ISI Research Report
ISI/RR-93-386
April, 1993

A Component Library Management System and Browser

John Granacki, Zia Iqbal and Tauseef Kazi

ISI/RR-93-386

April, 1993

University of Southern California
Information Science Institute
4676 Admiralty Way, Marina del Rey, CA 90292

A Component Library Management System

John Granacki, Zia Iqbal and Tauseef Kazi

USC/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292
310.822.1511

Abstract

This report describes a component library management system (LMS) that is implemented with the OCT database. The LMS consists of a user-extensible component library and, a browser that enables the user to search through component lists with the help of a graphical user interface. The LMS provides a query mechanism that allows the user to search for components by attributes and quickly identify parts that meet a design requirement. The LMS also provides a link between the browser and FAST (an ARPA-sponsored electronic parts broker) procurement data, so that designers can automatically generate electronic orders for parts to FAST.Although commercial library browsing tools exist (such as Cahners Computer-Aided Product Selection CAPS), these tools cannot process the component data or generate procurement requests. In this report we discuss the trade-offs between relational database systems and object oriented database system for implementing the LMS and describe the use of OCT in our implementation. We also present a flexible and extensible browser datastructure that has been created for rapid on-line access of the OCT component library information. Finally, we identify the limitations of the current implementation and propose some possible solutions and future work.

1.0 Introduction

This report describes a component library and associated library management tools.This Library Management System (LMS) is being developed primarily to aid the component search, selection and procurement tasks during the physical design [1] of PCBs (printed circuit boards) and MCMs (multichip modules). Key features of this system are a flexible user interface that supports searching component libraries using different attributes and, tools for component data management, management of design data and the ordering of parts through the ARPA-sponsored FAST brokerage (FAST is a service accessible by electronic mail for the procurement of standard electronic components). These tools will reduce the designers dependence on the current paper-based manufacturers' data books and accelerate the trend towards electronic data books. These tools are being developed using the OCT database tools and the other tools developed as part of the ARPA-sponsored UCB SIERA design system and Lager IV [5].

The proposed LMS is compatible with a broad range of commercial and university-developed tools and supports a neutral design style, that is, either text or graphical based system descriptions.

In this section we discuss the basic motivation for the component library and library management system and present a brief overview of the LMS. In section 2, the selection of the database for the component library is discussed and the choice of OCT is justified. Section 3 presents the OCT policy we have developed for the component library to support various library management functions. In section 4 the details of the implementation of the LMS are described and in particular we present a data structure that has been developed for efficient searches of the component database.

1.1 Requirements of the component library

Most CAE/CAD tools are delivered with some form of vendor-supplied library. In addition, there are vendors like Logic Automation that supply tool-specific libraries. The problem is that these libraries do not necessarily contain all of the information a designer usually needs to complete the physical design, the procurement of parts, fabrication and assembly of an ASEM (Application Specific Electronic Module). Also these libraries are often based on incompatible representations which makes it difficult to extend and/or update the libraries while maintaining data consistency.

There is also a cost issue since the commercial libraries are expensive and multiple versions must be obtained to support tools from different vendors. The specific tool vendors frequently do not make it easy to use their libraries with other vendor's tools; therefore; a user who has augmented a particular vendor's library is effectively locked into using a particular vendor's tool set. To solve these library problems, some large corporations have invested heavily in the development of their own internal proprietary libraries which are tied into their enterprise-wide management information systems. For cost reasons the proprietary library solution is not an alternative for small companies and academic researchers.

To solve the above problems, the component library in the proposed LMS has been defined to allow storage of a large amount of generic part and package information and special tools are being created to allow users to add data and extend the library. It is expected that researchers will share information thereby reducing the amount of data capture that any individual will have to do. In addition to data on standard packaged components, we plan on including data on bare die that will be needed in the design, manufacturing and assembly of MCMs.

1.2 Requirements of library management tools

Finding the "best" parts for a design is a tedious and time-consuming task that is often "hit or miss" and does not produce a part number that can actually be used to order the desired component. Currently, designers using various design capture tools, (both commercial and university-developed), do not have any automated tools to aid in searching for components. An example of a commercial library tool is CAPS (Cahners Computer-Aided Product selec-

tion). This is, a system for automated search, and simply replaces the collection of data books with a collection of CD ROMs and bitmap graphic displays. Since tools like CAPS do not produce or allow access to computer-readable or computer-processable data, the engineer must frequently transcribe this information to other tools manually, which is error prone and, for example, results in ordering on incorrect parts. The problems with incorrect part numbers often are not detected until the designer actually places the order for the parts, thus lengthening the development cycle. By creating a database for storage of the component library, the proposed LMS allows the computer to directly process component data for design entry and parts ordering, thereby avoiding manual entry. Furthermore, the definition of the component database allows the designer to drive the search by using different attributes as illustrated below in section 1.3. This feature can be used to quickly compare alternative parts for the same requirement from different manufacturers or in different technologies.

A second problem in component selection is that designers often have to redesign some or all of the circuitry because parts are unavailable or their availability impacts the development process. Since the proposed LMS provides a link between the component library and FAST procurement data, the designers can automatically generate electronic RFQs to FAST early in the design cycle and obtain accurate information about the availability of parts. This will help avoid costly redesign and decrease the time for the completion and fabrication of designs.

1.3 Overview of the library management system

A prototype component library and a browser tool have been implemented (Figure 1). The program is able to read in data from the set of existing libraries (Figure 1). Upon the choice of a particular library, a multiple column display is presented (Figure 2). The display consists of lists of components and their attributes. The user interface supports selections using both the keyboard and the mouse. The interface allows the browsing of the complete list of components in the library. The user can restrict the list of components at any time by selecting an entry in any column. For example, by selecting CMOS in the technology attribute column the user restricts the list of components to only those belonging to the class CMOS. The restriction by multiple attributes (for example, “TECH” {technology} and “MFR” {manufacturer}) further reduces the list to a set of components that have the attribute selected in each column (see Figure 2). Selecting an attribute highlights the particular selection and removes the other attributes from the list, components can be unselected by clicking again on the highlighted attribute, this rebuilds the lists according to the remaining selections, thus the user can move forward and backward in the query process till the desired component has been found. The list of components can be restricted by any attribute and by as many attributes as desired until a single component is selected. If a particular component is selected, its relevant attribute information will be displayed.

Once a component has been selected, procurement can be initiated with a single keystroke. The selected components will be automatically ordered from the FAST brokerage. The program generates an email message for ordering the selected components.

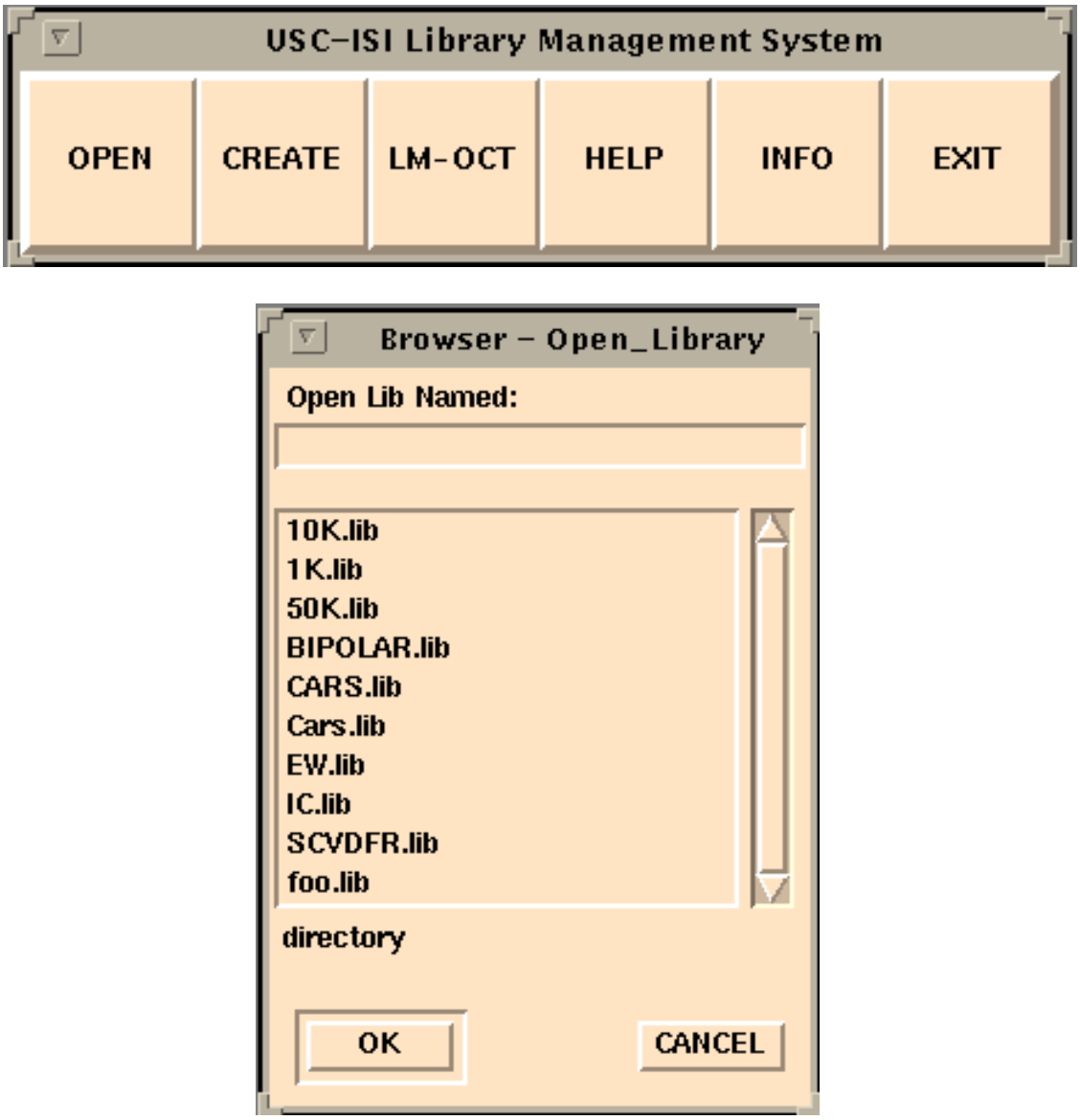


FIGURE 1. Library selection with the LMS

2.0 Selection of a data base system

To develop the component library and library management system, the first task is to select an appropriate database system. As discussed below we selected the OCT database and developed the library management system on top of OCT. To achieve this we defined a new OCT policy for electronic components. The OCT policy is described in section 3 and the implementation of the library management tools are described in section 4. In this section we discuss the criteria for selection of the OCT database.

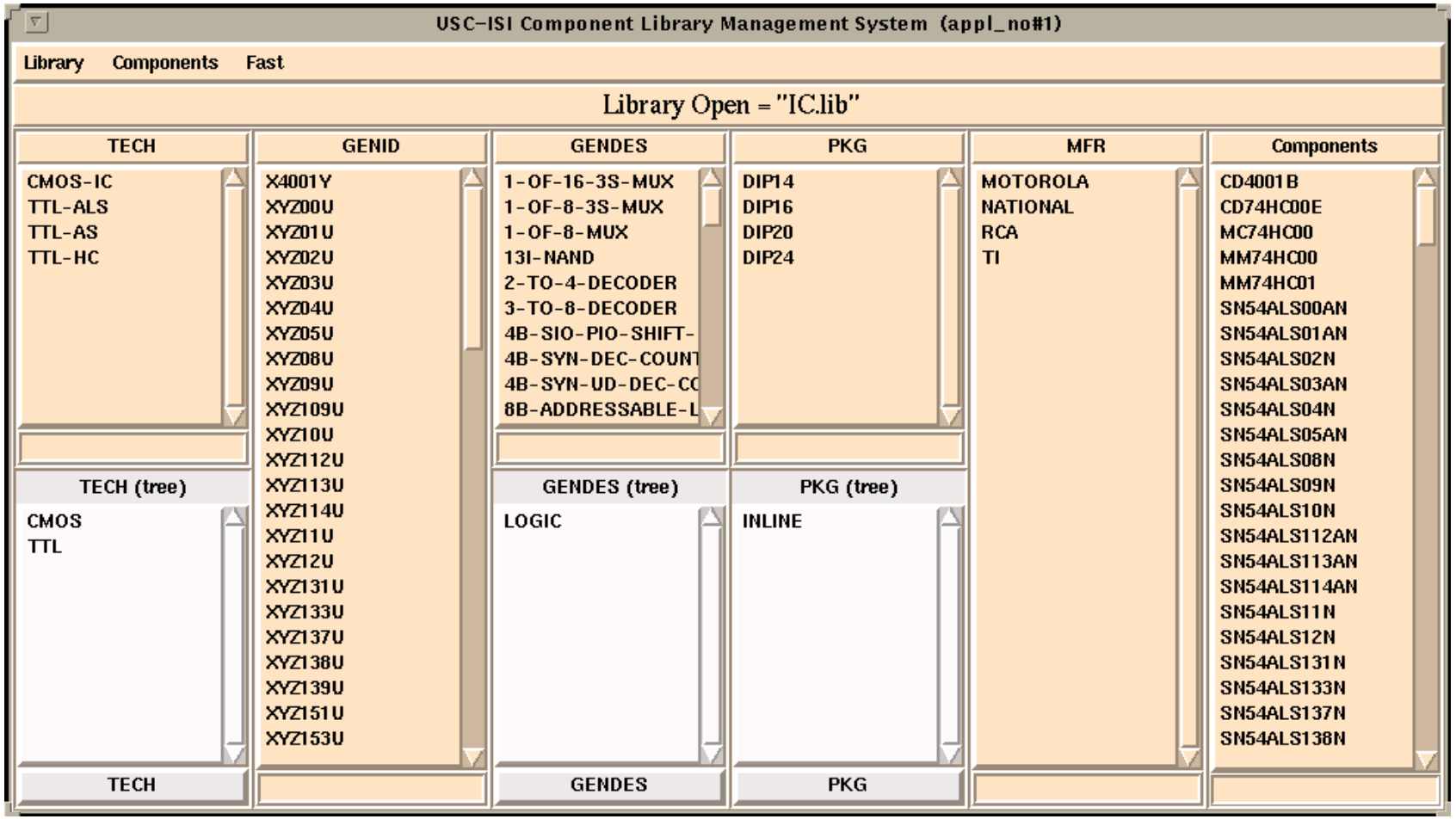


FIGURE 2. The browser interface as it appears without any selections.

2.1 Comparison of basic data models

Existing data management systems support two basic data models, those based on the relational data model (RDBMS) and those based on the object-oriented data model (OODBMS).

The relational data model provides efficient mechanisms for the storage and manipulation of non-hierarchical data (flat data) that can be tabulated, accessed and modified using high-level query languages. However the set of structures, operations and constraints in relational data is limited and fixed. Consequently, any modification or query operation required by the applications has to be mapped onto this limited set. As the applications become more complex, this mapping becomes less intuitive and harder to comprehend. In contrast, the object-oriented data model provides a natural mapping of concepts to data objects, it is well suited for CAD applications because CAD databases must deal with the complex relationships between multiple levels of abstraction.

The library application like most CAD applications is data intensive and has complex data associations (links), attempts to model these complex relationships with the relational data model result in complex relational functions that incur considerable processing overhead and duplication of data. The need to build these complex relationships from the select and join primitives seriously degrade performance.Examples of this processing overhead have been observed in the current implementation of MICON, RACAL REDAC, Visuala [4] and the Eve DBMS for USC's Advanced Design AutoMation System [6]. The Eve database is built on top of a commercial relational DBMS. Although Eve successfully models the data representations for an object-based model, but the performance degradation manifests itself in slow response times. The developers of Eve plan to reimplement Eve using an OODBMS to avoid this problem. Hence the OODBMs' ability to model complex relationships is better suited to this type of application.

2.2 Tables, query mechanisms and performance.

An advantage of the relational database technology is a well developed and standardized query language. However, the expressive power of the query languages comes with a substantial performance overhead.

The relational model stores data in the form of tables, and the relationship between two tables is established in terms of a common key (unique identifier) between the data stored in the two tables. Hence in order to retrieve information, which pertains to a particular entry in one table, from another table, searching has to be performed on the other table(s) based on the common keys and then a join operation is performed to obtain the required information. These operations lead to performance degradation, especially if the number of tables involved in the search is large. This problem is exacerbated by the need of CAD applications to use extensive join and restrict operations.

On the other hand, the approach adopted by object oriented data management systems allows the storage of the relationships between objects in the form of pointers and hence

provide a more practical and faster mechanism for data retrieval for these kind of applications. The concept of object identifiers simplifies the access mechanisms of objects and provides direct links to the objects through interpolated references (pointer chasing).

2.3 Schema development and maintainability.

The generalization and inheritance properties of the ODBMS's allow the development of a smaller and better structured schema since the common attributes can be factored out (generalization). Also modifications in schema are easier to implement in OODBMS' as compared to the RDBMS'.

2.4 Criteria for selecting OCT

For reasons given above we decided to select an OODMS to implement the library management system. OCT is a mature object-oriented database product, with available technical and software support. It is used by other CAD systems like SIERA, Ptolemy, the Lager toolset (particularly DMoct) and VANDA that are distributed and maintained by Mississippi State University under ARPA-sponsorship. Interfaces exist between OCT and several commercial tools and databases like Viewlogic's Workview System, Racal Redac's Visuala system and Harris' Finesse Layout System. Also, C-base system developed to support design for testability by Professor Breuer at USC has an interface to Lager/OCT for chip layout.

In designing a library management system with OCT the first task is to define a policy for storing the component information in OCT - this requires defining an OCT view as discussed below.

3.0 OCT policy used to implement library management system

Each library is defined as a cell as shown in Figure 3. Examples are a library (cell) of integrated circuit devices, passive components etc. Each cell consists of views which in turn contain facets. Each "view" contains the components of a particular manufacturer. So if our part list for a particular library (cell) contains components by ten manufacturers then the cell "library" would contain ten views. Each view has a contents facet contains all information pertaining to a manufacturers parts as described below.

The bag "comps" contains a list of all the components contained in the library by that particular manufacturer(view).

The OCT-bag "classes" contains a list of different attributes (OCT-bags) which are used to drive the search process (for example the attributes TECHNOLOGY, GENERIC ID, GENERIC DESCRIPTION, and PACKAGE in the IC-Component library). Each of these OCT-bags in turn contain properties that are the values of these attributes, for example the "tech" OCT-bag would contain OCT-properties like "TTL", "ECL", "CMOS" etc. These properties contain the names of their ancestors (OCT-bags) (for example CMOS has a field

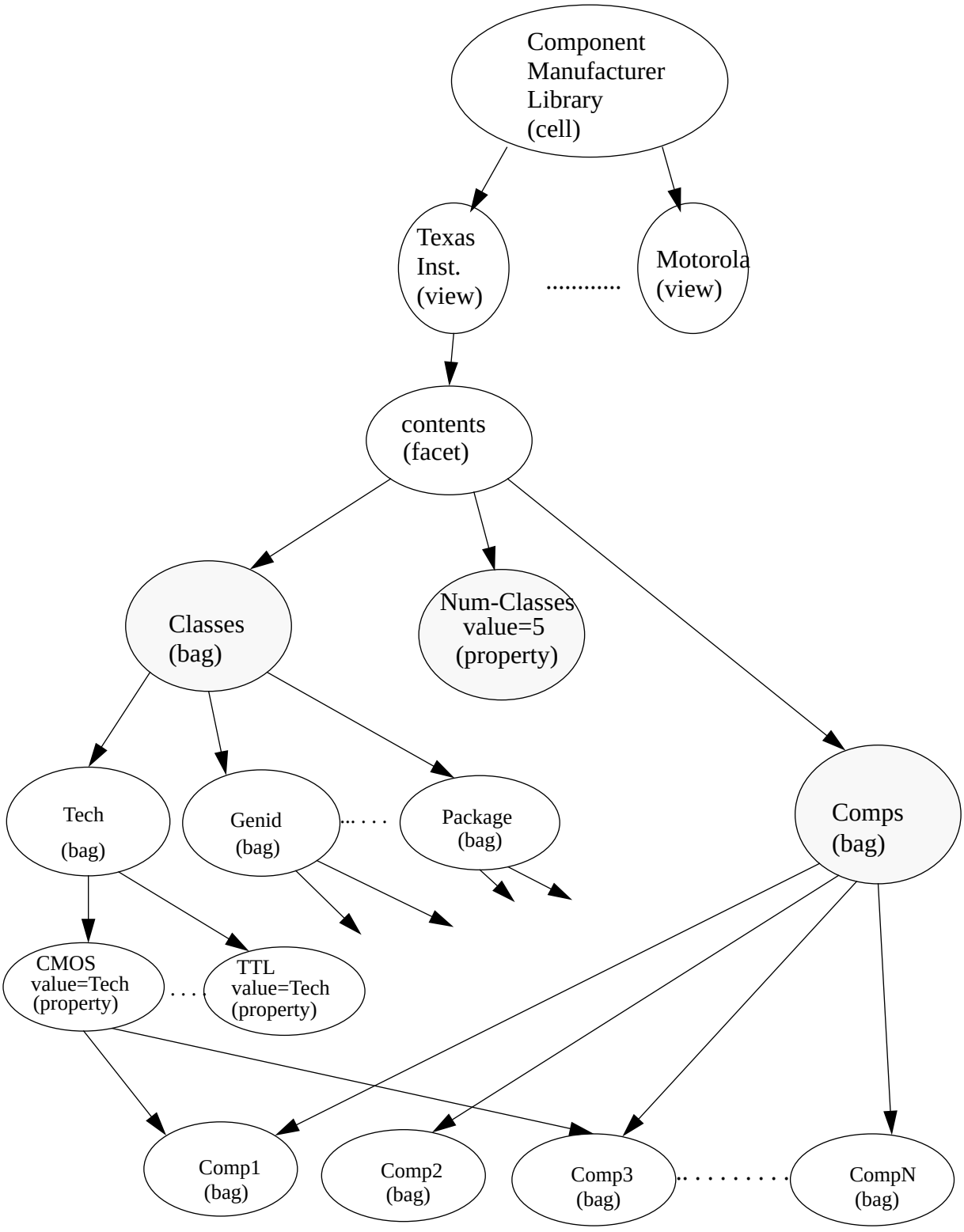


FIGURE 3. OCT policy of the Library Management System (LMS) for the Component Library.

containing TECH). This additional feature was incorporated so as to save on the searching process when setting up the structures for the user interface and the query program. This replication of ancestral information incurs no additional overhead in storage or memory space as the minimum size allocated for an OCT-bag or OCT-property is roughly the same, and typically there would be less than a hundred of these properties for a library of more than fifty thousand components. The property “num-classes” stores an integer count of the number of bags attached to “classes”. This allows each library to have a variable number of attributes.

Using the above policy, starting from the facet, the components can be reached either from the OCT-bag “comps” via the OCT-bag “classes” and OCT-properties “CMOS”, “TTL” etc. The whole list of the components can be traversed by opening the facets (containing components by manufacturers) for each manufacturer (either simultaneously or one by one) and processing each manufacturer’s lists of components.

3.1 Trade-offs between different OCT policies

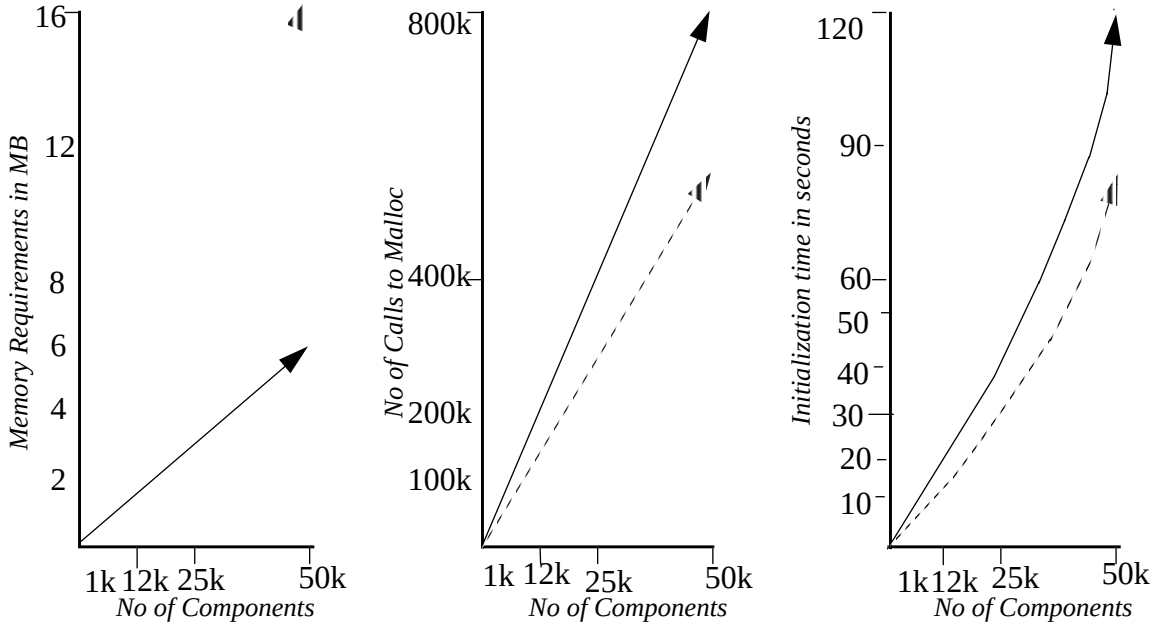
The basic choices that we had in the adoption of the above policy was at the cell level. We had to decide whether to organize data under one view containing all the information. We decided to break up the data in views based on a chosen attribute MFR (manufacturers). The selection was supported by the rationale that the separation by manufacturers provides a natural partition, each manufacturer generally supplies data about their parts and results in a more or less an balanced storage for the bulk of data. The decision to break up data into different (storage) views also provided the flexibility to work with all data in memory simultaneously or to work with selected data in memory at a time.

4.0 Details of implementation

This section describes the implementation of a browser tool that services queries from the user by accessing component information from the OCT database using the policy defined above.

4.1 Transaction processing vs. bulk processing

We decided to use a query mechanism based on memory resident data that is transferred from the OCT database when the browser is initialized. We first developed a prototype based on a hybrid scheme. The hybrid scheme utilizes the OCT data mechanisms for query purposes and has minimal amount of data in its private data structure therefore the speed of our query mechanisms was determined by the built-in routines of OCT. We examined the trade-off between the time required for initialization, individual queries and memory usage of OCT (Figure 4). The decision to use special data structures for query was driven by the requirement for fast response times and a desire to use the existing OCT system routines rather than modify them.



LEGEND:
----- OCT Implementation with selective facets open at a time.
----- OCT implementation with All Facets open simultaneously

FIGURE 4. The figure shows the trade-offs between different implementations, Plots show the effect of increase in the number of components upon performance. The particular results are based on implementations using the OCT Data Management system.

Plot A shows the effect of increase in the number of components in a library on the memory requirements of the system. Clearly slope for the implementation with all facets open simultaneously has a much higher gradient for the increase in memory requirements with the increase in the number of components. Hence inspite of speed limitations the selective facet approach has been adopted.

Plot B shows the effect of increase in the number of components over the number of calls to malloc function which effects the performance when such high number of calls are involved, but the difference in gradients of the slopes of the two implementations favor the selective facet scheme when the fig A and B are considered.

Plot C shows the initialization time of the library management system versus the number of components. Main reason behind the high initialization times is the sorting scheme and the complexity of the data structure involved in the process. The data has to be first read into the OCT Data structures and then transferred into the library manager's data structure, see Section 4.6 for details.

Figure 4 indicates that the implementation, with all data in memory simultaneously, increases the memory requirements of the system considerably, whereas the implementation with selective data in memory pays the penalty in the form of performance (time). Due to these observations we decide to read in the required data into our own data structures for performance purposes, and thus realize the advantage of enhanced performance and reasonable memory requirements. We found that a 50,000 component library implemented with

all data simultaneously in memory required 16 Mb of memory by the OCT structure itself. The other accounting information, required for storing and displaying the query results, increased the requirements by an additional 4-5 MB. The selective facet open scheme seriously degraded the performance and hence was not considered. In view of the above it was decided to import the required information into our own data structure and this reduced the memory requirement to a mere 6.5 MB for a library of comparable size. It was found that the size of the memory grows linearly with the increase in the number of components. The main reason for these saving in memory requirements lies in the fact that custom structures can dictate the size according to the requirements whereas OCT's generic structures always allocate the size of the largest possible structure and hence use more space than is actually required.

The design of the data structure for on-line queries of the component database is discussed next.

4.2 Trade-offs between different datastructure types

The various choices in terms of datastructures for the development of the query mechanism for the browser are b-trees, linked-lists and arrays. While an array-based implementation offers quick traversal time and better initial data structure setup time, the linked-lists are faster in terms of sorting while still offering reasonable times for traversal and setup. On the other hand, b-trees have the well-known property that insertion sort can be implemented with an average case complexity of n(log n). Searching techniques are faster with b-trees but as the trees grow in size, balancing mechanisms have to be employed to maintain the performance for searching functions. Unfortunately these balancing techniques become more and more complicated with the increase in size thus limiting the performance. Since this application involves large amounts of data arranged in small groups, b-trees are not the ideal candidates.

The nature of queries in the browser application is such that the restrict and intersection operations are applied rather frequently, which means that the updates on a conventional structure can take rather long times. The first prototype developed was based on arrays but we found that in order to implement efficient search mechanisms we needed to add pointer fields to the structure and thus we changed the implementation to a linked-list based structure containing a number of support structures overlaid on the basic linked-list to aid the search mechanisms. The use of b-trees for efficient search mechanisms over presorted data was rejected because our requirements primarily involve restrict operations over multiple attribute of data. In other words, a typical selection query for a component involves the lookup of the component in at least six lists and each query typically involves thousands of components. This was the primary reason for selecting this hybrid linked-list representation. The term hybrid here refers to the structure containing more then one primitive datastructure for the organization of data, that is, the combination of a linked list with an array structure as described below.

4.3 Proposed datastructure

The basic data structure consists of an arrangement of components in a linked-list structure (Figure 5). This datastructure is composed of a combination of lists and arrays for the organization of various types of data and also contains indices to enable quick access. Associated with each component are two arrays of pointers, one pointing to the different attributes of the component (example its MFR, TECH, PKG etc.) and the other pointing to the next component (in the list) belonging to that attribute (example next component in the list belonging to the same MFR, TECH etc.). The attributes (MFR, TECH etc.) are themselves arranged as an array and contain pointers to the components (stored as linked-lists). Hence there is a two way link between the components and their attributes which provides easy mechanisms for navigation through the data, based on a particular attribute, or via the component list itself while having access to the different attributes of the component at the same time.

The main reference to the data structure is the structure named “lib,” which contains pointers to the various sub structures containing different data (refer Appendix A for detailed structure). The various fields in the “lib” structure are now described in detail.

The field “name” contains the name of the library which is read in from the cell containing the information about the library. The next field “class” is a pointer to the structure-array “lib_class” containing the names of attributes (MFR, TECH etc.) and the list of pointers to components belonging to that attribute. These attributes appear as headings or labels on columns in the user interface display. The field “comps” is a pointer to a linked-list representation of components sorted on their names. The field “selected_comps” is a linked-list of components that are selected by a particular query at a particular instance and changes with different queries.The entries in this list appear in the user interface under the “Comps Selected” column. This list is dynamically built (for display purposes) as the queries are changed by unlinking and relinking pointers. The field “num_classes” stories a count of the number of classes in the library and the field “num_selected” stores a count of the number of components selected by the present query. The field “modified” is used as a flag for determining if the library has been modified since its last opening, if set then the modified data has to be written back otherwise the program exits without any writeback. The “indexed_comps” contains a pointer to a linked-list structure index_type_type which contains indices over the component list. “error_str” stores an error message as and when it occurs. The field “comp_input” is reserved for adding components to the library. The pointer “selected_names” is a pointer to an array of pointers which reference the names of the currently selected components. This field is exclusively for use by the user interface.

Each element of the structure “lib_class” contains a field “name” which stores the name of the attribute (appearing as a column in the GUI), “num_selected” which stores a count of the number of components selected of that particular attribute, and “list” which points to a linked-list of structure “class_type” which contains the members of the particular attribute present in the library.The members of this list appear as column entries in the GUI. The field “input” is used for addition of a new member in the library. The field “type” refers to the taxonomy of the class, it can be either hierarchical (taxonomical) or simple, if the type is

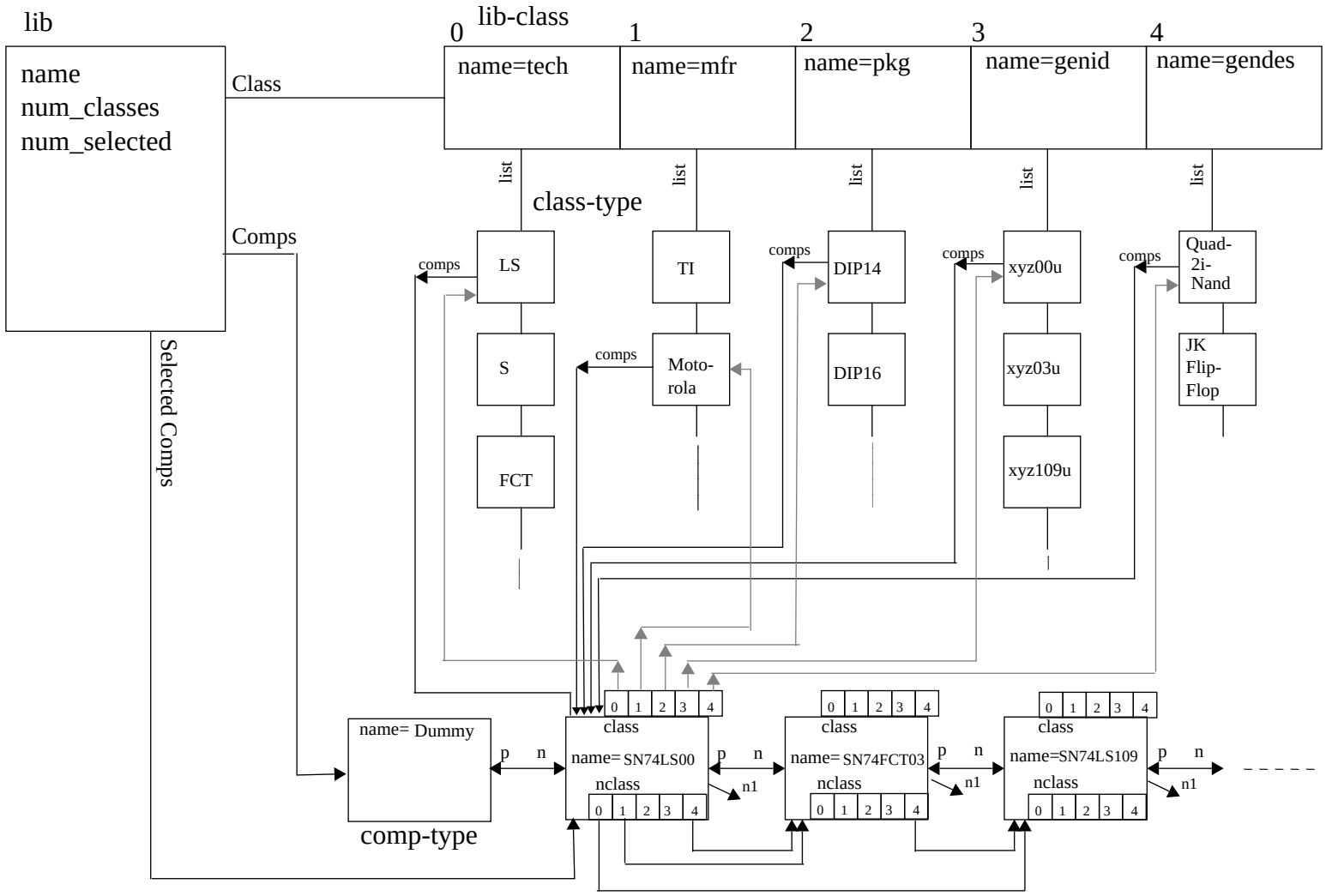


FIGURE 5. Proposed datastructure for browser for effcieint search of component by attributes

simple it is devoid of any hierarchy and just a plain list, but if the type is taxonomical then it contains a hierarchy of classes and the user can browse through the hierarchy to select a particular attribute to further select from within its subclasses.

To better understand the difference between the hierarchical and simple taxonomy consider the following example. Let the column “GENDES” (generic description) contain description of memory parts available in the library. In the case of a hierarchical taxonomy the list would contain the entries “Memory”, “Multiplexer” and “Counter” and the different types of memory (example the 32k-8bit-Static, 64k-16bit-Dynamic etc.) can be displayed by selecting Memory from the column. On the other hand if the view is simple then the column “GENDES” would contain the 32k-dram, 64k-sram etc. together with entries of other types like 4-1 multiplexer, 8-1 multiplexer, 8 bit-counter etc. Again the field “selected_names” is for the use of the graphical interface, it is a pointer to an array of pointers referencing the names of currently selected attributes for each column, note that there is a separate array for each column. This selected name array may have just one entry if a particular attribute has been selected or it may have multiple entries depending on the status of the query. For example the component list consists of entries comp1,comp1,comp3,comp4 etc. now suppose under column “GENDES” comp1 has attribute 32k-8bit ram, and comp2 and comp3 have atributes32k-8bit ram, and 4-1mux, now the selection of attribute 32k-8bit ram restricts the list of components to comp1 andcomp2, whereas the selection of the component comp2 would restrict the column “GENDES” to the entry 32k-8bit ram. Upon deselection of comp2 the component list and the attribute lists are restored.

The structure “class_type” contains “name” which has the name of that particular member of the particular column(attribute), “type” which points to the name of the column, “comps” which points to the first component belonging to that particular attribute. The component information is stored in the structure “comp-Type”. The field “n” is a pointer to the next member of the same attribute (next entry in the column) and “n1” points to the currently selected attributes in the column (this list is displayed under the column at a particular instance of the query). Note that “n” remains static during the query process whereas n1 gets updated based on the query under progress.

The structure “comp_type” contains the field “name” which contains the name of the component. The field “n” points to the next component “p” points to the previous component, “n1” points to the next component in the currently selected list (this list appears under the column “Comps Selected” in the GUI), again “n” remains static whereas “n1” changes with the progress of queries. The field “class” points to an array of pointers. This array contains pointers to the members of column “attributes” (example CMOS of column TECH) to which the component belongs. Hence the information about a particular component can be obtained by accessing the members of each of the column (to which the component belongs) through this array. Similarly the field “nclass” points to an array containing pointers to the next component in the lists (of the respective members of the attribute columns) belonging to the same members (of attribute columns). Hence starting from the structure “class_type” the first component of the member class-type can be accessed by the field “comps” in the structure and the subsequent members can be reached by using the pointer of the index [i] (example 1,2,3 etc. in Figure 5) in the array pointed to by the “nclass” field.

The structure “index_type” can be used for indexing upon the list of classes or components, it contains a union member “node” which can point to either a class_type structure or a component type structure depending upon the application. The field “n” points to the next element in the linked-list of structure index_type.

4.4 Version & concurrency control.

OCT does not directly support either versioning or concurrency control. Versioning should be handled by the design tools that interface with this LMS. Also concurrency is not a major concern for this application since most applications we plan to support are read only. Concurrency is only an issue when a new library is being created or new component data is being entered or deleted, or existing component data is being updated or corrected. All these activities will be controlled by our LMoct (Library Manager for OCT) facility which will function much like DMoct in helping the user to manage the library data and will rely on the UNIX file system's time stamps for versioning and concurrency control.

5.0 Current work and suggested future improvements

Work is under way to develop a yet more versatile user interface. Although the browser has a reasonable response time, there is room for improvement in the initializing time of the database, a major factor is the sorting required at the opening of the database. This is because OCT does not provide a mechanism for storage (retrieval) of data in a presorted manner. We are also incorporating a hierarchical taxonomy in the browser.

References

[1] Granacki, John J., “Research in Information Science and Technology: Systems Assembly Core Research,” Final Technical Report, USC/Information Sciences Institute, November, 1992.

[2] David S.Harrison, Peter Moore, Rick L. Spickelmier, A. R. Newton, “Data Placement and Graphics Editing in the Berkeley Design Environment,” *The proceedings of the IEEE International Conference on Computer -Aided Design*, pp 24-27, Nov,1986.

[3] Wayne Wolf, “Object-Oriented Programming for CAD,” *IEEE Design and Test of Computers*, pp. 35-42, March, 1991.

[4] S. Ahmed, A. Wong, D. Sriram, and R. Logcher, “Object-Oriented database management systems for engineering: A comparison,” *Report, Intelligent Systems Laboratory, Department of Civil Engr,1-253,MIT, Cambridge, MA 02139.* pp. 27-44, June, 1992.

[5] Rick Spickelmier and Brian C. Richards, “The OCT data manager,” in *Anatomy of a Silicon Compiler, Robert W.* Brodersen. eds., pp. 11-24, publication date.....

[6] Ketabchi, M. A., S. Mathur, T.Risch, and J. Chen., “Comparative analysis of RDBMS and OODBMS: A case study,” *Proceedings of Compcon IEEE Computer Society International Conference, San Francisco, CA,* February, 1990

[7] H. Afsarmanesh, E. Brotoatmodjo, K. J. Byeon. Alice C. Parker., “The EVE VLSI Information Management Environment,” *Proc. ICCAD -89,* pp 384-387.

[8] M. A. Breuer et al., “C base 1.0: A CAD Database for VLSI Circuits using Object Oriented Technology,” *Proc. ICCAD 88,CS Press, Los Alamitos, CA, order no. 869,* pp 392-395.

[9] Paul McLellanl, “Effective data Management for VLSI Design,” *22nd Design Automation Conference*, pp 652-657, paper 40.2, 1985.

[10] Michael R. Blaha, William J. Premerlani and James E. Rambaugh, “Relational database design using an Object-Oriented methodology,” *Communications of the ACM*, pp 414-427, vol 31, number 4,April 1988.

[11] Granacki, John J., “Printed Circuit Board Fabrication and Assembly Service: User Guide,” USC/Information Sciences Institute, November, 1992.

APPENDIX A. Data Structure

```
/* This is the list of different bags to be used in OCT*/
#define TECHNOLOGY “tech”
#define PACKAGE “pkg”
#define GENERIC_DESCRIPTION “gendes”
#define GENERIC_ID “genid”
#define COMPONENT_LIST “comps”
#define MANUFACTURER “mfr”
#define FACET_TYPE “MFR”

#define sgn(x) (x/abs(x))
#define NAME_LEN 100

typedef struct comp_type_type
  {
  struct class_type_type **class;
  struct comp_type_type **nclass;
  struct comp_type_type *n,*p,*n1;
  char *name;
  } comp_type;
comp_type *get_comp_type();

typedef struct class_type_type
  {
  comp_type *comps;
  struct class_type_type *n,*n1;
  char *name,*type;
  } class_type;
class_type *get_class_type();

typedef struct lib_type_type
 {
 FILE *fd;
 comp_type *comps,*selected_comps;
 struct lib_class
   {
   char *name, input[NAME_LEN], **selected_names, *type;
   int num_selected;
   class_type *list;
   } (*class);
 char *lib_name,error_str[256];
 char comp_input[NAME_LEN],**selected_names;
 struct index_type_type *indexed_comps;
 int oid,num_selected,num_classes,modified;
 } lib_type;
lib_type lib;
```

```
#define INDEX_LENGTH 100        /*Every next element of the index list points to the
                                  INDEX_LENGTH'th component in the lib->comps
list*/
typedef struct index_type_type
  {
  union node_type
    {
    struct class_type_type *class;
    struct comp_type_type *comp;
   node;
  struct index_type_type *n;
  } index_type;
index_type *get_index_type();
char **create_class_name_array(),**create_component_name_array();
char *get_class_name_of_type();
```