, $\mathbf{LCA}(T_i, T_j)$ is T_{system} that is a virtual transaction. $\mathbf{MLCA}(T_i, T_j)$ is a transaction that is directly followed by $\mathbf{LCA}(T_i, T_j)$ in $\mathbf{PATH}(T_i, T_{root})$. $\mathbf{MLCA}(T_i, T_j)$ implies a member transaction of $\mathbf{LCA}(T_i, T_j)$, which is an ancestor transaction of T_j . If $\mathbf{LCA}(T_i, T_j)$ is T_{system} , then $\mathbf{MLCA}(T_i, T_j)$ is T_{root} that is a root transaction of the transaction hierarchy which includes T_i .

For example, consider a transaction hierarchy illustrated in Figure 5. $\mathbf{PATH}(T_9, T_1)$ is a sequence of transactions $T_9 T_6 T_3 T_1$. Similarly, $\mathbf{PATH}(T_7, T_1)$ is $T_7 T_3 T_1$. $\mathbf{ANCESTOR}(T_9)$ is $\{T_9, T_6, T_3, T_1\}$. $\mathbf{ANCESTOR}(T_7)$ is $\{T_7, T_3, T_1\}$. $\mathbf{LCA}(T_9, T_7)$ is T_3 because the first intersecting transactions of $\mathbf{PATH}(T_9, T_1)$ and $\mathbf{PATH}(T_7, T_1)$ is T_3 . $\mathbf{MLCA}(T_9, T_7)$ is T_6 because T_6 is directly followed by T_3 in $\mathbf{PATH}(T_9, T_1)$. Similarly, $\mathbf{MLCA}(T_7, T_9)$ is T_7 itself.

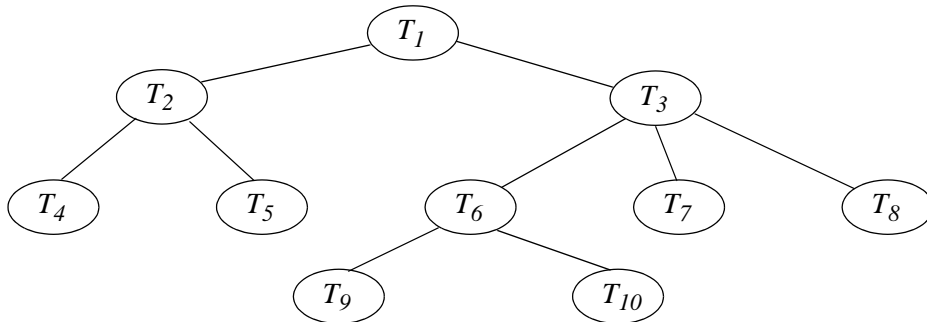


Figure 5 A sample transaction hierarchy

Now, we describe SCC. We will assume that the system maintains information about objects and transactions. In particular, for each object OBJ the system maintains:

- $READ(OBJ)$: the set of transactions that have read OBJ
- $WRITE(OBJ)$: the set of transactions that have written OBJ
- $ACCESS(OBJ)$ ³: $READ(OBJ) \cup WRITE(OBJ)$
- $IP(OBJ)$: the set of interleaving points that have been created on OBJ

For each active transaction T_x the system maintains:

- $WS(T_x)$: the work structure of T_x which can be *isolated*, *cooperative*, or *inconsistent*
- $IPCOND(T_x, OBJ)$: the set of conditions representing when interleaving points for OBJ are created
- $\delta(T_x)$: the set of objects accessed by T_x or T_x 's descendant transactions

When a transaction T_x begins, $\delta(T_x)$ is emptied and $IPCOND(T_x, OBJ)$ is created according to the interleaving specification of T_x . OBJ is appended to $\delta(T_x)$ when T_x accesses OBJ first. When T_x reads OBJ , T_x is appended to $READ(OBJ)$ and $ACCESS(OBJ)$. Similarly, when T_x writes OBJ , T_x is appended to $WRITE(OBJ)$ and $ACCESS(OBJ)$. When $IPCOND(T_x, OBJ)$ is satisfied, an interleaving point $\langle T_x, OBJ, mode \rangle$ is appended to $IP(OBJ)$. The term “*mode*” is a representative of R for read operation and W for write operation. In this case, if T_x is a group transaction, every interleaving point $\langle T_i, OBJ, mode \rangle$ in the $IPCOND(T_x, OBJ)$ is deleted from $IP(OBJ)$.

A transaction T_x can commit if every descendant transaction has committed. When T_x commits and it is not a root transaction, every object accessed by T_x is inherited to T_x 's parent transaction. That is, for each OBJ_i included in $\delta(T_x)$, $READ(OBJ_i)$ and $WRITE(OBJ_i)$ should be changed so that T_x is replaced by T_x 's parent transaction. This implies that every object associated with a non-leaf transaction has been inherited by committed descendant transactions. SCC can support nested atomicity by allowing transactions to access objects associated with their ancestor transactions. If T_x is a root transaction and it commits, T_x is deleted from $READ(OBJ_i)$ and $WRITE(OBJ_i)$ for all OBJ_i in $\delta(T_x)$.

Algorithm 1 shows an access protocol for processing T_x 's read operation on OBJ . If $WS(T_x)$ is

3. Even though $ACCESS(OBJ)$ is a redundant information, we use it for convenience. Of course, an implementation of SCC can omit it.

Algorithm 1 Processing T_x 's read operation on OBJ

```

CASE (WS( $T_x$ )) OF
  inconsistent:    RETURN(ACCEPT); (* Always accepted *)
  isolated:
    FOREACH ( $T_i \in \text{WRITE}(\text{OBJ})$ )
      IF  $T_i \notin \text{ANCESTOR}(T_x)$     RETURN(REJECT);
    ACCESS(OBJ) := ACCESS(OBJ)  $\cup T_x$ ; READ(OBJ) := READ(OBJ)  $\cup T_x$ ;
    RETURN(ACCEPT);
  cooperative:
    FOREACH ( $T_i \in \text{WRITE}(\text{OBJ})$ )
      IF  $\langle \text{MLCA}(T_i, T_x), \text{OBJ}, R \rangle \notin \text{IP}(\text{OBJ}) \wedge T_i \notin \text{ANCESTOR}(T_x)$ 
        RETURN(REJECT);
    ACCESS(OBJ) := ACCESS(OBJ)  $\cup T_x$ ; READ(OBJ) := READ(OBJ)  $\cup T_x$ ;
    RETURN(ACCEPT);
END;

```

inconsistent, T_x can read OBJ regardless of OBJ 's state (e.g. IP, WRITE, etc.). When the $WS(T_x)$ is isolated, T_x should access only committed objects. This is guaranteed by checking whether every write transaction is included in $\text{ANCESTOR}(T_x)$. T_x can read OBJ only if every write transaction is an ancestor transaction of T_x . When $WS(T_x)$ is cooperative, T_x can read OBJ if one of the following condition is satisfied for every write transaction T_i . First, if $\text{IP}(OBJ)$ includes $\langle \text{MLCA}(T_i, T_x), OBJ, R \rangle$, T_x can read OBJ . This is because the descendant transactions of $\text{MLCA}(T_i, T_x)$ do not update OBJ further. Note that $\text{MLCA}(T_i, T_x)$ is a member transaction of $\text{LCA}(T_i, T_x)$. It is the closest transaction of T_x that can inform the status of T_i . Second, if T_i is included in $\text{ANCESTOR}(T_x)$, T_x can also read OBJ . This is a case when a committed member transaction T_{sub} of T_i has inherited OBJ to T_i . Since T_i is a parent transaction of T_{sub} and an ancestor transaction of T_x , $\text{LCA}(T_{sub}, T_x)$ is T_i . This means that $\text{MLCA}(T_{sub}, T_x)$ is T_{sub} . Because T_{sub} committed, every interleaving point of T_{sub} has been created and therefore T_x can read OBJ .

Algorithm 2 shows an access protocol for processing T_x 's write operation on OBJ . Note that $\text{WRITE}(OBJ)$ of Algorithm 1 is replaced by $\text{ACCESS}(OBJ)$ in Algorithm 2. This is because a write operation conflicts to both read and write operations. In addition, a transaction with inconsistent work structure cannot execute a write operation. This is due to permitting write operations of the transaction could result in database inconsistency.

Example 6 (Semantic Concurrency Control): Suppose a group transaction T_{chip} . The structure of T_{chip} is represented at Figure 6(a). T_{chip} has five member transactions. $T_{\text{CellDesign}}$ and T_{CellMask} are group

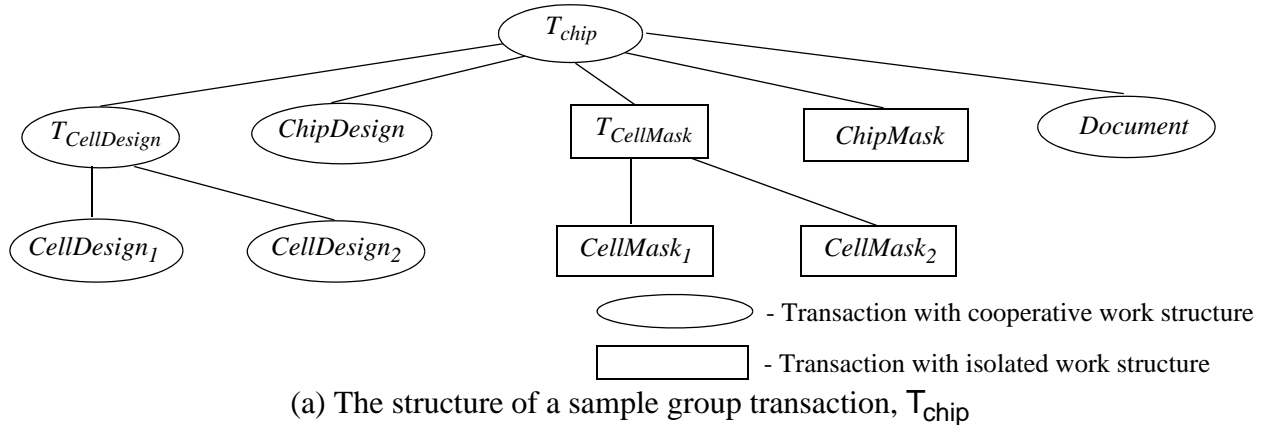
Algorithm 2 Processing T_x 's write operation on OBJ

```
CASE (WS( $T_x$ )) OF
  inconsistent:    RETURN(REJECT); (* Always rejected *)
  isolated:
    FOREACH ( $T_i \in$  ACCESS(OBJ))
      IF  $T_i \notin$  ANCESTOR( $T_x$ )    RETURN(REJECT);
    ACCESS(OBJ) := ACCESS(OBJ)  $\cup$   $T_x$ ; WRITE(OBJ) := WRITE(OBJ)  $\cup$   $T_x$ ;
    RETURN(ACCEPT);
  cooperative:
    FOREACH ( $T_i \in$  ACCESS(OBJ))
      IF  $\langle$ MLCA( $T_i, T_x$ ), OBJ, W $\rangle \notin$  IP(OBJ)  $\wedge$   $T_i \notin$  ANCESTOR( $T_x$ )
        RETURN(REJECT);
    ACCESS(OBJ) := ACCESS(OBJ)  $\cup$   $T_x$ ; WRITE(OBJ) := WRITE(OBJ)  $\cup$   $T_x$ ;
    RETURN(ACCEPT);
END;
```

transactions. ChipDesign is a task transaction that creates a chip_logic and chip_layout. ChipMask has a role to mask a complete chip. The role of Document is to produce a manual of the chip. Figure 6(b) represents the input and output objects of ChipDesign, ChipMask, and Document. Each task transaction is instantiated as follows:

```
CellDesign1(logic1, test1, layout1), CellDesign2(logic2, test2, layout2)
ChipDesign(logic1, layout1, logic2, layout2, chip_logic, chip_layout)
CellMask1(layout1, test1, cell1), CellMask2(layout2, test2, cell2)
ChipMask(chip_layout, cell1, cell2, chip_mask)
Document(chip_logic, chip_layout, chip_mask, manual)
```

Now, we describe three types of interactions between transactions on the basis of their work structures.



<pre> template ChipMask { input chip_layout, cell1, cell2; output chip_mask; ... } </pre>	<pre> template Document { input chip_logic, chip_layout, chip_mask; output chip_manual; ... } </pre>	<pre> template ChipDesign { input logic_oid1, layout_oid1, logic_oid2, layout_oid2; output chip_logic, chip_layout; ... } </pre>
--------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------

(b) Simplified transaction templates for ChipMask, Document and ChipDesign

Figure 6 A sample group transaction and its member transactions

• **Isolated vs. Isolated:**

ChipMask cannot read $cell_1$ produced by $CellMask_1$ until $T_{CellMask}$ commits. This is because, when $CellMask_1$ is active, $WRITE(cell_1)$ includes $CellMask_1$ and $ANCESTOR(ChipMask)$ does not include $CellMask_1$. When $CellMask_1$ commits, $CellMask_1$ is replaced by $T_{CellMask}$ in $WRITE(cell_1)$. $ANCESTOR(ChipMask)$ does not include $T_{CellMask}$, too. ChipMask can read $cell_1$ only when $T_{CellMask}$ commits and $T_{CellMask}$ is replaced by T_{chip} in $WRITE(cell_1)$. This is because $ANCESTOR(ChipMask)$ includes T_{chip} .

• **Isolated vs. Cooperative:**

ChipMask cannot read $chip_layout$ produced by ChipDesign until ChipDesign commits, even though $\langle ChipDesign, chip_layout, R \rangle$ is created. This is because $ANCESTOR(ChipMask)$ does not include ChipDesign. The interleaving points of ChipDesign do not affect ChipMask. Therefore, SCC is able to guarantee the atomicity of ChipMask. Similarly, Document cannot read $chip_mask$ produced by ChipMask, until ChipMask commits. This is because $\langle ChipMask, chip_mask, R \rangle$ is created when ChipMask commits.

• **Cooperative vs. Cooperative:**

ChipDesign can read $logic_1$ produced by $CellDesign_1$ when $\langle T_{CellDesign}, logic_1, R \rangle$ is created. $T_{CellDesign}$ is $MLCA(CellDesign_1, ChipDesign)$. Due to the interleaving specifications of $T_{CellDesign}$ and $CellDesign$, $\langle T_{CellDesign}, logic_1, R \rangle$ can be created even when $CellDesign_1$ does not commit.



6. CORRECTNESS PROOF: A SIMPLE SKETCH

In this section, we provide characterization theorems that specify the set of transaction executions generated by SCC. Because the details of the full proof are beyond the scope of this paper, we only sketch the main ideas here. Our approach of the proof is similar to that used by Bernstein, Hadzilacos, and Goodman [10] to prove the correctness of conventional concurrency control schemes. We can determine whether a transaction execution history, H , is serializable by analyzing a graph derived from H called a *serialization graph*. The serialization graph for H , denoted $SG(H)$, is a directed graph whose nodes are transactions, and has edges $T_i \rightarrow T_j$ if one of T_i 's operations precedes and conflicts with one of T_j 's operations. A history H is serializable if and only if $SG(H)$ is acyclic [10].

The proof that SCC is correct proceeds by showing that (1) the executions of transactions with isolated work structures are serializable, and (2) every object is accessed through a serializable fashion, even though the total executions of transactions with diverse work structures are not serializable. Now, we describe a theorem that characterizes the executions of transactions with isolated work structures.

Theorem 1 (Characteristics of Transactions with Isolated Work Structures): In SCC, the executions of transactions with isolated work structures are serializable.

Proof Sketch. Consider a transaction execution history H produced by SCC. If $T_i \rightarrow T_j$ is in $SG(H)$, then T_i released some data before T_j accessed that data. If there is a nonempty path in $SG(H)$ from T_i to T_i (i.e., a cycle) then, by transitivity, T_i released a data before T_i accessed some other data. This is not true because every interleaving point of a transaction with isolated work structure is created when the transaction commits. So, $SG(H)$ is acyclic, and thus H is serializable. \square

An appealing implication of Theorem 1 is that SCC is fully compatible with conventional two-phase locking. Transactions with isolated work structures do not access the uncommitted results made by transactions with cooperative work structures. This means that, in SCC, conventional transactions (almost with isolated work structure) can be executed with essentially no additional overhead. We now describe another theorem that specifies the characteristics of object access fashion.

Theorem 2 (Characteristics of Object Access Fashion): In SCC, every object is accessed through serializable fashion.

Proof Sketch. Consider a transaction execution history H_{OBJ} produced by SCC that consists of operations on object, OBJ , only. If $T_i \rightarrow T_j$ is in $SG(H_{OBJ})$, then T_i released OBJ before T_j accessed OBJ . In this case, T_i does not execute another operation that conflicts with T_j 's operation on OBJ . This is true if T_i has isolated work structure, as was specified in Theorem 1. When T_i has cooperative work structure, T_j 's read operation on OBJ can be accepted after $\langle T_i, OBJ, R \rangle$ was created. Note that T_i creates $\langle T_i, OBJ, R \rangle$ when T_i does not update OBJ further. So, T_i does not execute another operation that conflicts with T_j 's read operation on OBJ . The same is true for T_j 's write operation on OBJ . Therefore, a cycle $T_i \rightarrow T_j \rightarrow T_i$ cannot be created. In general case, the acyclicity of $SG(H_{OBJ})$ can be proved by induction on the length of cycle. Since $SG(H_{OBJ})$ is acyclic, H_{OBJ} is serializable. \square

Theorem 2 means that every object is accessed through serializable fashion. The consistency of the object, therefore, is guaranteed. Note that the total history H may not be serializable. This is because an object may be accessed with conflict mode even though the owner transaction is not committed yet. So, the accesses could result in cascading aborts when the owner transaction fails. We recommend, therefore, only the tightly coupled transactions should have the cooperative work structure.

In CAD environments, an object can often be released independent of the terminating state (committed or aborted) of its owner transaction [3]. In this case, the degree of cascading aborts can be reduced because other transactions that accessed the object need not be aborted. We believe that this notion is very popular in practical CAD environments, and the interleaving points may have a role of indicating the possible moment to release the intermediate design objects.

7. CONCLUSIONS AND FURTHER STUDY

In this paper, we proposed a long transaction management scheme, named LOT, for coordinating design activities in CAD environments. The major contribution of LOT is to provide a facility for encapsulating the complexity of design activities from designers. Therefore, any designer does not need to know the details of the design activity, and does not need to concern whether he can release intermediate results of a transaction or not. Furthermore, the conventional transactions can be performed with essentially no additional overhead because the concurrency control scheme of LOT is fully compatible with conventional two-phase locking scheme. Currently, we implement LOT on top

of the first ever relational DBMS developed in Korea by KAIST, called Information Management (IM).

REFERENCES

1. K. Dittrich, "Controlled Cooperation in Engineering Database Systems," *Proc. 3rd Conf. on Data Engineering* (1987) 510-515.
2. N. Barghouti and G. Kaiser, "Concurrency Control in Advanced Database Applications," *ACM Computing Surveys* 22(3) (1991) 269-317.
3. P. Klahold and et al., "A Transaction Model Supporting Complex Applications in Integrated Information Systems," *Proc. 11th Conf. on ACM SIGMOD* (1985) 388-401.
4. F. Bancilhon and et al., "A Model of CAD Transactions," *Proc. 11th Conf. on VLDB* (1985) 25-33.
5. E. Yeh and et al., "Performance Analysis of Two Concurrency Control Schemes for Design Environments," *Information Sciences* 49 (1989) 3-33.
6. K. Goldman and T. Stout, "A Design Automation Environment," *VLSI Systems Design* (1988) 46-49.
7. J. Daniell and S. Director, "An Object Oriented Approach to CAD Tool Control," *IEEE Transactions on CAD* 10(6) (1991) 698-713.
8. M. Bushnel and S. Director, "Automated Design Tool Execution in the Ulysses Design Environment," *IEEE Transactions on CAD* 8(3) (1989) 279-287.
9. J. Moss, "Nested Transactions: An Approach to Reliable Distributed Computing", *Ph.D. Thesis, MIT/LCS/TR-260* (MIT Laboratory for Computer Science, Cambridge, Massachusetts, 1981).
10. P. Bernstein and et al., *Concurrency Control and Recovery in Database Systems* (Addison-Wesley, 1987).