A Data Management Model for Cooperative Design

Takeo KUNISHIMA

Feb. 15, 1991

Abstract

In a design process utilizing a Computer-Aided Design (CAD) system, many design data are created by designers or the CAD system; the descriptions of the design objects, their auxiliary data (their creation date etc.), input data for tests, and the results of the tests, etc. Since these data are often referred to at any time of the design process, they must be retained until the design is completed. For this reason, database systems must be combined with CAD systems in order to manage design data efficiently, and there are many papers on CAD databases.

CAD database systems must have new functions which are not required for traditional database systems. One of these functions is to support cooperative design. So large and complicated design object that it could hardly be designed by one designer is usually implemented cooperatively by a number of designers. CAD database systems must offer some functions which make the cooperation of the designers to be easy, such as the methods of sharing design data, the control methods for accessing design data, change notification methods, etc.

A conventional design database system gives a management method for cooperative design based on a hierarchy of databases. However, this management method has less flexibility of controlling transactions.

In this thesis, we propose a new data management model in the database hierarchy. First we define a concept of an *access control function*, which determines whether to permit a transaction to operate on a design data. It is defined for each design data as a function from U (a set of users), T (a set of transactions), DB (a set of databases), and an n-bit vector of *test flags*, each of which is provided for each test tool on CAD systems and reflects the result of the latest test of the test tools. We also show an extension of the function in order to distinguish two situations of testing: confirming its validity and specifying causes of errors.

Next we propose a concept of "*testTool dependency*", a binary dependency between two test tools whose test results always implys the others. We also mention the relationships between the access control function and this dependency.

We also show an example of the access control function and the *testTool* dependency under a user transaction model, which is similar to actual CAD databases.

Finally we consider several extensions of the access control functions and the *testTool* dependencies. We also consider a concept of user hierarchy and change notification methods on our data management model.

Contents

1 Introduction

In order to design large and complex hardware and software, use of CAD (Computer-Aided Design) systems is essential. Due to the recent requirement to realize large systems, database function must be combined with CAD systems.

In a design process utilizing a CAD system, many design data are created by designers or the CAD system; the descriptions of the design objects, their auxiliary data (their creation date etc.), input data for tests, and the results of the tests, etc. Since these data are often referred to at any time of the design process, they must be retained until the design is completed. For this reason, database systems are used in CAD systems in order to manage design data efficiently, and there are many papers on CAD databases [**?**, **?**, **?**, **?**, **?**, **?**].

CAD database systems must have new functions which are not required for traditional database systems. One of these functions is to support cooperative design. Large and complicated design object, which could hardly be designed by one designer, are usually implemented cooperatively by a number of designers. CAD database systems must offer some functions which lighten the designers' burden for cooperative design, such as the methods of sharing design data, the control methods for accessing design data, change notification methods, etc.

A conventional design database system gives a management method for cooperative design based on a hierarchy of databases [**?**, **?**]. The system provides a public database for all the designers, a group database for each group on the design process, and a private database for each designer. Design data which are shared among the members of a project are placed in the database of the project. Any of the members can execute any transactions to the shared data, and the other members cannot execute any transactions to them. If a design data is assumed to be error free, it is placed into the parent database in the hierarchy. On the other hand, if some errors are found in a design data, it is placed back into a child database in the hierarchy.

This management method has some merits to support cooperative design. First, it provides simple and powerful data sharing methods and transaction control methods. Second, it is suitable for distributed design environment, which has been popular with the spread of low-priced and powerful workstations.

However, this management method has less flexibility of controlling transactions. For example, it often happens that shared data in a group database can be read by any member of the group, but few of the members are allowed to update them. The conventional method cannot support such a transaction control, since permission to access design data depends on only the database the design data places, not on the kind of transactions.

In this thesis, we propose a new data management model on the database hierarchy. First we define a concept of an *access control function*, which determines whether to permit a transaction to operate on a design data. It is defined for each design data as a function from U (a set of users), T (a set of transactions), DB (a set of databases), and an n-bit vector of *test flags*, each of

which is provided for a test tool on CAD systems and reflects the result of the latest test using the tool. We also show an extension of the function in order to distinct two situations of testing: confirming its validity and specifying causes of errors.

Next we propose a concept of "*testTool dependency*", a binary dependency between two test tools whose test results always implys the others. We also mention the relationships between the access control function and this dependency.

We also show an example of the access control function and the *testTool* dependency under a user transaction model, which is similar to actual user transactions on CAD databases.

Finally we consider several extensions of the access control function and the *testTool* dependency. We also consider concepts of user hierarchy, change notification methods, and error notification methods on our data management model.

The remainder of this thesis is organized as follows. In Section 2 we summarize the cooperative design process on CAD systems. We also mention the conventional CAD database systems for cooperative design, and consider problems of such database systems in this section. We describe our data management model, the access control function and the *testTool* dependency in Section 3. In Section 4, extensions of our data management model are discussed. Section 5 is concluding remarks.

2 Database Systems for Cooperative Design

In this section, we summarize how cooperative design process on CAD systems can be realized, and consider the requirements for database systems to support cooperative design. We will also mention functions of conventional CAD database systems for cooperative design and their problems.

2.1 Cooperative Design Processes

Cooperative design process consists of two phases, first top-down design and then bottom-up design. Let us consider a case to design a lare system.

At first, the system to be designed is recursively divided into some subsystems. In most cases, each subsystem is a functional block which can be designed independently from each other. In cooperative design process, a group, which consists of some designers, is organized to implement each subsystem. A group may divide the subsystem into more smaller blocks, each of which is designed by a subgroup of the group. After dividing the system in such a way, design of some blocks is assigned to a designer, and he implements those blocks.

Since the subsystem which is assigned to a group uses the blocks assigned to its members, the group cannot complete design of the subsystem without the implementations of the blocks. For this reason, his blocks are shared among the members of the group when he is convinced that he found and fixed all errors in the blocks. The shared blocks are merged into an implementation of the subsystem, and the members of the group continue design of the subsystem with testing it a number of times.

When some errors are found in his implementation in the design process of the subsystem, it is prevented to access from the members of the group except him, and he tests and modifies it until the errors are corrected. When the group finishes correcting errors in the subsystem, it is shared among the members of a larger group, and design process proceeds similarly until whole the implementation of the system to be designed would be completely implemented.

A lot of design data are created during these design processes; descriptions of the specifications of the subsystems, its implementations, input data for tests, results of the tests, etc. A designer may refer to these design data in various ways: he may read other design data — whose owner may be other designer — or its auxiliary data, such as their names, the time of creation, the time of the latest update, their owners etc; he may copy or snapshot other design data; he may test the design data shared among other designers; he may create a new version of the design data.

In a design process, even the design data with errors may be accessed. For example, they may be referred to correct similar errors in another design data. All the design data must be remained until the process finished, in order to be able to refer them at any time during the process. Moreover, there must be some kinds of distinctions between the design data with no error and with some

errors, in order to prevent from designers' misuse of a design data with some errors.

We also notice that the referring data are in most cases under implementation. This means that design data which a designer is referring may be updated frequently. As a result, he may redesign his blocks with referring the new design data.

2.2 Requirements for Database Systems to Support Cooperative Design

As described in Section 1, it has been very popular to design circuits or softwares on CAD systems with databases in order to manage a lot of design data efficiently. There are various requirements for CAD databases to support cooperative design on CAD systems.

First, varieties of transactions are necessary for the cooperative design processes on CAD systems, as follows:

- reading design data
- updating design data if it is allowed
- reading the auxiliary data of designs
- updating the auxiliary data of designs
- copying design data into a database
- making a snapshot of design data
- making design data to be shared among the members who belongs a same group
- making shared design data not to be accessed from other designers of a group
- merging some design data to create a new design data
- testing design data with test tools and some test data

CAD database systems must provide these transactions.

Second, CAD database systems must have several functions to support these transactions.

1. a function to manage various design data

   Various design data are created on design processes on CAD systems, and CAD database systems must manage these data well. For example, CAD database systems must make some restrictions to the transactions to design data with some errors.

2. a function to get all the information necessary for his design processes, including other designers' data and their auxiliary data.

   In cooperative design, a designer frequently wants to refer others' design data during his design process. CAD database systems must provide some methods to support these references.

3. a function to prevent other designers to access his design without his permission.

   In contrast to the function 2, a designer has a requirement of preventing other designers to read his design data, update them, or copy them, etc, without his permission.

   If it is allowed to read his design data including some errors, other designers may read them and may design systems with errors. If a designer's data under implementation would be updated without his permission, he could not know what changes would be made in his design or why those changes would be made.

4. a function to notify the errors in his design data which other designers find.

   If some errors are found in a designer's data which he does not allow other designers to update, the errors must be notified to him to correct them.

5. a function to notify the changes of design data to the designers who have referred them.

   A design data which was implemented with referring other data is more or less influenced by the contents of the referred data. If the contents of the referred data are changed, the changes must be notified to all the designers who have implemented systems with referring them.

2.3 Conventional Database Systems for Cooperative Design

Conventional CAD database systems for cooperative design are constructed using a hierarchy of a public database, project databases, and private databases, as shown in Figure 1 [**?**, **?**].

The design data in the public database are accessible by any designer. The design data in a private database are owned and accessed by only one user. A designer places design data in his group database when he wants to share data with other designers who belong to the same group. Other designers of the same group copy them, update them, etc, as new versions in the group database.

Such a database system satisfies the requirements 3 and 5 in Section 2.2. The design data which are placed in a designer's private database are accessible only to himself, and he permits other designers in a same group to update his

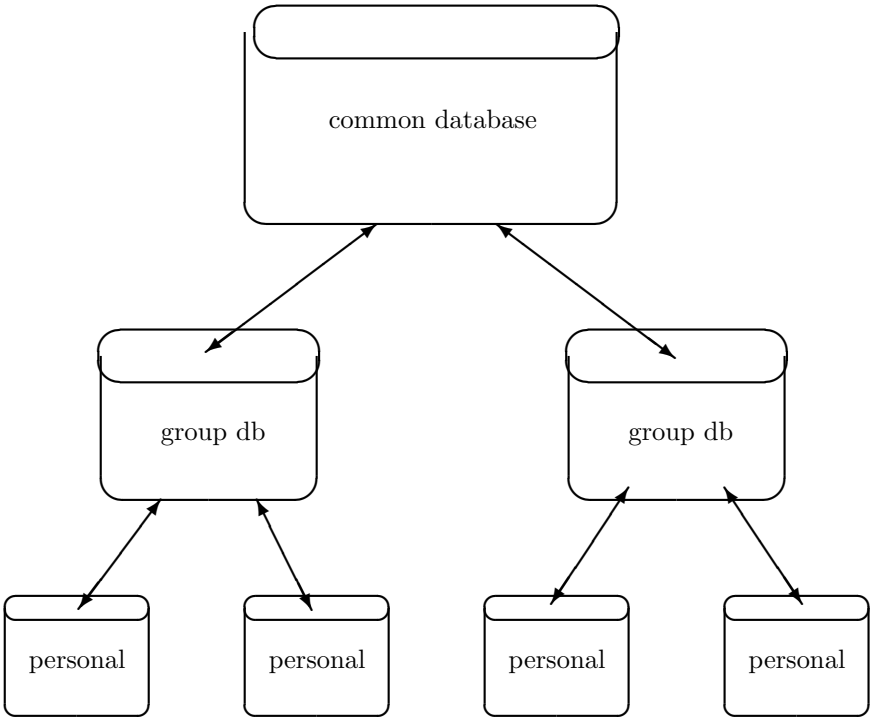


Figure 1 Conventional Database Model for Cooperative Design

design data by placing it in the group's database. In [**?**], message-based and flag-based change notification mechanisms are also provided.

However, there are several problems in this database structure. The first problem is that it does not always satisfy the requirement 2. All the design data a designer can access are in the public database, his private database, and the database of the group in which he belongs. He cannot get any information from the design data on other databases. Since it is difficult to divide a design object so that each subsystem could be implemented independently, CAD database systems must support more flexible data management methods for cooperative design that conventional methods.

The second problem is that it does not provide any methods for the requirement 4. The conventional model assumes that shared data in a group database have no error and have no need for testing. This assumption is invalid in actual cooperative design processes, because there is an requirement to share design data with some errors, for the reason that its errors cannot be fixed unless it is tested by the group.

The third problem is that methods for the requirement 1 are insufficient for actual cooperative design process. The conventional model provides no method but placing design data with some errors in a database, which can access only the members of a group. This method is insufficient for the same reason as that of the second problem.

3 A Data Management Model for Cooperative Design

In this section, we describe several methods necessary for cooperative design on CAD systems, summarized in Section 2.1. In the following sections, we suppose that a CAD database is constructed using a hierarchy of a public database, project databases, and private databases, as described in Section 2.3.

3.1 Preliminaries

In this section, we formally define several concepts in cooperative design processes.

First we give a formal notation of designers to define a concept "group".

定義 1 Define

$$U = \{u_1, u_2, \cdots, u_m\}$$

as a set of designers, where $u_i (1 \leq i \leq m)$ is a designer. □ □

In cooperative design, a number of designers relates to one project. Assumed that the members of each project are different from each other, we can define a concept of a project member, "group", as follows:

定義 2 Define

$$G = \{g_1, g_2, \cdots, g_n\}$$

as a set of project groups, where $g_i (1 \leq i \leq n)$ is a project group, which is a subset of U. □ □

定義 3 If two groups g and g' satisfy the following conditions

$$g' \in G \text{ s.t. } g \subseteq g', \text{ no group } g'' \text{ s.t. } g \subseteq g'' \subseteq g',$$

we call g' as a parent of g.
Define a child of group g as

$$g' \text{ is a parent of } g \iff g \text{ is a child of } g'.$$

□ □

In general, a group g has a number of parents and children.

定義 4 If two groups g and g' satisfy the following conditions

$$g' \in G \text{ s.t. } g \subseteq g',$$

we say g' as an ancestor of g.
If a group g' is an ancestor of a group g, we call g as a descendant of g'. □ □

3.2 Data Management Based on Test Results

3.2.1 Management of Test Results

In the design processes on CAD systems, many tools are used for tests, and the rank of validity for a design data can be defined by a set of test tools whose tests are passed. In the case of cooperative design, a designer is likely to decide whether to share a design data by what kinds of tests it has passed, so the information about test results is much important for design on CAD systems.

A designer often refers results of the latest test to a design data in order to decide whether to correct it. If the latest test was failed, he must investigate the causes of the errors and correct them.

However, results of the latest test is not enough to decide whether to share design data. He cannot make an exact judge from the results whether there are no more errors in it. For example, when design data passed design rule checking, which is the latest test, he convinces it to have no error against design rules. However, its results are insufficient to decide whether to share it as an released data, since he cannot get any information about other kinds of errors. Thus results of other tests are needed.

For this reason, we provide flags, called *test flags*, for each design data for the number of test tools. Each flag f_i means that the latest test using a test tool t_i is passed or failed. In general, each design data has different test flags on each other, since test tools applicable to each design data may be different from each other. Design data which are the same but in different databases may have different test flags.

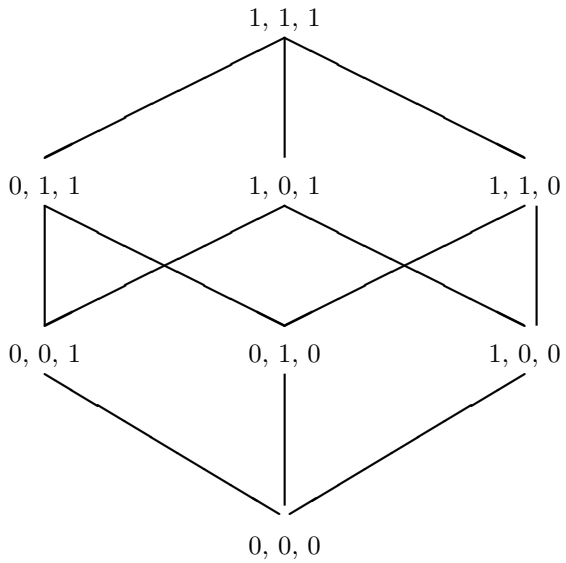
There are n test flags on a design data, which form a *test-flag vector* $tfs = (f_1, f_2, \cdots, f_n)$, if CAD systems has n test tools $t_1, t_2, \cdots, t_n$. A set of all the values of tfs equals to $\{0,1\}^n$, and organizes a lattice, called *tfs-lattice*, which is constructed as follows:

$glb(tfs_1, tfs_2) = tfs_1 \wedge tfs_2$
$lub(tfs_1, tfs_2) = tfs_1 \vee tfs_2$
where $\wedge$ and $\vee$ means bitwise-and and bitwise-or, respectively.

An example of tfs-lattice is shown in Figure 2.

Using tfs , more delicate data management can be done. For example, it can manage that only design data which passed tests with a certain number of tools are made to be sharable; or it also can do a management to make some kinds of tests to a design data to be valid only for the case that it passed a certain kinds of tests.

In general, it is possible to manage transactions on design data using a function $F(u, t, db, f_1, f_2, \cdots, f_n)$, where u is a designer, t is a transaction, db is a database in which the data places, and f_i is a test flag. F equals to 1 iff (if and only if) the transaction is valid. In the previous examples, F can be defined as a threshold function about the variables $f_1, f_2, \cdots, f_n$, and a function that



Each node of the lattice is a test-flag vector which means that:

$(simulation1, simulation2, designrulecheck)$

Figure 2 Lattice of Test-Flags Vectors

depends on only one variable, respectively. Hereafter we call F as an *access control function*.

3.2.2 Dependencies among Test Tools

In this section, we describe a management method on tfs-lattice for transactions.

We observe one characteristic about test tools on CAD systems. There are some cases that they have dependencies on each other. Examples are as follows:

例 1 Logic circuits which passed the tests of a logic simulator are expected to pass the tests of a design rule checker, since circuits which does not satisfy design rules cannot pass logic simulation. Then a designer might omit a design rule check to the circuits which passed logic simulation. □ □

例 2 It is reported in [**?**] that in some logic circuits, time-symbolic simulation gives more exact results than min/max delay simulation. This also means that logic circuits which passed tests of the min/max delay simulation always pass tests of the time-symbolic simulation. □ □

We formalize such dependencies among test tools to provide a concept "*testTool* dependency".

定義 5 If all the design data which passed tests using a tool t_1 are expected to pass tests of another tool t_2, we would say "there is a *testTool* dependency from t_1 to t_2", and notate as:

$$t_1 \longrightarrow_{tool} t_2.$$

□ □

例 3 In the previous examples 1 and 2, there are *testTool* dependencies

logic-simulator $\longrightarrow_{tool}$ design-rule-checker,
min-max-delay-sim $\longrightarrow_{tool}$ time-symbolic-sim,

respectively. □ □

If a test for a design data using test tool t_i has passed, value of a flag f_i becomes 1, and, a value of tfs changes the bitwise-or between tfs and the vector of length n that all but the i-th flag is 0. However, when test tool t_i has a *testTool* dependency to another test tool t_j, it can be considered that design data which passed the test of t_i would pass the test of t_j. In that case, test of t_j can be omitted, and design data whose f_i is 1 but f_j is 0 can be handled as if f_i and f_j are both 1. It is formalized by an access control function F in Section 3.2 as follows. If *testTool* dependency $t_i \longrightarrow_{tool} t_j$ exists, then

$$F\,\big|_{f_i=1,f_j=0} \equiv F\,\big|_{f_i=1,f_j=1}\ .$$

Though a *testTool* dependency $t_1 \longrightarrow_{tool} t_2$ exists, there are cases that tests of t_2 are required. One of the cases is that a designer uses t_2 and corrects as many errors as possible, to make tests of t_1 to be more simple.

3.3 Data Management after Testing

There are two different purposes to test design data on CAD systems:

1. to confirm that the design data has no error,
2. to specify the causes of the errors.

In the case 1, the design data's validity has increased if the test is passed. On the other hand, there are some errors in the design data in the case 2, and it has less validity than it had before the errors were found. We must distinct those two cases, since design data's validity is an important factor for deciding whether to access transactions on it.

We can distinct those two cases by results of the latest test. The case 1 occurs only when design data passed the latest test; on the other hand, the case 2 occurs only when the latest test was failed and causes of the errors have not been specified yet.

Then we provide a flag called *detectError* flag for each design data. If a designer has tested his design data to find some errors in its results, the value of *detectError* of the design data becomes 1. And if he finished specifying causes of the errors, the value of *detectError* becomes 0. In other words, if this flag equals to 1, the owner of the data is now specifying causes of the errors.

Using this flag, CAD database can distinct the cases 1 and 2, and can manage the same transaction in a different way. For example, it can do a management to make the design data to be accessible until the causes of some errors in it are specified.

This management method is formalized as a function $F'(u, t, db, f_1, f_2, \cdots, f_n, detectError)$, an extension of the access control function F in Section 3.2. In the above example, F' is defined as $F' = F \vee detectError$, where F is a function in Section 3.2.

3.4 A Management Model of Data Sharing

On a database hierarchy as mentioned in Section 2.3, each group has its own group database, in which design data are placed to be shared among the members of the group, and other designers cannot write into the group database.

Based on these facts, CAD database can support data sharing on a database hierarchy only to provide these two transactions: placing design data into a parent database, and placing design data into a child database. Then we provide two transactions in order to manage data sharing on the database hierarchy.

1. *up* transaction
   a transaction to place a design data into the parent database,
2. *down* transaction
   a transaction to place a design data into the child database for the data.

Here we use concepts a "child database" of a database for a design data in accordance with the following definition.

定義 6 Define a child database *child*(*db*) of database *db* for a design data d as a child database of *db* which is an ancestor of the designer of d. □ □

up and *down* transactions have some properties:

- *up* transaction on a design data is valid only when a group which owned the data regards it to be designed completely.
- When a *down* transaction is executed on a design data, the owner of the data is changed to the one who owned it before it was upped.

Using the management methods described in Sections 3.2 and 3.3, formalization of management methods about *up* and *down* transactions satisfying these properties is as follows.

A management method of *up* transaction is as follows. *up* transaction of a designer u will be accepted only if $F(u, up, db, f_1, f_2, \cdots, f_n) = 1 \wedge detectError = 0$ and he has an authority to write the parent database.

down transaction is managed as follows. *down* transaction only runs when *detectError* flag equals to 1. It is accepted only if a designer who executed the *down* transaction has an authority to write the child database for the downed data. A value of tfs is set to the one when it was upped.

3.5 An Example of the Data Management Model

In this section, we show an example of a data management model based on methods described in Sections 3.2, 3.3, and 3.4.

3.5.1 A User Transaction Model for Cooperative Design

In this example, we consider a user transaction model for cooperative design as follows:

- read(D)
  read design data D.
- update(D)
  update design data D, not creating a new version of D.
- create(D, V, DB)
  create a new version V of design data D into database DB.
- readinfo(D)
  read auxiliary data of design data D.

- writeinfo(D)
  rewrite auxiliary data of design data D.

- copy(D, V, DB)
  copy design data D into database DB as a version V.

- snapshot(D, DB)
  make a snapshot of design data D into database DB.

- up(D)
  make design data D to be shared among the members of the parent group.

- down(D)
  make design data D to be shared among only the members of the child for data D

- merge(D_1, D_2, $\cdots$, D_n, D)
  merge design data $D_1, D_2, \cdots, D_n$ as a design data D.

- test($TOOL$, D, $TDATA$)
  test design data D with a test tool $TOOL$ using test data $TDATA$.

3.5.2 A Data Management Model

In this section, we show an example of a data management model for the user transaction model described in Section 3.5.1, using management methods in Sections 3.2, 3.3, and 3.4.

First, the following user transactions are modeled without the management methods described in above sections.

- readinfo(D)
  This user transaction is always accepted for all designers.

- writeinfo(D)
  This user transaction is accepted only if it is executed by the owner of D.

Second, several user transactions are divided into some transactions as follows:

- copy(D, V, DB)
  This user transaction is divided into some subtransactions: read(D), create(D, V_0, DB). It is accepted only when both of them are accepted.

- snapshot(D, DB)
  This user transaction is also divided into some subtransactions: read(D) and create(D, V_0, DB). It is accepted only when both of them are accepted.

- up(D)
  This user transaction is divided into some subtransactions: read(D) and create(D, V_0, *parent*(DB)), where *parent*(DB) is the parent database of the database in which D is placed. It is accepted only when both of them are accepted and *detectError* of D equals to 0.

  When it is finished, the owner of D is changed to a group who owns *parent*(DB).

- down(D)
  This user transaction is divided into some subtransactions: read(D) and create(D, V_0, *child*(DB, D)), where *child*(DB, D) is the child database for the data D. It is accepted only when both of them are accepted.

  When it is finished, the owner of D is changed to a group who owns *child*(DB, D).

- merge(D_1, D_2, $\cdots$, D_n, D)
  This user transaction is divided into some subtransactions: read(D_1), read(D_2), $\cdots$, read(D_n), and create(D, V_0, DB), where DB is a database in which $D_1, D_2, \cdots, D_n$ are placed. It is accepted only when all of them are accepted.

- test($TOOL$, D, $TDATA$)
  This user transaction is divided into some subtransactions: read(D), read($TDATA$), and create(r, V_0, DB), where r is results of the test. It is accepted only when all of them are accepted.

  If the test would be passed, test flag of D which corresponds to $TOOL$ is changed to 1. If failed, the test flag is changed to 0, and *detectError* flag becomes 1.

Next, we model the transactions read(D), update(D), and create(D, V, DB) with a function F' in 3.3. Here we model them according to the following principles.

- read(D) is acceptable for designers who don't belong to the group owned DB as well as the members of the group.

- create(D, V, DB) is only acceptable for the members of the group owned DB.

- update(D) is only acceptable for a few of the members of the group owned DB.

Reasons are:

1. There may be a requirement that a designer implements his design with referring other groups' design data.

2. In the case 1, update of shared design data may affect a large number of design data. Then there must be more authorities to update shared design data.

We construct a function F' as

$$F' = F''(u, t, db, evalTest(f_1, f_2, \cdots, f_n), detectError).$$

That is, $f_1, f_2, \cdots, f_n$ are made to be independent of other variables.

Details of F'' is

- $F''(u, read(D), DB, evalTest, 0) = 1$ for all designer u
  $F''(u, read(D), DB, evalTest, 1) = 1$ iff u is a member of group(DB)
- $F''(u, create(D, V, DB), DB, evalTest, detectError) = 1$ iff u is a member of $group(DB)$
- $F''(u, update(D), DB, evalTest, 0) = 1$ iff u is a member of group(DB)
  $F''(u, update(D), DB, evalTest, 1) = 1$ iff u is a member of group$(child(DB, D))$

At last, a function $evalTest$ would be modeled. Suppose the CAD system has test tools t_1, t_2, t_3, $evalTest$ is a Boolean function of f_1, f_2, f_3. For example, in the case that design data must pass tests of at least two tools to be shared. $evalTest$ is formed as a threshold function

$$evalTest(f_1, f_2, f_3) = f_1 f_2 + f_2 f_3 + f_3 f_1.$$

If there is a $testTool$ dependency $t_1 \longrightarrow_{tool} t_2$, then $evalTest$ is

$$evalTest(f_1, f_2, f_3) = f_1 + f_2 f_3 + f_3 f_1.$$

4 Extensions of the Management Model for Cooperative Design

In this section, we will discuss several extensions of our data management model described in Section 3.

4.1 Extensions of Dependencies among Test Tools

We proposed *testTool* dependencies, dependencies among test tools, in Section 3.2.2. The meaning of a *testTool* dependency $t_i \longrightarrow_{tool} t_j$ is that all the design data which passed tests using a tool t_i are expected to pass tests of another tool t_j, by Definition 5.

There are many ways of extensions for this dependency. First, we can consider more dependencies among test tools on CAD systems.

- If a design data passed a test of tool t_i, it can be tested by another test tool t_j.
- If a design data passed a test of tool t_i, it cannot be tested by another test tool t_j.

Second, we can extend the dependencies, which is binary, to the non-binary ones between two sets of test tools.

Then we extend the *testTool* dependency as follows:

定義 7 Define an *extended testTool dependency* as

$$t_{i_1} t_{i_2} \cdots t_{i_n} \longrightarrow_{tool(kind)} t_{j_1} t_{j_2} \cdots t_{j_m}$$

where $t_{i_1}, t_{i_2}, \cdots, t_{i_n}, t_{j_1}, t_{j_2}, \cdots, t_{j_m}$ are test tools, and *kind* is one of the following commands.

- *omit*
  tests of $t_{j_1}, t_{j_2}, \cdots, t_{j_m}$ can be omitted if tests of $t_{i_1}, t_{i_2}, \cdots, t_{i_n}$ are all passed.
- *permit*
  tests of $t_{j_1}, t_{j_2}, \cdots, t_{j_m}$ are permitted if tests of $t_{i_1}, t_{i_2}, \cdots, t_{i_n}$ are all passed.
- *prohibit*
  tests of $t_{j_1}, t_{j_2}, \cdots, t_{j_m}$ are prohibited if tests of $t_{i_1}, t_{i_2}, \cdots, t_{i_n}$ are all passed.

□ □

These dependencies are written by the access control function F in Section 3.2.2 as:

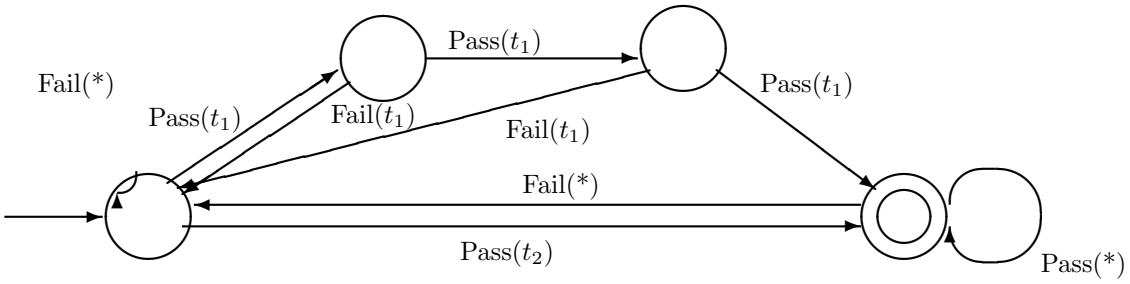


Figure 3 An Example of a Design Process Automaton

- the case of $kind = omit$
  It is as same as in Section 3.2.2.
- the case of $kind = permit$
  For each t in $\{t_{j_1}, t_{j_2}, \cdots, t_{j_m}\}$,

$$F(u, t, db, f_1, f_2, \cdots, f_n) = 1 \iff f_{i_1} \wedge f_{i_2} \wedge \cdots \wedge f_{i_n} = 1.$$

- the case of $kind = prohibit$
  For each t in $\{t_{j_1}, t_{j_2}, \cdots, t_{j_m}\}$,

$$F(u, t, db, f_1, f_2, \cdots, f_n) = 0 \iff f_{i_1} \wedge f_{i_2} \wedge \cdots \wedge f_{i_n} = 1.$$

4.2 Design Process Automaton

Data management by the access control function F in Section 3.2.2 is based on results of the latest test of each test tools. Then it cannot support such a case that design data can be shared if it passes tests of a tool more than n times. This situation often occurs in a design process of CAD systems since designers repeats tests of a test tool with various input data. In order to manage such cases, we define a design process automaton for each design data.

定義 8 Define a *design process automaton* (*DPA*) for each design data as

$$DPA = (I, S, S_0, \delta)$$

where I, S and S_0 are finite, nonempty sets of inputs, states, and initial states, respectively; $\delta : I \times S \to S$ is the state transition function. □ □

Figure 3 shows an example of *DPA* of a design data. This *DPA* manages a design process that the design data must pass tests using t_1 more than three times to be shared, but it may pass a test using t_2 only once.

We can construct F by *DFA* with the following S, I, S_0 and δ:

- S is a set of tfs $(f_1, f_2, \cdots, f_n)$.
- I is a set

$$\{Pass(t), Fail(t) | t \in T, T \text{ is a set of all test tools } \}.$$

  If $F(u, t, db, f_1, f_2, \cdots, f_n) = 1$, then DPA makes a transition by $Pass(t)$. If $F(u, t, db, f_1, f_2, \cdots, f_n) = 0$, then it makes a transition by $Fail(t)$.
- S_0 is a set of tfs , $\{(0, 0, \cdots, 0)\}$.
- δ is defined as

$$\delta((f_1, f_2, \cdots, f_i, \cdots, f_n), Pass(t_i)) = (f_1, f_2, \cdots, 1, \cdots, f_n)$$
$$\delta((f_1, f_2, \cdots, f_i, \cdots, f_n), Fail(t_i)) = (f_1, f_2, \cdots, 0, \cdots, f_n)$$

4.3 User Hierarchy

In both the conventional management model and our model in Section 3, the members of a group are modeled to have the same authority over design data. In actual cooperative design, however, they may have different authorities on each other. For example, each group usually has a leader as a manager of the project of the group. The leader usually has more authority over design data than normal designers in order to do smooth managements. He may be able to read some design data which is not accessible for normal designers, such as the ones in a private database of one of the members.

Then we consider the way to manage such situations in this section. First we define orderings between the members of a group.

定義 9 For each users u_i and u_j in the group,

$$u_i \succeq u_j \iff F(u_i, t, db, f_1, \cdots, f_n) \geq F(u_j, t, db, f_1, \cdots, f_n)$$

for all transactions t, where F is the access control function of a design data D.

In particular,

$$u_i = u_j \iff F(u_i, t, db, f_1, \cdots, f_n) = F(u_j, t, db, f_1, \cdots, f_n)$$

for all t. □ □

This ordering is partial, since a set of executable transactions of each designer may be different from each other. Then a set of all members of the group forms a hierarchy. We call it as a *user hierarchy*.

We find a simple example of user hierarchy in an example in Section 3.5. Since $F''(u, read(D), DB, evalTest, 1) = 1$ iff u is a member of group(DB), $u_j \succeq u_k$, where u_j is a member of $group(DB)$ while u_k is a designer without group(DB). On the other hand, since $F''(u, update(D), DB, evalTest, 1) = 1$ iff u is a member of group$(child(DB, D))$, $u_i \succeq u_j$, where u_i is a member of group$(child(DB, D))$ while u_j is a member of group(DB). Then $u_i \succeq u_j \succeq u_k$.

4.4 Change Notification

As described in Section 2, changes of a design data may affect the validity of other design data in cooperative design. So notification of the changes is very important.

To consider methods for change notification, we find a problem to whom the changes are notified. Here we take an approach to notify the changes to the designer who referred the changed data. The reason is that CAD systems cannot decide precisely whether to correct the affected data or not, and the designer must make some decisions.

In order to lighten designers' burden for searching the affected data, the change notification must include information which data is affected. Then CAD system must manage information which data may be affected by the change of a design data. We propose a concept of *referred data* for a design data.

定義 10 Define a set of *referred data* for a design data D as:

1. All design data which are created by transactions *create*, *copy*, *merge*, and *test* to D are referred data.

2. All design data which are designed using results of transactions *read* and *readinfo* to D are referred data.

3. All design data which are created by transactions *create*, *copy*, *merge*, and *test* to a referred data for D are referred data.

4. All design data which are designed using results of transactions *read* and *readinfo* to a referred data for D are referred data.

□ □

We notice that, in the cases 2 and 4, referred data cannot be decided automatically, because a designer may implement some design objects simultaneously.

A design data which is a referred data for D may be affected by the change of D. Since design data which had referred the changed data may even be affected, we provide a method of notifying all referred data for D when D is changed.

We provide an *affected event queue* for each design data. If a design data D have changed, the system sends a message, which includes information which data has changed, to all the referred data for D. The message appends to the affected event queue of D. It doesn't send to the owner of D directly, because the changes may affect to older versions, which has no need to update.

When a designer u accesses the affected data AD, all messages in its affected event queue would be managed, the results of the management would be sent to the owner of AD, and the owner would correct AD according to the received results. On the other hand, the system sends to u a message to keep his transaction waiting.

4.5 Error Notification

In our data management model, in contrast with the conventional approach, a designer u can test a design data if the access control function $F(u, test, db, f_1, \cdots, f_n) = 1$, even in the case he doesn't belong to a group which owns it. If he cannot update the design data in which he found some errors, CAD system must provide a method to notify these errors to designers who can update it.

If $F(u, test, db, f_1, \cdots, f_n) = 1 \wedge F(u, create, db, f_1, \cdots, f_n) = 0 \wedge F(u, update, db, f_1, \cdots, f_n) = 0$, the system sends a message to the owner of the tested data what kind of tests were executed, and what the results were.

5 Conclusion

In this thesis, a new data management model for cooperative design has been presented. It is based on a database hierarchy which is used in conventional data management models, but it has more flexible management methods than the conventional ones.

First an *access control function* has been defined, which manages several user transactions in a flexible way mainly by results of the latest tests of each test tools on CAD systems. We have also shown an extension of the function in order to manage data considering meanings of tests.

Next we have proposed a concept of *testTool dependency*, a binary dependency between two test tools whose test results always implys the others. We have also mentioned the relationships between the access control function and this dependency.

Finally several extensions of the access control function and the *testTool* dependency have been discussed. We have also considered a concept of user hierarchy and change notification methods on our data management model.

In future we will research more characteristics about the access control function, the *testTool* dependencies, and their extensions. In particular, we will research the ways to construct them for various requirements on CAD systems, and the mathematical properties on them.

We will also research how to support constructing our model. Our model has more flexibility, but it is more difficult to organize them into data management systems of some CAD systems. Researches about the way of support this organization process, particularly about user interfaces, are important for its practical use to CAD database systems.

Acknowledgements

The author is much grateful to Prof. Shuzo Yajima, for his giving the author a chance of doing this research, and for his helpful comments and suggestions.

The author is also grateful to Prof. Yahiko Kambayashi, Integrated Media Environment Experimental Laboratory, Kyoto Univ., for his incisive comments and suggestions.

The author thanks Associate Prof. Hiromi Hiraishi, Dr. Naofumi Takagi, Mr. Nagisa Ishiura, Mr. Hiroyuki Ogino, and the members of the Yajima Laboratory for their useful comments.

Appendix

List of Publications by the Author

1. T. Kunishima, S. Matsumoto, H. Ogino, H. Hiraishi, and S. Yajima: *Computer-Aided Design using Multi Screen Graphic MCMS System.* IPSJ Proc. Symposium on Graphics and CAD (Nov. 1989), 227–236, in Japanese.

2. T. Naruse, T. Kunishima, and T. Saito: *Consideration on Setting up Bounding Volumes.* IEICE Reports of Technical Group on Pattern Recognition and Understanding, PRU89-65 (Oct. 1989), 53–60, in Japanese.

3. T. Naruse, T, Saito, and T. Kunishima: *On an Algorithm Obtaining an Approximated Minimum Ball Containing n Balls.* Trans. IEICE, Vol. J73-D-II, No. 10 (Oct. 1990), 1792–1795, in Japanese.

4. T. Kunishima: *A Data Management Model for Cooperative Design.* Proc. Biannual Meeting of Studies on Object-Oriented Data Models for High Level Database Applications, Japanese Scientific Grant-in-Aid for Cooperative Research (A) (Jan. 1991).