An Extended Relational Database System and It's Application to Management of Logic Diagrams

Yosihisa Udagawa and Tetsuo Mizoguchi

Information Systems & Electronics Development Lab.
Mitsubishi Electric Corporation
5-1-1 Ofuna, Kamakura City, Kanagawa, 247, Japan

Abstract

An extended relational database and its application to VLSI data management are described. The following shortcomings mainly arise when we try to apply conventional DBMSs to management of engineering data : lack of facilities to support various data types, a design hierarchy, version control and efficient access to design data. We have extended the relational data model by supporting (1) a set of graphics functions that bridge relational data structures to graphics representations, (2) a special data type that allows a designer of the database to model the design hierarchy (including versions) explicitly. Ordinary relational databases appear to be too slow to be used on-line in a design process. Our hypothesis is that it does not come from the nature of the relational model, but rather internal access mechanisms. We have developed our system in such a way that data on a design object are stored as interrelated collections of records and are manipulated as a logical group. As a result, our system offers sufficient facilities and performance for interactive use of design data at design system level.

1. Introduction

As the complexities of electric circuits increase, an information management component is becoming crucial to the success of a computer aided design (CAD) system. In most cases, the CAD systems that exist today each use their own data format. This often requires data translation each time a new CAD tool is added to the design system. Data translation is adverse to integration of CAD tools because it increases not only the difficulty to maintain integrities of design data but the amount of data translation software that must be written and maintained [7,14,18,25,27].

Database management systems support such facilities as uniform access to data on secondary storage, dynamically changeable schemas, concurrent access mechanisms for data sharing. Placing all design information under the control of a single database management system make it easier to maintain the self-consistency. Design database have long been of interest in CAD community and a number of design systems have been build upon CODASYL database systems with sufficient performance. CODASYL systems, however, provide operations to access a record at a time and do not adequately isolate the design management software from the physical data organization. Further, a CODASYL database schema is static. Since tools are closely tied with database schema, new tools can not be added without a major reorganization of the database [3,23]. The relational approach, on the other hand, offers the dynamic view of the data necessary to share data between application programs. Further, relational systems support a set of operations to manipulate relations at a time. These achieve high data independence, i.e. physical data structures can be changed without affecting existing programs. Hence, many specialists on database systems agree that the relational approach is the most promising one to design data management. Some design systems have been build on relational database systems, however, only with limited success. Conventional relational database systems are primarily designed for business applications that only deal with formatted alphanumeric data. Thus, these database systems fail to support many of the facilities needed in design activities. In particular, the missing features include (Katz[17] presents a more detailed discussion.) :

(1) support for various data types, such as pictures, images,
(2) explicit representation of a design hierarchy,
(3) support for version control,
(4) efficient access to design data.

VLSI circuits are a particularly interesting design domain because of the rich set of representations used in their description. These representations may include system architecture, register transfer, logic diagrams, Boolean expression and process specification. We limit the scope of this paper to management of logic diagrams only. In particular, we concentrate on a logical structure and graphical representation. Circuits are usually represented as block diagrams, i.e. named boxes with input/output signals to denote data and control flow. The key information that the block diagram presents is how subsystems are wired together. Because the block diagram is naturally created and viewed graphically, database systems must be able to support geometric data [11,12, 17,28].

Hierarchical levels of abstraction are essential for tackling any complex problem. VLSI design is no exception. VLSI systems can be conceptualized as a complete design, and then recursively decomposed into more primitive pieces. The leaves of the hierarchy represent the detailed specification of primitives that constitute the design. To support design data adequately, the design database must reflect this hierarchical levels [13,16,23, 28].

Versions are objects that share the same interface descriptions and differ only in their implementations. A hierarchical design approach provides a conceptual partitioning of the design. An engineer is responsible for a conceptual unit of the design. The concept of versions allows the engineer to proceed as he wishes within the conceptual unit, as long as his implementation does not alter the specified interface [17,19].

CAD applications require object-oriented accessing and manipulation. Typical design objects are stored as interrelated collections of records and are manipulated as a logical group. While fast access to records is important, the overhead of accessing a record at a time seems too great for the large quantities of data. It is more efficient to extract and process large aggregations of design data as a unit [1,20,29].

In this paper, we discuss an extension of the relational model and its application to management of logic diagrams. A number of design systems have been implemented upon the relational techniques, notably System R at IBM Research, San Jose [13, 20], INGRES at the University of California, Berkeley [11], VDD, at AT&T Bell Laboratories [5]. Each system takes different ways to provide better support for design applications : System R with complex objects, INGRES with abstract data types and VDD with rather specific approach to VLSI design. While many researchers have proposed novel data models for CAD applications, little work has been done to design a coherent data model which supports all the requirments peculiar to CAD applications. The coherent data model is needed to understand the basic requirements of database support for VLSI applications and how existing database techniques can be applied, refined or generalized for the engineering environment. In our research and development efforts, we believe that we are making two valuable contributions to the understanding of design databases. One is the development of a coherent data model, termed **ADAM** (Advanced Database with Abstraction Mechanism), which supports the requirements (1),(2) and (3) mentioned above. Another is an implementation of the ADAM with good performance in the design environment.

This paper is organized as follows. We begin by outlining the ADAM data model in terms of the design hierarchy, graphical representation, version control and its implementation. In Section 3, we discuss use of ADAM data model for managing logic diagrams and describe the structure of the database itself. Section 4 deals with implementation of the logic diagram management system and performance. We close the paper with conclusions and future works.

2. Over View of ADAM Data Model and Its Implementation

2.1 Design Hierarchy

A design object consists of its interface description and implementation description. The interface description should contain enough information about the object so that it can be used without having a detailed understanding of its implementation. It usually contains such information as overall functions of the object, input and output connection points, geometric boundary. The implementation description specifies how the design object is composed of more primitive components. It typically includes listing of components, their interconnections and graphical representations.

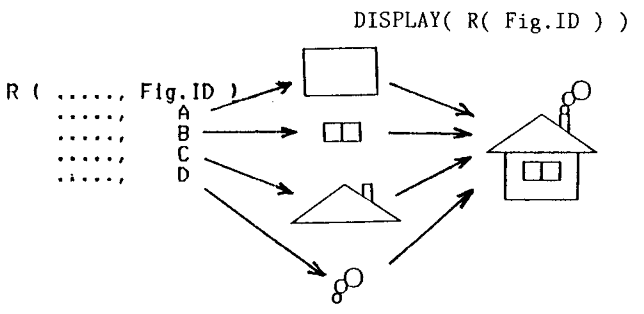
The skeletal structure of ADAM data model is a directed acyclic graph of design objects. Vertices represent objects. Leaves are primitive objects that have no implementation descriptions. Vertices but leaves mean objects that are formed from the recursive composition of its descendants in the graph. Edges are directed from an object to its component objects. Each vertex include the following specifications.

(Name of object)
(Parameters of object)
(Interface description for structure)
(Interface description for graphical notation)
(Implementation description for structure)
(Implementation description for graphical notation)

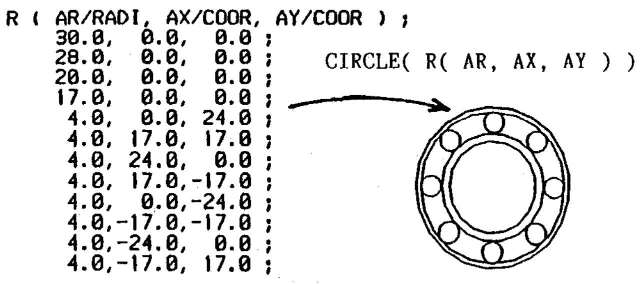
(Name of object) declares a name of the design object. The design object can be parameterized with geometric transformation information describing how it should be placed or rotated. In ADAM, a special data type, termed "abstract object", is devised to identify the parameterized object. An instance of the abstract object data type , having a syntax <identifier> (<parameter>, [<parameter>, ...]), is a straightforward counterpart of the "molecular object" addressed by Batory and Kim [2]. <identifier> is the name of the design object and must be unique in the design hierarchy. Structural description of the object, i.e. input/ output ports, listing of components and their interconnections, is described in terms of the relational data model. However, the usual collection of relational domains has been extended to include the abstract object type. Components of the design object are listed in a relation involving this special domain type.

2.2 Graphical Representation

The graphical representation provides engineers with outline of the design object for viewing on a graphics display. In ADAM, three types of graphics functions are invented to bridge the structural description and the graphical representation. One is a graphics function that manipulates named graphical segments are stored in an image file. The name of the segment may be listed in a relation. The other is a set of functions that draw a parametrized pictures, e.g. circle, triangle. Values of the parameters may be stored in a relation. Fig. 1 illustrates the general relationships between data in a relation and their graphical notation. The graphics functions mentioned above have a conspicuous feature, i.e. they ensure that the displayed notation is an up-to-date representation of the structural description. These functions are useful to specify both the interface and implementation descriptions of the

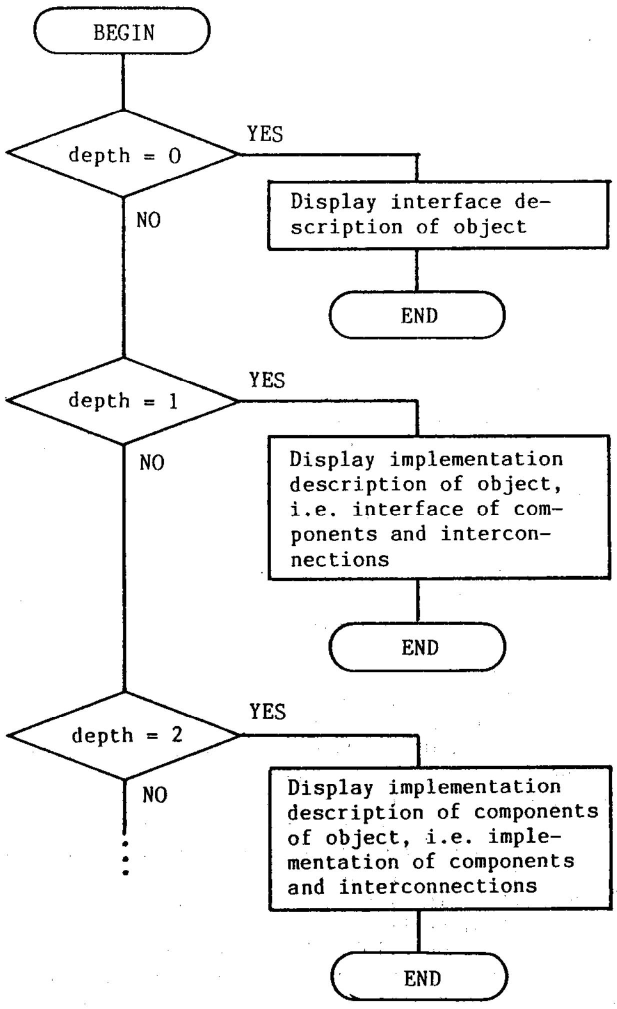


(A) Displaying named segments



(B) Drawing parameterized pictures

Fig.1 Graphics functions in ADAM.



BEGIN
depth = 0
YES
NO
Display interface description of object
END
depth = 1
YES
NO
Display implementation description of object, i.e. interface of components and interconnections
END
depth = 2
YES
NO
Display implementation description of components of object, i.e. implementation of components and interconnections
END

Fig.2 General Algorithm to view various hierarchical levels.

objects graphically.

Another type of the graphics function supports a facility of letting designers show lower-level (i.e. more detailed) views of the design objects in context of the overall graphical notation. The design object is usually defined by less complex components and their interconnections, where each component is assigned its own interface and implementation descriptions. Thus, to support the facility, a sophisticated algorithm has to be developed to display the structures of the related objects in a hierarchy, i.e. the underlying object and its components and so on. Fig. 2 gives a general idea of the algorithm. For more details, Udagawa and Mizoguchi [28] can be referred.

2.3 Version Control

With designs proceeding to more concrete ones, engineers typically generate or use equivalent design objects using different technologies (TTL, CMOS etc.), different physical characteristics (consuming power or area etc.). Thus, design databases prefer to support a special mechanism, i.e. version control, to manage these design objects efficiently. Though many researchers have addressed the problem of version control, many crucial issues are still not well understood or clearly defined partly because the absence of a coherent data model for design objects [2,17]. In this paper, we simply define versions as objects that share the same interface but have different implementation. Version control offers a mechanism by which convenient groupings of versions can be formed.

Logic circuits are typically divided into about ten categories ,e.g. gates, counters, registers. Each category of the circuits has its own attributes that are typically related to the interface description not the implementation description of circuits. For example, counters can be featured in terms of count type (binary, decade, etc.), count frequency, data load type (synchronous or asynchronous) and so on [26].

For the purpose of version control, our system provides an entry for each category. Each entry contains information that features the circuits in terms of relations. In our system, the design hierarchy is organized in such a way that each design object can be identified through the entries. Though our system only supports primitive facilities for version control, we believe our version control facility is valuable in the sense that it is based on a coherent data model and is implemented with good performance.

2.4 Implementation

The use of a database system in design applications has the potential to free its users from routine data management tasks. However, existing commercial database systems appear to be too slow to be used on-line by engineers in a design process. A relational database is no exception.

Wilkins et al.[29] and Barabino et al.[1] make the claim that the relational data model in itself is not a critical factor in performance, rather internal access mechanisms of the database significantly affect performance. As we have mentioned earlier, design applications require object-oriented accessing and manipulation. A direct implementation based on a conventional relational database system results in suffering from poor performance primarily because the overhead of entering and leaving the database system to extract a record at a time is too great for large quantities of data involved. So it is quite natural to think about accessing related design data as a unit and manipulating them in main memory.

Fig.3 shows an overall architecture of our system that has been implemented in this direction. Our system runs on a MELCOM COSMO 900 II computer system that has approximately 4 MIPS processing power. ADAM is approximately 10000 lines of FORTRAN code. "Management Information on DBMS" in Fig.3 includes information about the skeletal structure of ADAM, names and parameters of objects in the structure. The "Relation File" stores a relation or a set of homogeneous records. The "Picture File" includes a set of graphics segments defined by a sequence of graphics functions, while the "Image File" involves a set of digitized images. In our system, units of access to secondary storage are the "Relation File", "Picture File" or "Image File." Extracted data are then processed primarily in main memory allowing various design tools to handle them. A logic diagram editor **EVE** (**E**ditor in **V**isual **E**nvironment) is constructed on top of ADAM. EVE, approximately 3000 lines of FORTRAN code, supports facilities need to edit a logic diagram adequately, e.g. retrieving components from a design database, connecting components, checking validity of the diagram. Our system appears to provide sufficient performance at design system level. Typical operations to edit a logic diagram are executed in less than 100 msec.

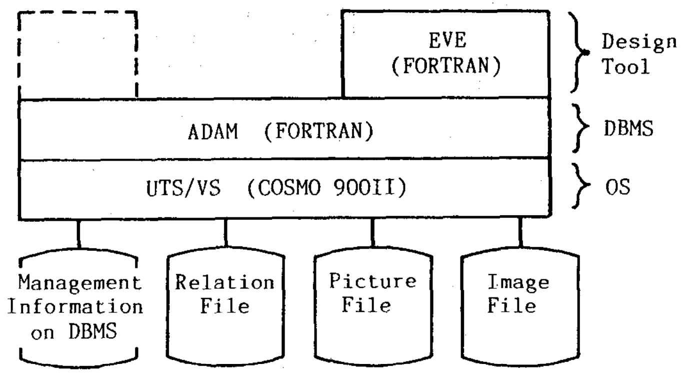


Fig.3 Overall architecture of the system.

3. Use of ADAM for Managing Logic Diagrams

3.1 Describing Structural Information

In this section, we will discuss how structures of a logic diagram are represented in terms of relations in ADAM data model. As we mentioned earlier, design objects have two levels of description, i.e. an interface and implementation. Both levels are modeled using standard entity-relationship (ER) techniques [4]. The ER model views the world as consisting of entities and relationships among entities. An entity is something that exists in the world. Entities and relationships have attributes that describe them.

Fig. 4 shows the interface (in broken lines) and the implementation (in solid lines) of a circuit that is composed of a 2-input NAND gate, a switch and resistors, etc. An ER schema of circuits is shown in Fig.5. In this figure, "*COMPONENTS" means a parent circuit that is constructed from child circuits ("COMPONENTS"), points and wires. "*PORTS" specifies an associated interface, i.e. a list of connection points that are used when an instance of the circuit is incorporated as a child within a higher level parent circuit. "*PORTS" and "PORTS" are disjoint subsets of "POINTS" and this is depicted in the schema using "DISJOINT" category [10]. The translation of the schema to relations in ADAM is straightforward. Rules for mapping ER schemas to Codd's relations are discussed in Elmasri et al.[10]. Note,

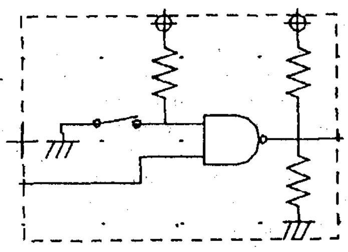


Fig.4 An example circuit.

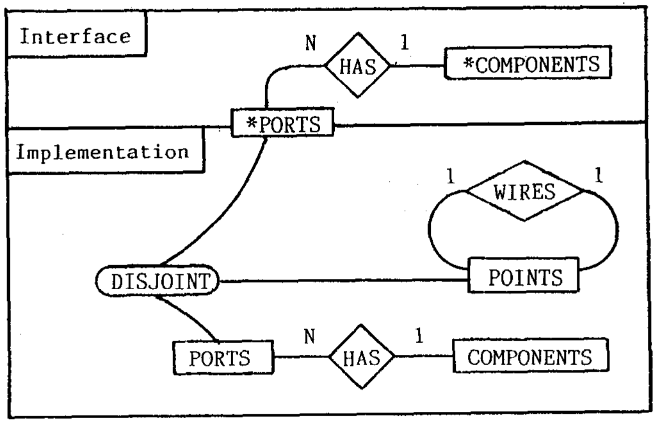


Fig.5 An ER schema of a circuit.

however, in ADAM data model, domains of a relation may include "abstract object" mentioned in Section 2.

The ER schema in Fig.5 is reduced into relations below.

```
COMP( DIV_ID/ID, DIV/ABS )
TERM( T_ID/ID, DIV_ID/ID, CHR/ID,
      XC/AD, YC/AD )
CONN( L_ID/ID, S_ID/ID, DIV_S/ID,
      D_ID/ID, DIV_D/ID )
```

Each tuple in the COMP relation represents an instance of a circuit incorporated within the parent circuit. Each item in the DIV column encodes an identifier of the instance and geometric placement. The TERM relation lists information about points of the circuit. The CONN relation describes wires of the circuit. It contains one tuple per wire. Fig.6 gives a collection of relations that describe the circuit in Fig.4. Each tuple including "D0" in the relations indicates that it specifies the interface of the circuit. Fig.7 shows relations describing the 2-input NAND gate. Since the NAND gate is a primitive component in our logic diagram management system, the relations in Fig.7 only consist of tuples specifying the interface description.

Haskin and Lorie have investigated an extension of the relational database for VLSI design [13]. In their database, the hierarchical structures among the design objects are described implicitly in the sense that the hierarchy of the design object is encoded by so called "referential integrity," i.e. a tuple for a child object must include an identifier for its parent object [9]. In our database, on the other hand, each design object is encapsulated in the sense that the structural and graphical information of the object is stored in one conceptual partition. Each partition is uniquely identified by an instance of the "abstract object" mentioned in Section 2. No partition is accessed which need not be. By virtue of the explicit management of the hierarchy, our system achieves high performance, which will be discussed in Section 4 of this paper.

3.2 Specifying Graphical Representation

Graphical representation offers a convenient way for engineers to understand structures of a logic diagram. Many geometric models consist of a mixture of data structures and procedures [21]. ADAM data model is no exception. ADAM's graphics facilities consist of the relational data structure supplemented by some graphics functions. The following SQL like formulas [9] define the graphical

```
COMP ( DIV_ID/ID, DIV/ABS ).
       D001, GND(+5.0,+0.0)
       D002, GND(+35.0,-10.0)
       D003, SW2(+12.0,+2.0)
       D004, NAT2(+31.0,+0.0)
       D005, REGV(+18.0,+8.0)
       D006, REGV(+35.0,+8.0)
       D007, REGV(+35.0,-5.0)
       D008, VCC(+18.0,+14.0)
       D009, VCC(+35.0,+14.0)
       D0, SWD(X,Y)

TERM ( T_ID/ID, DIV_ID/ID, CHR/ID,
       XC/AD, YC/AD ).
       I0, D001, TP_N, +5.0, +1.0
       I0, D002, TP_N, +35.0, -9.0
       I0, D003, TP_N, +9.5, +2.0
       I1, D003, TP_N, +14.5, +2.0
       Q0, D004, TP_N, +31.0, +0.0
       I0, D004, TP_N, +23.0, +2.0
       I1, D004, TP_N, +23.0, -2.0
       I0, D005, TP_N, +18.0, +5.0
       I1, D005, TP_N, +18.0, +11.0
       I0, D006, TP_N, +35.0, +5.0
       I1, D006, TP_N, +35.0, +11.0
       I0, D007, TP_N, +35.0, -8.0
       I1, D007, TP_N, +35.0, -2.0
       Q0, D008, TP_N, +18.0, +14.0
       Q0, D009, TP_N, +35.0, +14.0
       I000, D0, TP_N, +0.0, -5.0
       Q000, D0, TP_N, +40.0, +0.0

CONN ( L_ID/ID, S_ID/ID, DIV_S/ID,
       D_ID/ID, DIV_D/ID ).
       L001, I0, D003, I0, D001
       L002, I1, D003, I0, D004
       L003, I000, D0, I1, D004
       L004, Q0, D008, I1, D005
       L005, I0, D005, I1, D003
       L006, Q0, D009, I1, D006
       L007, I0, D006, I1, D007
       L008, Q0, D004, Q000, D0
       L009, I0, D007, I0, D002
```

Fig.6 Relations describing a circuit in Fig.4.

```
/*****************************************/
/*                                       */
/*          DEFINITION OF                */
/*     ABSTRACT INSTANCE 2-INPUT NAND    */
/*     WITH TOTEM-POLE OUTPUT            */
/*                                       */
/*****************************************/
/**/
$VARIABLE-DECLARATION ;
     X    /  REAL  ;
     Y    /  REAL  ;
$ABSTRACTED-INSTANCE ;
    NAT2  ( X, Y ) ;
$GENERAL ;
 *GDP ;
        .ORI( X, Y ) ;
        CIRCLE( 0.5, -0.5, 0 ) ;
        APX( 3, -4, 0, 3, -3 ) ;
        POLY((-4,3),(-8,3),(-8,-3),(-4,-3));
        LINE( -9, 2, -8, 2 ) ;
        LINE( -9,-2, -8,-2 ) ;
        .V( X,Y ) ;
$DOMAIN ;
        ID  /  CHAR ;
        ABS /  ABST ;
        AD  /  REAL ;
$RELATION ;
   COMP( DIV_ID/ID, DIV/ABS ) ;
        D0, NAT2( 0.0, 0.0 ) ;
$RELATION ;
   TERM( T_ID/ID, DIV_ID/ID, CHR/ID,
        XC/AD, YC/AD ) ;
        Q0, D0, TP_N, 0.0, 0.0 ;
        I0, D0, TP_N, -9.0, 2.0 ;
        I1, D0, TP_N, -9.0,-2.0 ;
$END ;
```

Fig.7 Definition of a 2-input NAND gate.

representation of the interface of the logic diagram. It is preceded and followed by calls to the translation function, i.e. .ORIGIN(X,Y), that sets the position of the diagram to be displayed.

```
    .ORIGIN( X,Y ) ;
    DISPLAY  CIRCLE ( RADIUS = '0.5',
      SELECT XC, YC
      FROM   TERM
      WHERE  DIV_ID = 'DO'
        AND  CHR = '***I*' ) ;
    DISPLAY  CROSS  ( RADIUS = '0.5',
      SELECT XC, YC
      FROM   TERM
      WHERE  DIV_ID = 'DO'
        AND  CHR = '***N*' ) ;
    DISPLAY  IMAGE  ( GF1 ) ;
    .ORIGIN( -X,-Y ) ;
```

The condition DIV_ID = 'DO' indicates that the displayed symbols are those of the interface description. The condition CHR = '***I*' specifies a partial match, i.e. the forth letter of the CHR column be 'I'. The graphics function DISPLAY IMAGE (GF1) draws the line segment named GF1, which is defined by a list of coordinates stored in the image file (Fig.3).

The following formulas display the interface of the components and connecting wires in the logic diagram. In other words, it draws the implementation of the logic diagram graphically on a display.

```
    DISPLAY   INTERFACE
    ( SELECT  DIV
      FROM    COMP
      WHERE   DIV_ID /= 'DO' ) ;
    DISPLAY   IMAGE
    ( SELECT  L_ID
      FROM    CONN ) ;
```

Since the interface description of each component is stored in the design hierarchy, the DISPLAY INTERFACE function is implemented by some sophisticated procedures, e.g. executing relational and graphics operations in lower levels.

3.3 Detecting Errors in Logic Diagrams

Finding errors in logic diagrams potentially offers time and economic advantages for circuit engineers. This subsection describes use of the relational operations in connection with error detection. Some of the potential errors that can be detected at a schematic design step are listed below (even though they may apply to other steps in VLSI design cycle [22,24]) :

(1) each input/output of a component must be connected,
(2) all inputs of a component must be connected,
(3) outputs of a component must be connected to compatible inputs,
(4) outputs of a component must not drive too many inputs.

3.3.1 Detecting Unconnected Ports

Unconnected ports are ports in a diagram that is neither origin nor destination of wires. All ports in the diagram are listed in the TERM relation. The origin and the destination of wires are stored in the CONN relation. Thus, to find unconnected ports, we get the following formula :

```
    TR <---  SELECT  T_ID, DIV_ID  FROM  TERM
             DIFF
             ( SELECT  S_ID, DIV_S  FROM  CONN
               UNION
               SELECT  D_ID, DIV_D  FROM  CONN );
```

The following formula retrieves the coordinates of the unconnected ports and display a predefined symbol 'X' at their position.

```
    DISPLAY   MARK
    ( SELECT  XC, YC
      FROM    TERM, TR
      WHERE   TERM.T_ID = TR.T_ID
        AND   TERM.DIV_ID = TR.DIV_ID ) ;
```

3.3.2 Detecting Components with Unconnected Inputs

Unconnected inputs on components are inputs that are not connected to any wire. Note, however, we must exclude input ports of the diagram under design. The following formula discovers a set of unconnected inputs together with identifiers of the related component.

```
    TR <---  SELECT  T_ID, DIV_ID
             FROM    TERM
             WHERE   DIV_ID /= 'DO'
               AND   T_ID = 'I*'
             DIFF
             SELECT  D_ID, DIV_ID
             FROM    CONN ;
```

To display the result graphically, we must retrieve the abstract instances related to the components.

```
    DISPLAY   INTERFACE
    ( SELECT  DIV
      FROM    COMP, TR
      WHERE   COMP.DIV_ID = TR.DIV_ID ) ;
```

3.3.3 Detecting Illegal Connection

There are many kinds of illegal connections in logic circuits [22,26]. Among them, we consider the following typical one. In the standard TTL, three output types can be included for organizing circuits, i.e. two-state, three-state and open-collector. The three-state outputs can interface directly with and drive data lines of bus-organized circuits. The open-collector outputs can be tied with other similar outputs to perform the wire-AND function. Thus, outputs of these two types may be connected to other outputs of an identical type, while two-state outputs must not be tied with other outputs of any type. The following expressions formulate the last part of the statement and display results on a display.

```
    TR <---  SELECT  S_ID, DIV_S, L_ID
             FROM    CONN
             GROUP BY  D_ID, DIV_ID
             HAVING  COUNT(*) > 1 ;
```

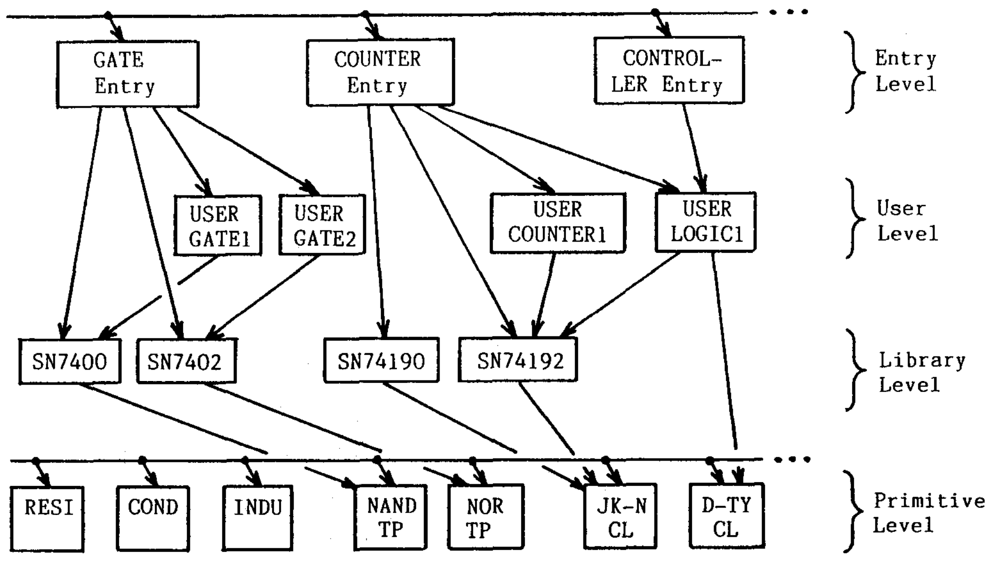


Fig.8 Overall structure of logic diagram database.

```
DISPLAY   IMAGE
( SELECT  L_ID
  FROM    TERM, TR
  WHERE   TERM.T_ID = TR.S_ID
    AND   TERM.DIV_ID = TR.DIV_S
    AND   TERM.CHR = 'TP' ) ;
```

The condition CHR = 'TP' selects tuples related to two-state outputs.

3.3.4 Checking Fan-out

Fan-out check is done to ensure that the outputs of components are not driving too many inputs. The check usually takes into account inputs with a value of other than one standard load. However, in order to make formulation simple, we assume that all inputs be with one load. The expressions below detect outputs driving more than N load and display a predefined symbol at their position.

```
TR <--- SELECT  S_ID, DIV_S
        FROM    CONN
        GROUP BY  S_ID, DIV_S
        HAVING  COUNT (*) > N ;
DISPLAY  MARK
( SELECT  XC, YC
  FROM    TERM, TR
  WHERE   TERM.T_ID = TR.S_ID
    AND   TERM.DIV_ID = TR.DIV_S ) ;
```

3.4 Organizing Structure of Logic Diagrams

Logic diagrams may be, and generally are, defined within other diagram definitions. The definitions may be nested to any depth. ADAM data model allows the engineers and CAD application programs to express diagram structures in a hierarchy. Some diagrams can be shared to create a directed acyclic graph rather than a tree.

Functions of digital circuits can be divided

```
PLIS( PI/ID, AIN/AID, FUNC/ID, N_GAT/R,
      N_IN/R, N_OUT/R, QTY/ID ).
SN00, SN00(X,Y),NAND, 4.0, 8.0, 4.0, TP
SN16, SN16(X,Y), INV, 6.0, 6.0, 6.0, OC
SN02, SN02(X,Y), NOR, 4.0, 8.0, 4.0, TP
SN08, SN08(X,Y), AND, 4.0, 8.0, 4.0, TP
SN09, SN09(X,Y), AND, 4.0, 8.0, 4.0, OC
SN10, SN10(X,Y),NAND, 3.0, 9.0, 3.0, TP
SN01, SN01(X,Y),NAND, 4.0, 8.0, 4.0, OC
SN33, SN33(X,Y), NOR, 4.0, 8.0, 4.0, OC
SN133, SN133(X,Y),NAND, 1.0, 13.0, 1.0, TP
```

Fig.9 An example relation in the GATE entry.

into about ten categories listed below according to descriptive information that features them.

(1) Drivers / Gates
(2) Decoders / Encoders
(3) Asynchronous counters
(4) Synchronous counters
(5) Registers / Latches
(6) Shift registers
(7) Arithmetic units
(8) Memories
(9) Controllers
(10) CPUs / Systems

Our logic diagram management system supports an entry for each category. The entries provide users with powerful tools for finding circuits for which he or she is looking. Fig.8 shows the overall structure of logic diagrams. It consists of four levels, i.e. entry, user, library and primitive levels. The entry level consists of a collection of relations that include both descriptive information (e.g. the number of gates contained, types of outputs, clock frequency) and uninstantiated abstract instance for accessing diagrams in the user or the library level. Fig.9 gives an example of the relations in the GATE entry. User-defined diagrams are stored in the user level. Since a user-defined diagram may incorporate other user-defined diagrams, this level may constitute subhierarchies of any height. Standard logic diagrams and primitives organize the library level and the primitive level respectively, which are usually created by database system engineers. The diagrams in the library level may consist of some primitives. Thus they may include their implementation description, whereas the diagrams in the primitive level do not.

4. EVE : Editor for Logic Diagrams

4.1 Objects and Actions in Editing Logic Diagrams

As mentioned earlier, engineers deal with design objects as a logical group that are usually represented by a collection of heterogeneous records. It is therefore feasible to design a user interface that allows users to access and manipulate these objects as a unit. This section describes an editor, termed EVE, that is tailored for logic diagram editing based on the ADAM database system.

The following are five intrinsic objects that construct the diagrams.

(1) component diagrams,
(2) wires,
(3) input / output ports,
(4) graphical representation of the objects (1), (2) and (3),
(5) text to name the objects (1), (2) and (3).

The following actions are also required to edit the diagrams.

(1) archive diagrams,
(2) retrieve archived diagrams,
(3) re-edit archived diagrams,
(4) verify diagrams,
(5) define, move, delete the objects,
(6) control display representations of the objects.

(2) is used to retrieve components from the database to edit a new diagram, whereas (3) is used to retrieve a diagram in order to revise it.

4.2 Command Menu

The editor EVE is designed and implemented for a storage tube display. Since this kind of display is not a very powerful interactive device, we must design a command menu in which users can explicitly specify each objects and actions. The resulting menu is shown in Fig.10. The menu displays plainly on the screen the full range of commands available to the user. It is generally necessary to use such a large menu in storage tube display user interface to avoid the need for frequent menu changes. Each time the user retrieves a component,defines a wire, etc., he gives the command by pointing a corresponding item in the menu. Errors simply abort the current command.

The commands preceded by '*CIRCUIT LIBRARY' are provided to archive, retrieve or re-edit diagrams. The menu items 'GATE', 'CONTROL', etc. are directly related to the corresponding entries in Fig.8. The commands preceded by '*VALIDITY' activate the diagram verification procedures discussed in Section 3.3. The commands preceded by '*BASIC

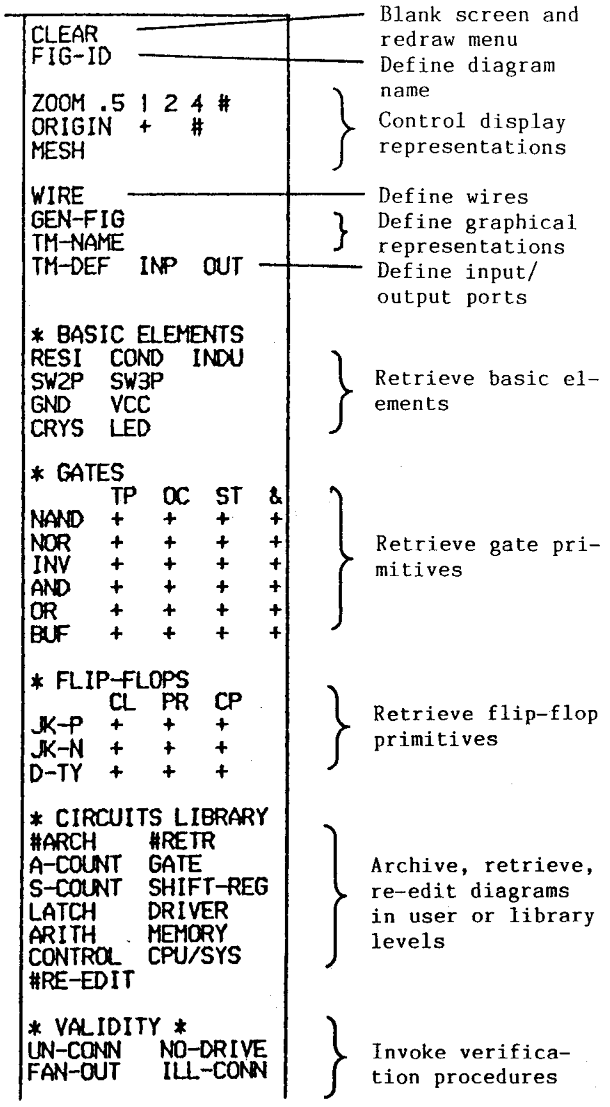


Fig.10 Command menu of EVE.

ELEMENTS', '*GATES' and '*FLIP-FLOPS' are used to retrieve primitives such as resistors, NAND gates and flip-flops, respectively. Each item of these commands is related to the entries in the lowest-level in Fig.8.

Other commands are as follows.

CLEAR.	Blanks the screen and redraws the menu.
FIG-ID.	The user types a name of a diagram followed by a RETURN.
ZOOM.	The scale of the drawing area is reset according to the selected scale.
ORIGIN.	The origin of the drawing area is reset according to the indicated position or specified coordinates.
WIRE.	Defines wires.
MESH.	The width of the mesh on the drawing area is reset according to the specified width.
GEN-FIG.	The user draws a graphical representation of the interface of the diagram under design.

TM-NAME. The user types a port name followed by a RETURN.

TM-DEF. Defines input/output ports of the diagram under design.

4.3 Operations and Performance

A typical model for an editing cycle is sketched in Fig.11. When EVE is initialized, the three relations for describing a logic diagram, i.e. COMP, TERM and CONN, contain no tuples. In editing the diagram, four objects are primarily manipulated, i.e. components, input/output pots, wires and graphical representations. Following editing, it is necessary to check whether the diagram involves errors. In our diagram management system, the final diagram can be registered in one or more categories.

4.3.1 Retrieving Primitives and Components

In order to retrieve a primitive, the user points to the corresponding item in the menu and then indicates the position at which the primitive is displayed. To retrieve a component, first, the user chooses a circuit category. Then the system responds by showing the column names related to the category in a blank skeleton table displayed at the bottom of the screen (Fig.12). Now he can express a query for retrieving the desired component in the same way as Query by Example [15,30]. Finally, he points to the target point. Retrieving primitives or components, which amounts to a direct counterpart of the concept "instantiation" [2], gets the system to merge the interface descriptions of the objects with the implementation descriptions of the diagram under design. Each retrieval takes approximately 250 msec of elapsed time on a MELCOM COSMO 900 II for the first time because it requires access to data on secondary storage, whereas the following retrieval takes approximately 60 msec.

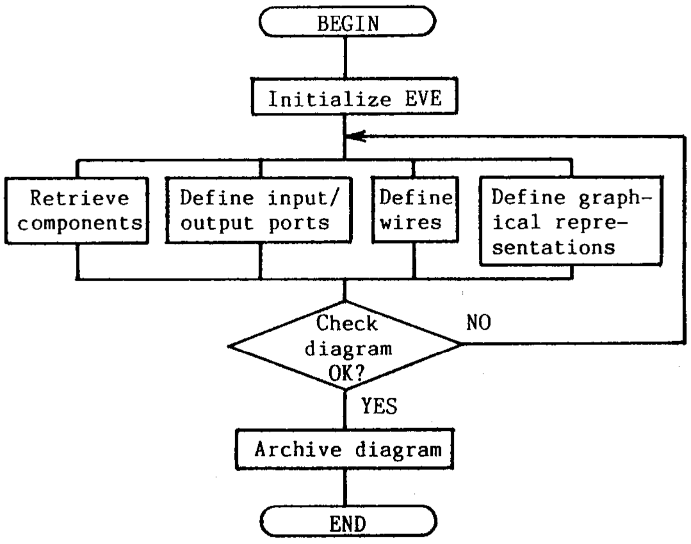
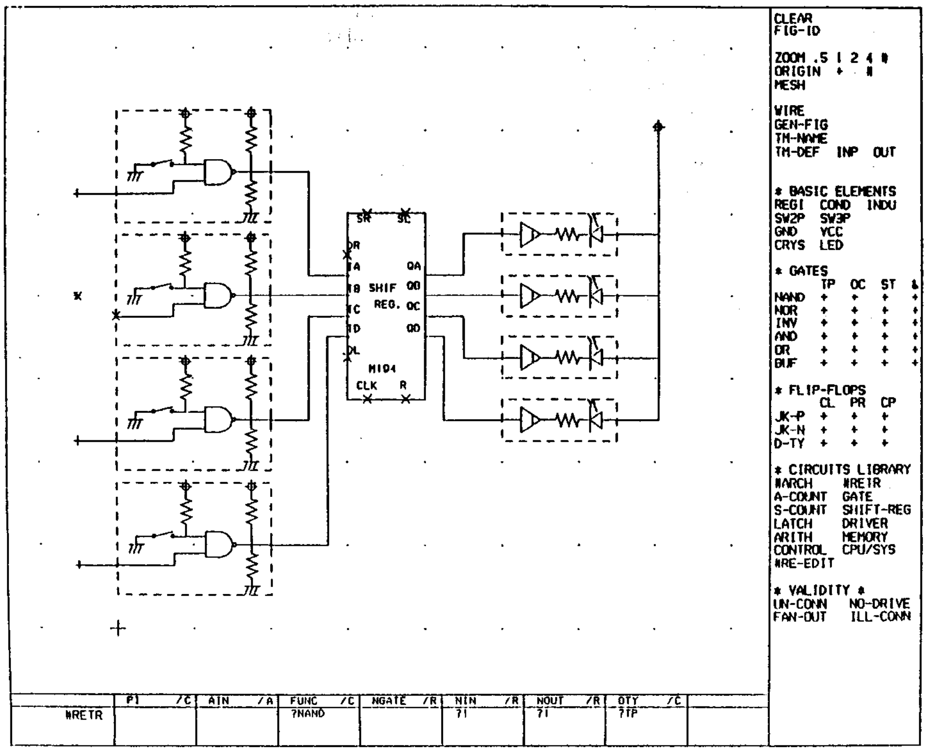


Fig.11 A typical editing cycle.



	PI /C	AIN /A	FUNC /C	NGATE /R	NIN /R	NOUT /R	OTY /C	
#RETR			?NAND		?1	?1	?TP	

Fig.12 Editing a diagram by EVE. 'X's indicate unconnected ports.

4.3.2 Defining Input/Output Ports

To define an input/output port, the user chooses the TM-DEF command and then moves the cursor to the desired position. This operation gets the system to insert a single tuple to the TERM relation. Since the TERM relation is kept in main memory, defining a port is performed quite efficiently. It only takes approximately 30 msec.

4.3.3 Defining Wires

To define a wire, the user selects the WIRE command and then indicates a sequence of points that define the wire. First, the system checks whether ports to which the wire is connected are defined in the TERM relation by a simple selection operation. If they are defined, the system inserts a single tuple to the CONN relation and stores coordinates of the points in the image file. Since the relations and the image file are kept in main memory, defining a wire typically takes 80 msec.

4.3.4 Defining Graphical Representation

To define a graphical representation of the interface of the diagram under design, the user chooses the GEN-FIG command and then indicates a sequence of points that define it. The system stores coordinates of the points in the image file. Defining a graphical representation typically takes 100 msec.

4.3.5 Verifying Diagrams

To detect errors in the diagram, the user chooses one of the commands preceded by '*VALIDITY' in the menu. Since all the relations needed to perform the verification are kept in main memory, verifying a diagram is executed efficiently. However, the CPU time needed is largely depend on the number of tuples in the relations. It typically takes 1500 msec to detect illegal objects and display them.

4.3.6 Archiving Diagrams

To archive the diagram, the user points the item '#ARCH' and chooses a circuit category. Then the system responds by showing the column names related to the category in the blank skeleton table on the screen. Now he can type the descriptive information about the diagram in the appropriate place in the table. Finally, he types a special character, say blank. Then the system updates the management information about the structure of the design objects and then writes the relations, the picture file and the image file to secondary storage. Archiving a diagram takes approximately 3000 msec.

5. Conclusions

The use of a database system has the potential to free its users from routine data management tasks. For example, database systems usually provide features for data sharing, concurrency control and automatic crash recovery. With relational database systems its users can define different views of the same data and also dynamically alter them, allowing many programs with different data requirements to use one common database. However, no existing relational system supports the many facilities needed to support design activities. They include :
(1) an explicit representation of the design hierarchy,
(2) support for various data types,
(3) support for version control,
(4) efficient access to design data.

In this paper, we have described a logic diagram management system built on top of an extended relational database called ADAM. ADAM is a consistent database management system which supports all the facilities mentioned above. By using hierarchical levels of abstraction, the design data can be organized into meaningful groupings for application programs. Data that are likely to be used together will be stored together and can be retrieved with only a few operations. In our system, a unit of access to secondary storage is the meaningful group of the design data. Extracted data are then processed primarily in main memory. It appears that these implementation strategies lead to the design system with improved performance over systems using traditional relational database management systems.

The future research and development works are summarized as follows :
(1) extending the data model to manage behavioral and functional information,
(2) developing a comprehensive version control mechanism,
(3) implementing a powerful user interface including browsing capabilities.

Acknowledgments

We gratefully acknowledge the helpful comments of Dr.M.Sudo, the manager of Software Department at Information Systems and Electronics Development Lab. of Mitsubishi Electric Corporation.

References

[1] Barabino, G.P., Barabino, G.S., Bisio,G. and Marchesi,M. : A Model for Improving Data Access and Management in an Integrated CAD Environment, Proc. 22nd ACM/IEEE Design Automation Conference (June 1985), 577-583.

[2] Batory,D.S. and Kim,W : Modeling Concepts for VLSI CAD Objects, ACM Trans. Database Syst. 10,3 (Sept 1985), 322-346.

[3] Beetem,A, Milton,J. and Wiederhold,G. Performance of Database Management Systems in VLSI Design, IEEE Database Engineering, 5.2 (June 1982).

[4] Chen,P.P.S.: The Entity-Relationship Model - Toward a Unified View of Data, ACM Trans. Database Syst. 1,1 (March 1976), 9-36.

[5] Chu,K-C, et al.: VDD - A VLSI Design Database System, Proc. ACM SIGMOD Conference on Engineering Design Applications, San Jose, CA (May 1983).

[6] Codd,E.F.: Extending the Database Relational Model to Capture More Meanings, ACM Trans. Database Syst. 4,4 (Dec. 1979), 397-434.

[7] Crowley,J. : Automation Smoothes CAD Process, Electronics (Dec.3,1984), 65-69.

[8] Datam,P., Lum,V. and Werner,H.D. : Integration of Time Versions into a Relational Database System, Proc. 10th Intl. Conf. on Very Large Databases (1984), 509-522.

[9] Date,C.J. : An Introduction to Database Systems, Vol.1, 3rd ed. Addison-Wesley, Reading, Mass, 1981.

[10] Elmasri,R., Weeldreyer,J. and Hevner,A. : The Category Concept : An Extension to the Entity-Relationship Model, Data & Knowledge Engineering, North-Holland, Vol.1 (1985), 75-116.

[11] Guttman, A. and Stonebraker, M. : Using a Relational Database Management System for Computer-Aided Design Data, IEEE Database Engineering 5,2 (June 1984), 56-60.

[12] Hartzband,D.J. and Maryanski,F.J. : Enhancing Knowledge Representation in Engineering Databases, IEEE Computer (Sept. 1985), 39-48.

[13] Haskin,R. and Lorie,R. : On Extending the Functions of a Relational Database System, Proc. ACM SIGMOD 1982, ACM, New York, 207-212.

[14] Haynie, M. N. : The Relational Data Model for Design Automation, Proc. 20th ACM/IEEE Design Automation Conference (1983), 599-609.

[15] Heiler,S. and Rosenthal,A. : G-WHIZ, a Visual Interface for the Functional Model with Recursion, Proc. 11th Intl. Conf. on Very Large Databases (1985), 209-218.

[16] Katz,R.H. : A Database Approach for Managing VLSI Design Data, Proc. 19th ACM/IEEE Design Automation Conference (June 1982), 274-282.

[17] Katz,R.H. : Information Management for Engineering Design, (Surveys in Computer Science) Springer-Verlag, 1985.

[18] Keller,K.H., Newton,A.R. and Ellis,S. : A Symbolic Design System for Integrated Circuits, Proc. 19th ACM/IEEE Design Automation Conference (1982), 460-466.

[19] Kim.W and Batory,D.S. : A Model and Storage Technique for Versions of VLSI CAD Objects, Proc. of Intl. Conference on Foundation of Data Organization (May 1985), 329-334.

[20] Lorie,R, Kim,W, Mcnabb,D., et al. : Supporting Complex Objects in a Relational System for Engineering Databases, In Query Processing in Databse Systems, W. Kim, D.Reiner and D.Batory, Eds., Springer Verlag (1985), 145-155.

[21] Newman,W.M. and Sproull,R.F. : Principles of Interactive Computer Graphics, 2nd ed., McGraw-Hill, 1979.

[22] Ravid,E. and Nerat,T. : Problems in Logic-Array Design on Engineering Workstations, VLSI Design (Sept. 1984), 46-51.

[23] Roberts,K.A., Baker,T.E. and Jerome, D.H. : A Vartically Organized Computer-Aided Design Data Base, Proc. 18th ACM/IEEE Design Automation Conference (1981), 595-602.

[24] Robson,G. : Benchmarking the Workstations, VLSI Design (March/April 1983), 58-61.

[25] Smith,D.C. and Wagner,B.S. : A Low Cost, Transportable, Data Management System for LSI/VLSI Design, Proc. 19th ACM/IEEE Design Automation Conference (1982), 283-290.

[26] The Bipolar Digital Integrated Circuits Data Book, Texas Instruments, 1985.

[27] Tool Set Links All Stages of Full-Custom IC Design, Electronics (July 1,1985), 60-62.

[28] Udagawa,Y. and Mizoguchi,T. : An Advanced Database System ADAM --- Towards Integrated Management of Engineering Data, Proc. 1st IEEE Intl. Conf. on Data Engineering (1984), 3-11.

[29] Wilkins, M. W., Berlin, R., Payne, T. and Wiederhold,G : Relational and Entity-Relationship Model Databases and Specialized Design Files in VLSI Design, Proc. 22nd ACM/IEEE Design Automation Conference (June 1985), 410-416.

[30] Zloof,M. : Query by Example - A Data Base Language, IBM Sys. J. 16,4 (1977).