The Design and Implementation of Class Interface Manager of Unix Dee

Joseph Koi Kwok Yau

A Report
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements for
the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

Sept. 1992

ISBN 0-315-84656-9

Abstract

The Design and Implementation of Class Interface Manager of Unix Dee

Joseph Koi Kwok Yau

This report describes the design and implementation of the Class Interface Manager (CIM) layer of the Dee Compiler. The Class Interface Manager is a database maintaining class interface information and attribute information generated by Dee programs. It provides read, write and query services to the Dee Semantic Analyser (SA), the Dee linker (DINK) and the Dee browser (RI). The database is able to support multiuser access. In this report, we describe the design and implementation of the Unix version of Dee's CIM and a design proposal for the next version of the CIM.

Acknowledgments

The author wishes to express the deepest gratitude to his supervisor, Dr. Peter Grogono, without whose careful supervision and patience this work would not have been possible. The author also wishes to thank Benjamin Yik-Chi Cheung and Lawrence A. Hegarty for their valuable help and discussions.

Abbreviations

OOPL	Object Oriented programming languages.
ALN	Attribute list node.
AST	Abstract syntax tree.
AT	Attribute table.
CIDB	Class interface database.
CIM	Class interface manager.
CIM-AST	Abstract syntax tree for CIM.
CLN	Class list node.
CMM	Cache memory manager.
COMM	Communication manager.
CTN	Class tree node.
DBM	Database manager.
DINK	Dee linker.
FI	Functional interface layer.
FLM	File locking manager.
HS	High system.
LFI	Local functional interface layer.
LPU	Local processing unit.
LS	Low system.
MTN	Method tree node.
RFI	Remote functional interface layer.
RI	Read interface (Dee browser).
RPU	Remote processing unit.
SA	Semantic analyser.
TRAN	Translation layer.
VTN	Variable tree node.

Contents

List of Figures

Chapter 1

Introduction

Object oriented programming languages (OOPL) introduce the concept of programming using *classes*, *methods* and *inheritance*. These concepts make the development and organization of software more "natural" and make programs easier to maintain. It is also because of these new concepts, that the approach in the design of a traditional compiler cannot be applied to the design of an OOPL compiler. A mechanism is needed to access the massive amount of class interface information during the compiling process.

1.1 Background

Dee is a strongly typed, object oriented language and environment with advanced facilities for developing and maintaining programs developed here at Concordia University. Currently there are three implementation of Dee.

- PC-Dee version 1: Dee[8].
- PC-Dee version 2: Dee[12].
- Unix[1] Dee: Current version of Dee.

[1]Unix is a registered trademark of AT&T Laboratories

The major components of Dee are the parser, the semantic analyser, the linker, the class interface manager and the browser. There are two types of final physical outputs generated by the compiler: the object code and the *interface* of a class. Both of these files are generated by the compiler if the compiler detects no error in the source code. The class interfaces generated by the compiler are stored in a database. Only the compiler can update this database. The linker and the browser, however, can only read from the database. The interface of a class contains implementation details of a Dee class. Different programmers see different *views* of this interface. A programmer who chooses to inherit a class will see a more detailed interface compared to one who is a client. During a compiling session, the compiler reads interfaces from a database and has access to implementation details that are not shown to the programmers. On the other hand, by using the browser, programmers can see selected comments which are not needed by the compiler. Therefore a Dee programmer can use the browser to display information about the view of a class or the inheritance relationship between different classes. The information provided by the browser simplifies the programmer's task to develop a Dee application. Implicitly speaking, the database is designed to support object oriented development. The compiler writes interface information to the database and the database in turn supports the compiler, the linker and the browser.

Other examples of object oriented programming languages with a browsing interface are Smalltalk [6, 7] and Eiffel [20, 21]. Neither of these programming languages use a database to store valid class view or interface information. The browser of these systems extract information directly from the source text of a class. The OMEGA programming system [18]however uses a relational database to provide different views of a program. It has a similar approach to Dee but, as in Smalltalk and Eiffel, the stored information is obtained from program text. In these kinds of implementation, the browser may have the chance of providing out-of-date information to the programmer. This happens when a programmer makes an update to the source and uses the browser for query without recompiling the new source. In this case, the source code and the compiled code may be out of step. As a result, the browser may not provide valid view or interface information to the programmer. In Dee, a database is used to store only the valid compiled class. The browser extracts interface information from the database only, therefore eliminating the out of step situation.

A Dee program is a collection of *classes*. It consists of a *root class* and the classes needed by the root class. A class consists of a header and a list of attributes. During the compiling process, the source text of a class, e.g. C, is read by the compiler. The compiler gets class interfaces information about other classes, which C depends on, to perform type checking and validation. If the compilation is successful, a class interface of C is generated by the compiler. All such interfaces are stored into a database. So only valid classes' interface are stored in the database. As a result, by using the database, the browser always can provide up-to-date valid class views to the programmer.

1.2 Class interfaces

A Dee program consists of a *root class* and the classes needed by the root class. A class consists of a header and a list of attributes. A class interface of a class is an extract of the class. It consists of two types of information: class information and its associate attribute(s) information.
Each Class interface contains the following information about the class itself:

1. Name of the class.
2. Parameters in the class declaration.
3. Comment in the class declaration.
4. Inherit list.
5. Extends list.
6. Uses list.
7. Ancestor list.
8. Attribute list.

Each item in the attribute list of the class contains the following information about the attributes of the class:

1. Name of the attribute.
2. A flag indicating whether the attribute is a variable or a method.
3. A flag indicating whether the attribute is public or private.
4. Comment in the attribute declaration.
5. Parameters of the attribute declaration.
6. Result of the method.
7. A flag indicating whether the attribute is a method or a constructor.
8. A flag indicating whether the attribute is concrete, abstract or special.
9. Defined by class name.
10. Implemented by class name.

Not all of the interface information is extracted directly from the Dee source program. Part of the information in the class interface is generated by the compiler, not written by the programmer.

In the implementation of Dee, the class interfaces are stored into a database. There are three implementations of such a database:

- PC-Dee version 1: Dee[8]: Class interfaces in many small files.
- PC-Dee version 2: Dee[12]: Class interfaces in a B-tree.
- Unix version: Class interfaces described in this report.

The database is designed to suit the need of the compiling process in various stages. The stages are the Semantic Analyser (SA), Dee Linker (DINK), and the browser (RI). RI stands for *read interfaces*.

- SA requires class interface information from the database to do type checking and other validation.
- DINK requires class interface information to generate C code.
- RI requires class interface information to display the inheritance information

The Class Interface Manager (CIM) is built on the database (CIDB) to provide query and update services about class interface to the semantic analyser, DINK and RI.

1.3 New CIM

The performance of the Class Interface Manager of the PC version of Dee [1] was not satisfactory. The inefficiency was due to the limitation of the architecture of the index file system that PC-Dee used in its implementation. The data schema of the class interfaces contained a lot of repeated and redundant elements for each of the classes stored.

A direct disk access method is used by CIM to maintain its database in the PC version of Dee, therefore the system would spend most of its time (about 80%) of the update just for the updating of the index file. The performance is not acceptable for serious use of the compiler.

In addition to the above, the PC version of Dee only supports single user system (MS-DOS). With the price of Unix machines today and their multiuser capability, it is reasonable and feasible to develop a new version of Dee on Unix platforms that supports multiuser access.

In designing Unix Dee, CIM is one of the modules that needed improvement. The new design of CIM should consider the following goals:

- Efficiency Improve query/update access times.
- Redundancy Reduce redundancy of storage.
- Resource sharing Allow sharing of class interface database.
- Data Consistency An integrity strategy is needed to prevent the corruption of CIDB due to the introduction of resource sharing schema.
- Portability The source code should be portable to other Unix platforms.

Achieving these goals involved total rewriting and redesigning the access methods and data structures of the CIM. In addition, some kind of resource locking mechanism for the database had to be introduced to allow multiuser access. The system should be coded for a standard C compiler on a Unix platform to make system migration simple. The GNU GCC[2] compiler is used in this project to write the source code.

In Chapter 2 of this report, the detailed design and implementation of the current version of Unix Dee's CIM is discussed. In Chapter 3, a proposal of the future version of CIM is presented in detail.

[2]GNU and GCC is a trademark of the Free Software Foundation, Inc

Chapter 2

CIM: Cache Model

2.1 The Idea

The primary function of the class interface manager is to provide database access services to its clients. In our case the clients are the Dee semantic analyser (SA), the Dee linker (DLNK) and the Dee online browser (BI).

There are three kind of Dee libraries used by Dee. The *standard Dee library* containing standard basic Dee class, the *user Dee library* containing user's own Dee class and the *shared Dee library* containing shared Dee class for a team of Dee programmers. Therefore three kinds of databases must be maintained by the CIM

The following are the three kind of databases (CIDB) maintained by the CIM.

- The system class interface database contains the class interface information which is generated by SA through CIM using the standard class library. This database contains the default standard class interface information used by every user.
- The shared class interface database contains the class interface information which is generated by the CIM through SA using the shared class library. This database contains class interface information shared by a group of users
- The user class interface database contains class interface information of the

user's Dee code. All Dee users have their own database to store their own class information.

When the compiler needs interface information of a class, it will send a request to the CIM. The CIM will then search the class interface information in the user class interface database. If the class does not exist in the user class interface database, the CIM will then search the class interface information in the shared class interface database. If again, the class does not exist in the shared class interface database, the CIM will search the class interface information in the system class interface database.

In the PC-Dee, the main contribution to inefficiency of the CIM is the disk access required to update the indexes. Since in the PC version the size of the index file is quite big, due to the design of the data structure of the database, it is very expensive to perform updates and queries. In the event of an update, the system spends about 80% of the time to perform the update of the index file alone. Since direct disk I/O is used to perform the update, the system spends most of its time performing disk I/O by moving the disk head back and forth.

In order to reduce the number of disk accesses and hence the time to perform I/O, the Unix CIM uses a region of memory as a cache for disk accesses. This region is used to store intermediate and final results before they are written to the secondary storage. The data in CIDB consists of two index structures and a large number of records containing class information and attribute information. Any updates or queries are performed in the cache memory through the two index structures. When the update is finished, the data in the cache are copied directly to secondary storage without further transformation. This greatly reduces the amount of disk I/O compared to the PC-Dee's CIM.

To allow multiuser access, a simple locking manager is used by the CIM to manage resource sharing among users. Another purpose of the locking manager is to keep the database from getting into an inconsistent state due to the introduction of multiuser accessibility.

The data structure of the index and the data structure of the records are designed in such a way that only necessary information is included, thus reducing the redundancy experienced in the first version of PC-Dee.

2.2 System Architecture

In designing the structure of the CIM, a top-down design approach is used. A higher level of abstraction of the system's structure is first identified with its major functional areas. The design at this level of abstraction is called the *logical architecture*. It describes the major functional areas of the system and their inputs and outputs without going into details. A lower level of abstraction of the system's structure can be obtained by further extending each major functional area into details. The *physical architecture* describes the physical design of the system in reasonable detail. We use the top-down design approach to help us organize the design of the system. In the next two sections we describe the logical architecture and the physical architecture of the CIM.

2.3 Logical Architecture

The goal of the design of the CIM was to achieve an architecture in which several almost independent components communicate through simple interfaces. In designing the functional architecture of CIM, functional areas that are highly independent of each other are identified. They are the *functional interface* (FI), the *database manager* (DBM), and disk I/O layer. Figure 2.1 shows the logical architecture of Unix CIM.

The functional interface layer provides an interface to communicate with the external environment. It acts as an interface for CIM to communicate with other components of the Dee system. Update/query requests are received through this interface. The FI's job is to transform the request into a set of DBM layer's operations and send them to the lower layers for further processing.

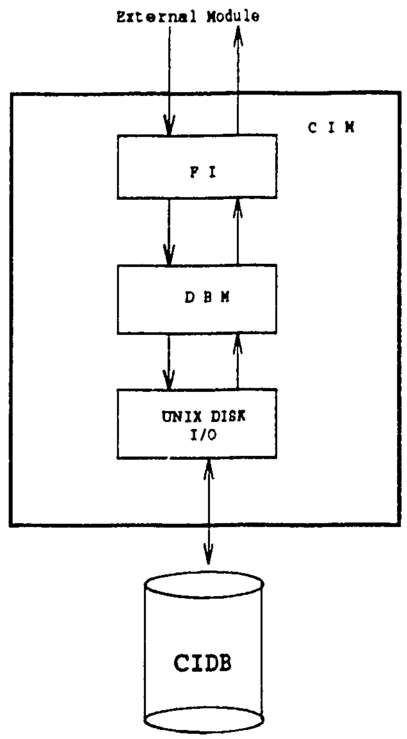

Figure 2.1: Logical Architecture of Unix CIM

The database manager layer consists of all the procedures necessary to manage the database in a logical sense. Actual searching and update on the database are performed only by the DBM primitives.

The disk I/O layer contains primitive Unix operating system's file I/O procedures. This layer contains Unix I/O calls such as fopen(), fclose(), fread() and fwrite().

2.4 Physical Architecture

Figure 2.2 describes the actual physical architecture of CIM with its processing units, mass storage device and its interconnections. The physical architecture of CIM is divided into 3 parts:

- *high system* (HS).
- *low system* (LS).

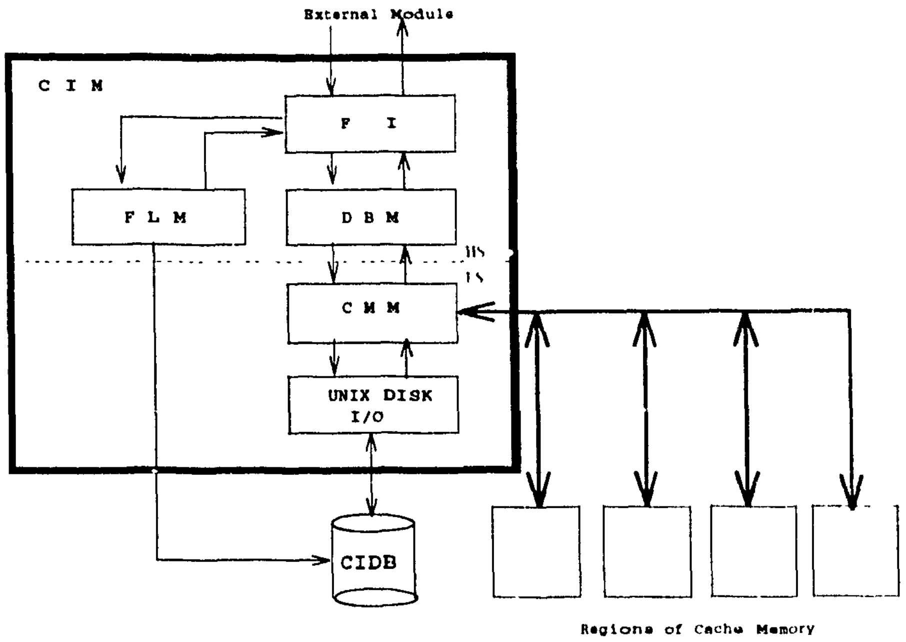

Figure 2.2: Physical Architecture of Unix CIM

- Physical storage.

Basically the HS is devoted to the management of the update/query transactions and resource locking. The LS is devoted to the management of the management of the cache memory and the management of the file I/O system.

2.4.1 High System

The HS consists of *functional interface* layer (FI), *file locking manager* (FLM), and *database manager* (DBM). The FI uses services from both FLM and DBM. It provides the functional interface to communicate with the external world. Update/query request are received through the interface and are sent to the DBM. The FLM provides a file locking service to the FI. The DBM performs database update through the LS which acts as the *cache memory manager* (CMM).

2.4.2 Low System

The LS consists of the cache memory manager (CMM) and the Unix Disk I/O layer. The CMM acts as an intermediate stage between the database manager and the Unix disk I/O layer. It handles the cache memory management and performs high level file I/O. Any update/query request sent down by the DBM are performed on the cache memory before being written to the secondary storage. The CMM provides cache memory management and cache memory update/retrieval/allocation services as well as high level file I/O services to the DBM.

2.5 Data Structure

During the compilation of a Dee class, an abstract syntax tree (AST) is created at runtime by the SA[14]. The AST contains information which will be used by the code generator. Since only part of the information in the AST are useful to the SA, DINK, and RI, keeping the full AST in the secondary storage as the CIDB is a waste of secondary storage. The solution is to create another AST which stores only the interface information needed by the SA, DINK and RI.

The new CIM abstract syntax tree (CIM-AST) is created in the cache memory at the same time as the AST is created. It contains information extracted from the AST. Each node of the CIM-AST contains only the data needed by the SA, DINK and RI. Each node contains information of a class interface. The CIM-AST is created by invoking an update request to the CIM-DB through CIM's functional interface. The data structure of CIM-AST is defined in the DBM which is in the HS. Figure 2.3 describes the overall index structure of the database.

There are two physical entry points to access the database. These entries are made through two index structures. One is through the hash table index structure and the other is through the CIM-AST index structure.

Generally speaking, the hash table index stucture organizes the class interface

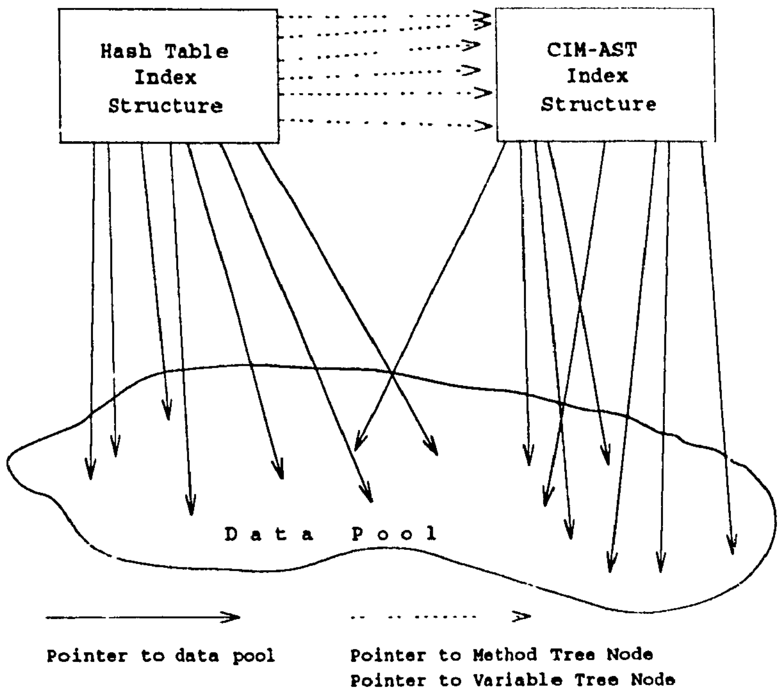
Hash Table Index Structure
CIM-AST Index Structure
Data Pool
Pointer to data pool
Pointer to Method Tree Node
Pointer to Variable Tree Node

Figure 2.3: CIM's Overall Index Structure

information by attributes. It contains implicit information such as "in what classes a particular attribute is declared". Whereas the CIM-AST index structure organizes the class interface information by classes. It contains implicit information such as "for a particular class, what are the attributes it declares".

The browser requests class interface information that are indexed by the hash table and CIM-AST. The semantic analyser(SA) and the linker (DINK) request class interface information indexed by CIM-AST only. CIM-AST has indexes pointing to the data pool which contains records of class interface. These records are of variable length. The hash table has indexes pointing to the nodes in the CIM-AST.

In the event of an update/query, the request is sent through the functional interface (FI), the database manager (DBM) and then to the cache memory manager (CMM). The database is updated or queried by the DBM through the CMM's service routine in the specified *cache memory region.* The CMM updates the database through the CIM-AST index structure first and then through the hash table index structure in the *cache memory region.* The indexes of the two index structures are

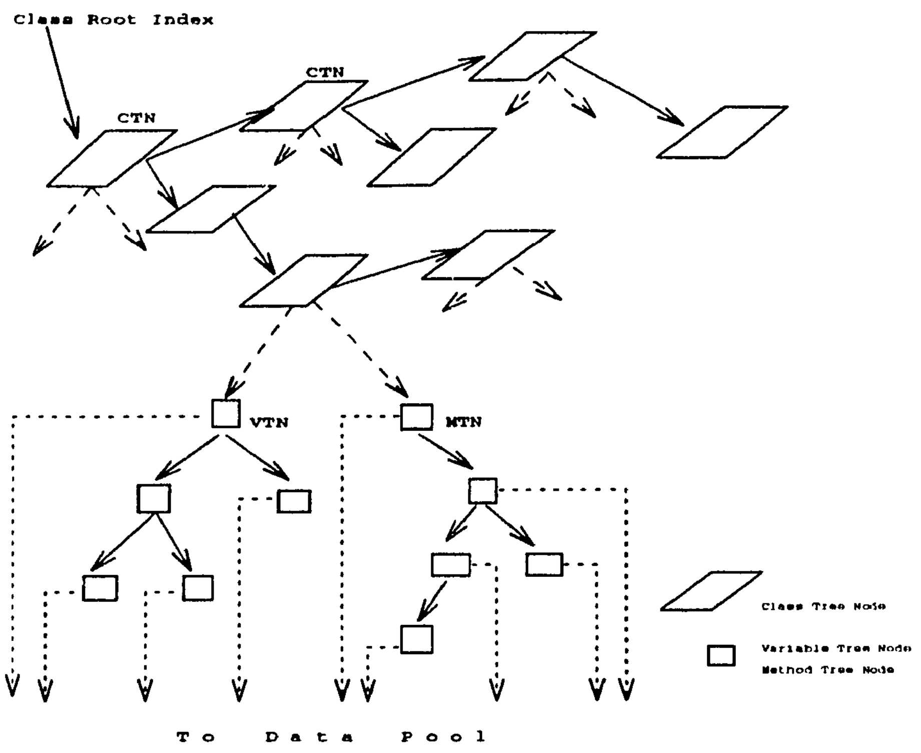

Figure 2.4: CIM-AST Index Structure

defined as the CIM *cache memory address*. Multiple sets of these structures exist due to the fact that there are multiple sets of databases (*user*, *share*, *standard*) in the system.

2.5.1 CIM-AST

All the classes and their associate attributes' information are indexed by a tree structure. Figure 2.4 shows the physical structure of the CIM-AST index structure. On one dimension of the tree structure, it is a *binary search tree* of nodes sorted by class name. Each of these nodes is called *Class Tree Node* (CTN). The sort *key* of CTN is the *class name*. On the other dimension, the roots of two *binary subtrees* are indexed by the CTN. The nodes of these subtrees are called *Method Tree Node* (MTN) and *Variable Tree Node* (VTN). The binary subtrees are sorted by the *method name* and the *variable name* respectively.

In addition to the index to its method tree and variable tree, the CTN also contains indexes to the data pool which contains records of class interface information, such as *class name, class parameter, comment, inherit list, extends list, uses list, ancestor list.*

Each method tree node (MTN) and each variable tree node (VTN) also contains indexes to the data pool which in turn contains attribute-related information of their associated class, such as *attribute name, attribute kind, attribute type, comment, method parameters, result of method, etc.*. The data structure of CIM-AST is defined in the database manager (DBM) layer.

The entry point of the CIM-AST index structure is called a *class root index*. It points to a class tree node (CTN) which is the root of the CIM-AST.

2.5.2 Hash Table

A hash table contains an *Attribute Table* (AT) of 26 slots and a number of linked lists. It is a hash table managed by a very simple hashing algorithm (the first letter). Figure 2.5 shows the structure of the hash table. The slots of the Attribute Table are the entry points of the index structure. The *hash key* is the *attribute name* and the *hash index* is the first letter of the *attribute name*. The hash function is not case sensitive. Each slot has an index pointing to a linear linked lists of *Attribute List Nodes* (ALN).

The ALN are sorted with the *attribute name* as the *key*. Each ALN has an index pointing to another linear linked lists of *Class List Node* (CLN). In addition to these the number of CLN under each attribute is also recorded in the attribute list node (ALN).

The class list node CLN is sorted with *class name* as the *key*. It contains an index to an associate Attribute Tree Node (ATN) of the CIM-AST subtree index structure

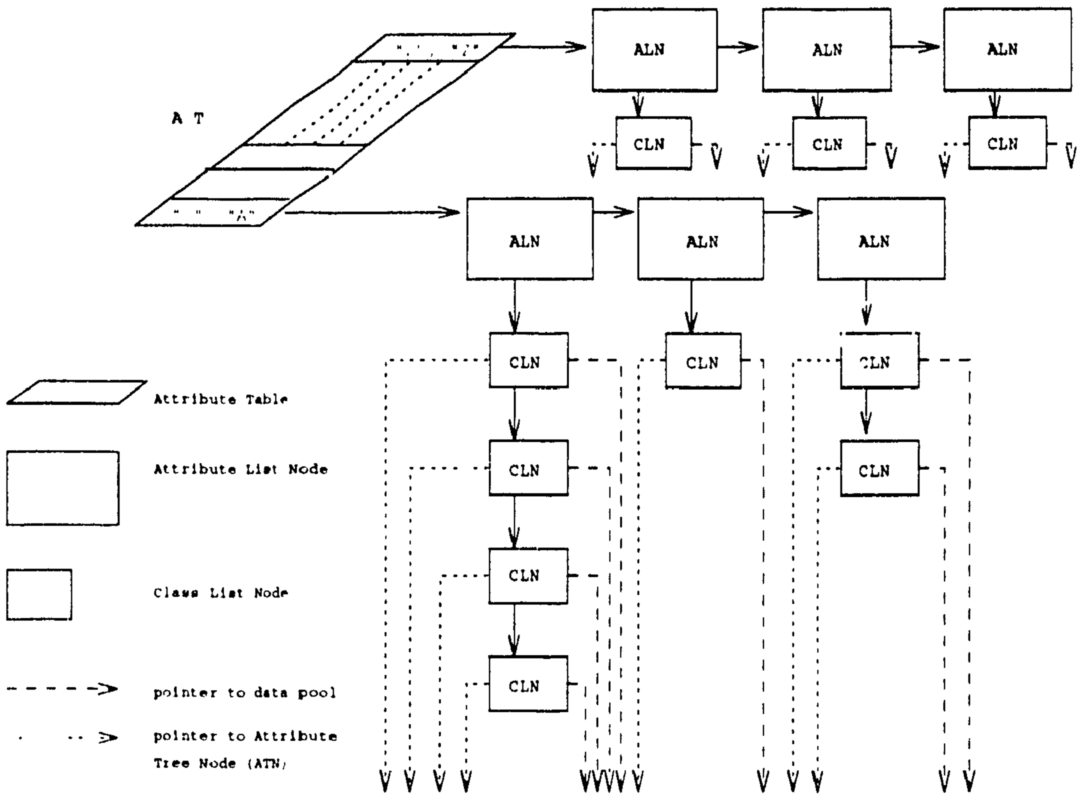

Figure 2.5: Hash Table Index Structure

2.6 CIM Memory

2.6.1 Image and Memory

The CIM uses a cache device to keep the database in the memory. Any update or query are performed by using the database in the cache. The cache is a large cache memory where intermediate and final results of data are stored before they are written to the secondary storage. Since all intermediate updates and queries are performed using cache memory, the number of physical file I/O transfer is greatly reduced. The cache memory is maintained by the cache memory manager (CMM). Several cache memory regions are used to keep the CIDB.

An *Image* is a snapshot of the runtime memory used to keep CIM-DB. An image represents the database in bit map form. There is a *standard image*, a *share image*, a *user image* and a *temporary image*. Each of these *images* resides in separate cache memory region. When a specific *image* is copied to secondary storage, the bit map of the cache memory region is copied bit for bit to a physical file. The resulting file is called an *image file*. There are three types of file corresponding to the three types of image: the *standard image file*, the *share image file* and the *user image file*.

An image is only created when an image file is cached into the cache memory region. The address in the cache memory region is called the CIM *cache memory address*.

2.6.2 The Cache Memory Manager

The cache memory manager is a physical layer under the database manager DBM. It is responsible for performing various cache memory management services for the DBM. These services include cache memory allocation, some cache I/O services and address conversion between CIM cache memory address and Unix runtime memory address. The conversion is necessary because of the way in which Unix responds to a request for memory. In the next section, we will discuss the conversion process in detail.

2.6.3 Unix Memory Address Space

In SunOs[1], the physical runtime memory is divided into pages. The size of each page is 4,096 bytes. The whole memory address space can be represented as a linear linked list of pages. In a logical sense, this address space is one long continuous address space, but in a physical sense it is not.

When a *malloc()* call is received by the Unix runtime memory system, the system allocates memory space from the pages of the runtime memory. If there is not enough space on the current page to satisfy the request, *malloc()* allocates all of the remaining space on the current page and as much additional space as required from further pages. Since the requested memory comes from two different pages, an improper memory alignment always occurs at the end of the first page and at the beginning of the second page. Therefore writing to the memory returned by *malloc()* results in having contaminated data in the memory.

The only way we can prevent this kind of event from happening is to keep track of the size of the free memory in the current page. This can be achieved by building another memory management system on the top of the Unix memory system and letting it monitor the memory allocation process.

2.6.4 Cache Memory Address Space

To prevent memory space being split in two different pages, the cache memory manager (CMM) is created on top the Unix memory manager to monitor the memory allocation. It has a similar architecture to the Unix memory manger.

One of the services provided by the cache memory manager (CMM) is address conversion. It provides a service to convert address format between CIM cache memory

[1]SunOs is a registered trademark of Sun Microsystems, Inc

addresses and the Unix runtime memory addresses. The CMM maintains a linear linked list of Unix runtime memory pages. The page length is 4,096 bytes.

When a *cache_malloc()* request message is received, the cache memory manager CMM will compare the requested memory size to the free memory size in the current memory page. If the current memory page has enough memory to allocate the required space, CIM will allocate the memory in the current page. In the event of insufficient memory in the current page, the system will allocate memory on the next page. The CIM cache memory address is the only memory address which is accessible by the DBM.

2.6.5 Page Structure

There are four cache memory regions maintained by the cache memory manager (CMM), each of which is indexed by a table containing the following information:

1. The address of the first memory page of the cache memory region linked list.
2. The address of the first free memory page in the linked list.
3. The address of the first available free memory in the free memory page.
4. The size of the cache memory.
5. The size of the free memory.
6. The size of the available memory in the first free memory page.

2.6.6 Indexing and Swapping the Cache

All CIM cache memory addresses within a region are relative to the start address of the region. The minimum offset is 1. The maximum offset is the size of the region. Offset 0 represents a null pointer.

Since any query/update operation is performed in the cache memory regions, it is possible for the memory regions to have memory fragmentation due to a delete operation. The data structure of the memory does not allow us to perform memory de-allocation of a segment in a page. This means in order to maintain the consistency of the cache memory address, segments of fragmented memory cannot be reclaimed individually. Segment fragmentation is cumulative and will increase the load time of the image. A memory swapping facility is implemented to eliminate fragmentation. Memory swapping in cache memory region is done automatically by the Dee system. Memory swapping will be triggered when a pre-set number of delete events have occurred or during memory allocation time. Example of these delete events are class deletion and class update.

CIM keeps three memory regions: one for the standard classes, one for the shared class classes and one for the user's own classes. In addition, there is a temporary region for memory swapping. The default standard class region is static in size and will not change during runtime. The users' region and the shared class region are dynamic in size. These regions will increase their size when the user compiles a new class of the corresponding class library. These regions also will be updated when the user has modified the interface of an existing class of the corresponding class library.

The condition which triggers the swapping mechanism is when the number of delete event has reached to a pre-set limit or when the cache memory system has used up all the pre-allocated cache memory.

2.6.7 Algorithm

The cache memory allocation algorithm is listed below:

1. If current free cache memory space is bigger then required, goto step 3.
2. *Expand* current cache memory.
3. Allocate the current cache memory regions.

The swapping of cache memory region performed by *expand* is listed below:

1. Move the current cache memory pages to the temporary cache memory region
2. Allocate a new current cache memory region from system with size 1.5 times of the old region.
3. *Copy* the logical content of the temporary region into the new current cache memory.
4. De-allocate the memory in the temporary cache memory region.

The *copy* operation in step 3 is defined as follows:

- Traverse the CIM-AST of the temporary image in preorder. For every unmarked node, build a CIM-AST in the specified image. When it is complete, the resulting image will not have any memory fragmentation.

2.7 Resource sharing

2.7.1 Locking Schema

One of the major goals in this project is to introduce multiuser capability to the CIM and hence to the Dee system. One way to avoid corrupting the database in a multiuser environment is to require that access to the database be done in a mutually exclusive manner. This means that when a process is performing an update/query operation on a database, no other process can modify the same database.

In the CIM, the resource locking schema is controlled by the file locking manager (FLM). There are two type of requests received by the functional interface (FI): update/query request and lock request. Lock request received by the functional interface will involve a call to FLM to perform a lock on the image file. If it is a query request, a read lock will need to be applied to the image file before the query request

	fcntl(readlock)	fcntl(writelock)
fcntl(readlock)	Granted	Not Granted
fcntl(writelock)	Not Granted	Not Granted

Figure 2.6: Compatibility Matrix of Locks

is sent to the database manager (DBM). For an update request, a write lock will need to be applied to the image file before the update request is sent to FI from external modules. Request from the RI will require a read lock on the database, whereas request from the compiler will require a write lock on the database.

In the implementation of the file locking manager (FLM), FLM uses the Unix system file control call *fcntl()* to lock the image file. There are two kinds of *fcntl()* lock: a read lock and a write lock. The locking of a database file is performed by a *fcntl()* call in the FLM layer. The compatibility relation between the two modes of locking can be represented by a compatibility matrix. Figure 2.6 shows the compatibility matrix. If a read lock is currently applied on the database, a further read lock can be applied to the database at the same time. If a read lock is currently applied on the database, any further write lock will be blocked until the current read lock is freed. If a write lock is currently applied on the database, any further read lock or write lock will be blocked until the current write lock is freed.

2.7.2 Algorithm

The following algorithm performs a read lock on the current image file. This algorithm will block the granting of read lock if there is currently a write lock applied on the database.

```
loop
   exit when fcntl(readlock) granted or timeout
end loop
If timeout
   then return "lock request not granted"
   else return "lock request granted"
end if
```

The following algorithm performs write lock on the current image file. This algorithm will block the granting of write lock if there is currently a read lock or write lock applied on the database.

```
loop
   exit when fcntl(writelock) granted or timeout
end loop
If timeout
   then return "lock request not granted"
   else return "lock request granted"
end if
```

2.8 Data Consistency

One of the important issues in designing the CIM is data consistency. The CIM should be able to keep the database from data corruption due to a crash during a compiling session. The update to the database should be atomic. Either the database is updated successfully or the update is not done. The combination of the locking mechanism and the cache mechanism provides a scheme to maintain the database in a consistent state. In order for a user to use the database, the database will first be locked by the compiler process and then the content of the database will be read into the cache memory. Any update or query is then performed by using the cache. If during this time the compiling process crashes, the database in the secondary storage does not get corrupted. The crashing of a process usually involves the killing of the process either by the user or by the system. For example, the killing of a window which is compiling a new class terminates the compilation. An update is considered successful if it completes writing the cache to the secondary storage. If during the update (write operation) to the secondary storage, the system crashes, the update does not take effect and the contents of the previous version of CIDB do not get corrupted. In any case, the database is free of corrupt data.

2.9 Functional Interface

The functional interface (FI) provides an interface to let the compiler and the browser communicate with the CIM. The interface contains a number of CIM system routines. These routines have the form of a prefix MODULE. followed by the Action and the Object and then followed by the parentheses. Each of the functions returns a value of type STATUS which indicates either that the operation is completed successfully or that is failed.

- STATUS MODULE-ActionObject()

2.9.1 Interface for System

There are two routines that are used by the Dee upper level system. They are the routines to load and save the images. During system start up, the Dee system's initialization routine makes a call to the CIM module to load the database images into the system memory. The save image routine is also controlled by the Dee upper level system.

- Status CIMDB-LoadImage()
  –Load all CIM images into the cache memory.

- Status CIMDB-SaveImage()
  –Save the current CIM images to a image file.

2.9.2 Interface for the compiler and the browser

Aside from providing routines for the Dee upper level system, CIM also provides a number of routines for the compiler and the browser. These routines provide database update/query services as well as the database locking services to the compiler unit and the browser unit.

- Status CIMDB-WriteClass(*ClassRec*)
  –Insert a class record into the current image.

- Status CIMDB-WriteAttribute(*ClassName*, *AttributeRec*)
  –Insert an attribute record of a class into the current image.

- Status CIMDB-CheckClass(*ClassName*)
  –Check if a class is in the current image.

- Status CIMDB-DeleteClass(*ClassName*)
  –Mark a class and all of its attributes in the current image as deleted.

- Status CIMDB-ReadClass(*ClassName,&ClassRec*)

  Attempt to read interface information for the given class from CIM images. The function looks for the class first in the user image. then in the shared image. then in the standard image.

- Status CIMDB-ReadClassList(*ClassName,&linklist*)

  Attempt to get a linked lists of class interface records information for the given class in the current image.

- Status CIMDB-ReadClassAtt(*ClassName,AttName,&AttRec*)

  Read a record containing attribute information for a given class and a given attribute in the current image.

- Status CIMDB-ReadClassAttList(*ClassName,&linklist*)

  Get a linked lists of attribute information of a given class in the current image.

- Status CIMDB-ReadSameAtt(*character,&linklist*)

  Attempt to get a linked lists of attribute records of different class in the current image by giving the first letter of the attribute name.
  If success return DONE, otherwise return ERROR.

- CIMDB-Lock(*image_no, mode*)

  Attempt to apply a lock on the image indicated by *image_no*. To apply a read lock, set *mode* to READLCK. To apply a write lock, set *mode* to WRITELCK.

  Condition to grant a read lock is as follow:
  If a write lock is already on the image, this call will be blocked until the write

lock is released. Once this call is successfully executed, a read lock is applied on the image *image_no*.

If there are read locks on the image *image_no*, this call will not be blocked. Once this call is successfully executed, the read lock is applied on the image indicated by *image_no*.

Condition to grant a write lock is as follow:

If write locks or read locks are already on the image *image_no*, this call will be blocked until these locks are released. Once this call is successfully executed, a write lock is applied on the image *image_no*.

- CIMDB-UnLock(*image_no*)

  Release a read lock or write lock on the image indicated by the *image_no*.

- CIMDB-UnLockAll(*image_no*)

  procedure to release all the locks on the image indicated by the *image_no*

Chapter 3

A Proposal: Server Model

The cache version of CIM described in the previous chapter satisfied the goals that we set. It achieved our goal of minimizing the storage of data file. A file locking facility was introduced to manage multiuser access of the database. By using a cache, the response time of CIM was greatly improved comparing to the PC version of CIM. However a weakness of the Unix CIM is observed in the implementation:

- It does not provide adequate file security.

3.1 Deficiency in File Security

In the current implementation of Unix CIM, there are three database files maintained by the CIM. The user image file, the shared image file and the standard image file. The size of the standard image file is static due to the fact that no user can modify the standard image. The file protection mode of the standard image file is set to "world: read only" in the file system since reading is the only operation that a user process can perform it.

The size of the user image file and the size of the shared image file is dynamic. The only processes that can access the user image file are the user's own processes. The file protection mode of the user image file is set to "owner: read and write" in the file system.

However, in the case of the shared image file, the file protection mode is set to "world: read and write" in the file system in order to allow multiuser access. Since the shared image file is set to "world: read and write", a potential security leak exists. It provides the opportunity that an unauthorized process could modify or delete the image file. In order to fill up this security hole, a server is proposed in this chapter to handle multiuser access to the shared database.

3.2 The Idea

To overcome the deficiency in file security, a CIM remote server is created to handle the shared database's access. The ownership of the shared file belongs to the server process. Any update or query request to the shared database are performed by the server. Update or query request to the user and the standard database are performed by the user process through CIM. In this case the security hole is filled.

3.3 System Architecture

In this design the system architecture of the CIM is divided into two separate parts: the LOCAL CIM and the CIM SERVER. The CIM SERVER provides remote query/update services to the LOCAL CIM. A remote query/update request is an query/update performs on the shared database by CIM SERVER.

The LOCAL CIM processes query/update requests on the standard database and the user's database only. Any query/update requests on the shared database receives by LOCAL CIM is sent to the CIM SERVER through the Unix TCP/IP[1] for remote processing.

[1]TCP/IP stand for Transfer Control Protocol/Inter Process Communication

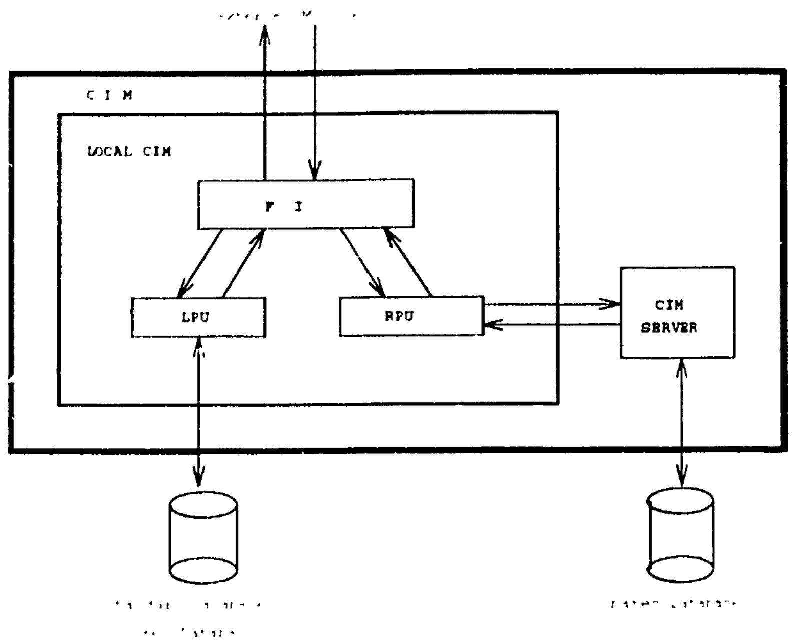

Figure 3.1: Physical Architecture of CIM (Server Model)

3.3.1 Local CIM

Figure 3.1 shows the physical architecture of LOCAL CIM. It is divided into three parts: the *functional interface*(FI), the *local processing unit* (LPU) and the *remote pre-process unit* (RPU).

The *functional interface* is the interface of CIM to the outside world. It routes different requests to different processing units. There are two kinds of request: *local requests* and *remote requests*. A local request is a query/update request on standard database or user database received by the CIM. A remote request is a query/update request on the shared database received by the CIM. The functional interface will route a local request to the local processing unit and will route a remote request to the remote preprocessing unit.

The way that LOCAL CIM work is as follows: Upon receiving query/update request from the external module, the *functional interface* will route the request to the *local processing unit* or to the *remote processing unit* according to which database file the request will act on. If a query/update request on shared database is received by the

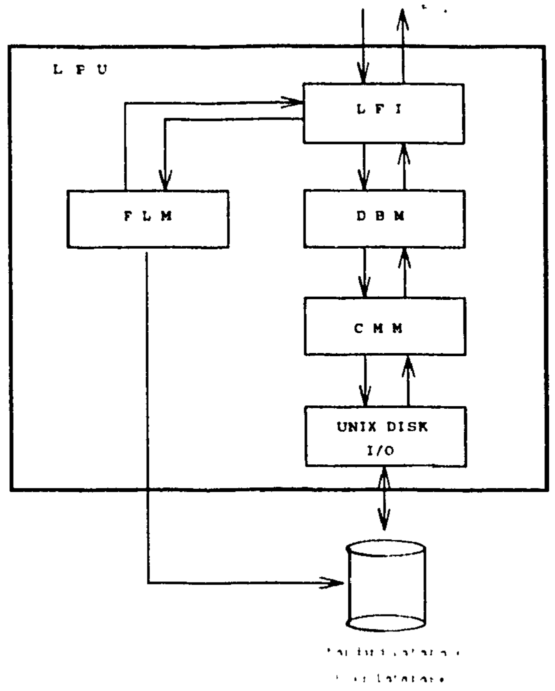

Figure 3.2: Physical Architecture of Local Processing Unit (LPU)

FI, it will send the request to the RPU for pre-processing. When the RPU finished processing the request, it will send the result to the FI and the FI will in turn send the result to the external module. If a query/update request on other (standard, user) database is received by the FI, it will send the request to the LPU for processing. The result is returned to the FI by the LPU which in turn sends it to the external module.

LPU

The *local processing unit* (LPU) is a module which is dedicated to processing the query/update request on the user's database and on the standard database. It consists of the *local functional interface layer* (LFI), the *file locking manager* (FLM), the *database manager* (DBM), the *cache memory manager* (CMM) and the Unix Disk I/O layer. It has exactly the same architecture as the version of Unix CIM described in chapter 2.

Local requests received by the local functional interface layer are transformed into

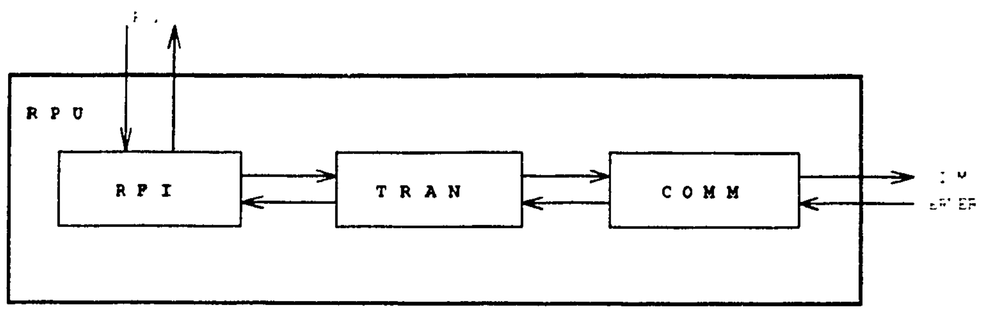

Figure 3.3: Physical Architecture of Remote Preprocessing Unit (RPU)

high level database requests. These requests are then sent to the database manager. The database manager in turn transfers the database requests into a set of low level database primitives. These primitives use the facility provided by the cache memory manager to perform the query/update on the cache. Once the database manager has finished the query/update, results are passed up to the local functional interface layer which in turn sends the results to the functional interface. Local locking requests received by the local functional interface layer are directed to the locking manager.

RPU

The *remote preprocessing unit* (RPU) is responsible for preprocessing the request before sending the request to the CIM SERVER. Basically speaking, it is made up of a translation layer, a communication layer and a communication link to the CIM SERVER. The translation layer is responsible for two things: packing each request into a textual form for communication purpose, and unpacking the result and transforming it into a standard interface format.

The actual architecture consists of the *remote functional interface* layer (RFI), the *translation* layer (TRAN) and the *communication manager* (COMM). Upon receiving a request from the *functional interface* (FI), the RFI sends the request to TRAN which will pack the request together with its parameters into a text record. The record is then sent to the COMM for external communication. The COMM layer is responsible for the communication between the RPU and the CIM SERVER. When a record is

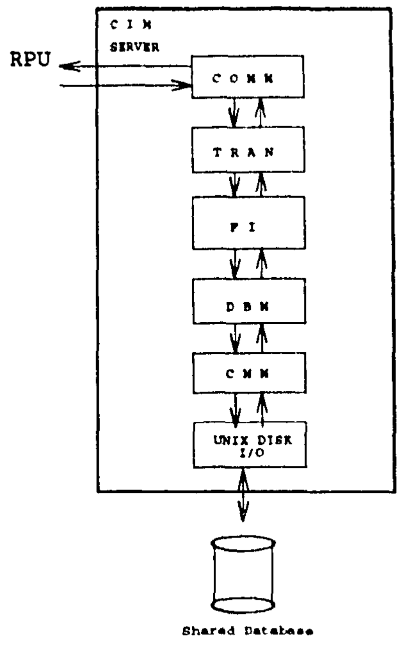

Figure 3.4: Physical Architecture of CIM SERVER Model

received by the COMM, the COMM will send the content of the record to the CIM SERVER for remote processing. COMM will then monitor the communication link for the response of the CIM SERVER. Once the CIM SERVER has finished processing the request, it will send the result through the communication link. The COMM will pick up the result and put it into a record and send the record to the TRAN for unpacking. After the TRAN has unpacked the result into standard interface form, the result is then sent to the RFI and then to the FI in the LOCAL CIM and finally to the external module.

3.3.2 CIM Server

The CIM SERVER consists of *communication manager* (COMM), the *translation* layer (TRAN), the server's functional interface (SFI), the database manager (DBM), the cache memory manager (CMM), Unix disk I/O layer. In a logical sense, it provides concurrent query/update services on the shared database. There can be multiple communication linkages existing between CIM SERVER and several remote processing unit (RPU) links at any one time. The COMM layer constantly monitors the commu nication link for the arrival of packed messages. Since there may be multiple links existing at any one time, the monitoring of each link is done by scheduling.

Once packed requests are received by the COMM, packed requests are kept in a request queue. The COMM is responsible for the communication system and event scheduling. The scheduling mechanism is used to monitor the request queue. Once a complete request is spotted in the head of the queue, the complete request record will be send to the TRAN for unpacking into the functional interface format. After the request is unpacked, it is sent to the server functional interface (SFI), the database manager DBM for processing. From Figure 3.4, we can see that all the layers under the SFI have exactly the same architecture as the local processing unit with the exception that the file locking manager (FLM) is removed. The file locking manager is not needed in the server because the scheduling mechanism only allows one request for database processing at any one time. Once the DBM carried out the query/update. results are sent back to the TRAN for packing into text record form. Once the results are in text record form, the COMM will send the record through the communication link to the LOCAL CIM.

3.4 Communication

A communication protocol is used to handle the communication between the LOCAL CIM and the CIM SERVER. Three stages are identified in a communication session.

- Establishing the connection.
- Data transfer.
- Connection termination.

3.4.1 Establishing the Connection

The initial phase of communication between LOCAL CIM and CIM SERVER is a two way handshake communication. At the beginning of the session, the client (LOCAL CIM) connects with the server (CIM SERVER) through a public communication channel (UNIX SOCKET). Once a connection has been initiated, the operating system at the server side will atomically notify the client to use a private communication channel instead of using the public communication channel for further communication. The public channel will then freed for the connection with other process. Once a private channel is established, an initial handshake is performed. The client will send a connect request to the server. According to the information in the request message, the server will check whether the client has permission to use to the server. Then the server will reply with a "connection acceptance" or "reject message" to the caller. The client must send a handshake message to confirm the acceptance. After received the confirmation, the server will establish all system resource needed by this connection session.

The purpose of the handshake is to introduce the client to the server. The request message that the client sends to the server consists of these parameters: client process id., station id., priority. The acceptance message send by the server consists of the following parameters: connection time, job number.

3.4.2 Data Transfer

The data transfer for the server is nonblocking send/receive. A nonblocking mechanism is used so that the server will achieve a higher level of *concurrency*. Data transfer for the client is blocking send/receive. Service request is sent by the client and service reply is sent by the server.

3.4.3 Connection Termination

The connect session only can be activated by the client by initiating a terminate request to the server. After the server has received this message, it will send an acknowledgment to the client. Then both side de-allocate all system resources for this session.

Chapter 4

Conclusion

4.1 Cache Model

The implementation of the cache model of the class interface manager has been completed. It has been successfully integrated to the other components of the Dee programming environment [2] [14]. Currently the Dee programming environment runs on a network of SUN IPC[1] workstations under SunOs[1] Release 4.1.1 and X Windows. We have also succefully ported the Dee compiler and the linker (DINK) to a 80386[1] type's computer running Xenix[2]. Porting the entire Dee programming environment together with the browser to the 80386 is possible. It is not done because our version of Xenix does not have X Windows. Although we have not ported the Dee programming environment to Unix platform other than the IPC and 80386, however porting it to other Unix platforms should be simple, since the Dee compiler is totally written in standard C programming language with standard Unix routines. The source code of the cache model of Unix CIM is about 90K and is made up of 11 files.

Up to the present date, the standard database has about 28 standard Dee classes. A number of students is building a wider standard library for Dee.

[0]Sun, IPC and SunOs are registered trademarks of Sun Microsystems, Inc

[1]80386 is a registered trademark of Intel Corporation

[2]Xenix is a registered trademark of Microsoft-Corporation

4.1.1 Performance

The cache load-in and cache write-back time of the CIM is so quick that a user would not notice any delay. By integrating it to the Dee programming environment, we obtained quite satisfactory performance. Since the performance of the CIM is Dee code dependable, it is not possible to make a precise measurement on its performance. Instead we try to represent its performance on an average scale. Currently there are 28 standard Dee classes (excluding special classes) with a total of 262 attributes successfully compiled by Unix Dee. Base on 1,000 load-in and write-back cache operations on the current standard Dee classes, each load-in time is about 19.1 milliseconds and each write-back time is about 14.0 milliseconds. Notice that the load-in time is longer than the write-back time: this is because the load-in operation includes the time to initialize the cache manager.

4.1.2 Storage Overhead

The current database file is 61,664 bytes long. The class interface data occupies 27,093 bytes and the system overhead is 34,571 bytes. The storage overhead is about 128% of the actual data. Although the storage overhead is quite high, it is a necessary for the performance. The storage overhead of CIDB includes address spaces for pointer fields, system table and index structures.

4.2 Server Model

The implementation of the server model has been started, but is not complete. A prototype of the communication manager and its scheduler has been finished and both are giving encouraging result. The prototype performs messages transfer between the CIM SERVER and a number of clients. It also handles events scheduling. These events include constant monitoring the communication links, putting the messages into the corresponding message queue and the detection of complete messages. Design of the protocol layer (TRAN) has been finished and implementation is in progress. Other components can be easily implemented by reusing the code of the cache model.

4.3 Further Work

Since the slowest operations performing by the CIM are the cache load-in operation and the cache write back operations, therefore for further work on CIM, we focus on enhancing its performance. Although the performance of cache version CIM is quite satisfactory, one can further enhance the time spend on cache-writeback operation. In the case of a very large CIDB, the time saved by the enchanced write-back operation is substantial. The cache operation of the CIM uses a cache strategy such that the cache-loadin loads the whole database into the memory at load time. Any updates to the database would require writing the whole cache back to the disk at save time. Although it gives quite a satisfactory performance, it is possible to further minimize the write-back time by writing only the changed portion of the cache to the secondary storage. The cache memory is organized in a linked lists of memory pages and the cache memory address location is same as its corresponding file address, one can easily modify the cache manager to write only the dirty pages to the database file. This requires the cache memory manager (CMM) to mark a particular page with a dirty bit whenever a change is made to a memory location.

Further enhancement on the server model of the CIM includes, adding the capability of source code sharing in addition to the class interface sharing. These provide the users of a remote node with the ability to compile a Dee class that inherits some shared classes in a local node. All the necessary tools are already designed in the server model. To add the source code sharing mechanism, only minor modifications to the current design are needed. This involves setting up a database to store the shared source code data.

4.4 About This Project

The objective of this project is to implement a Unix version of Dee. The Dee project is headed by Dr. Peter Grogono. There are four other graduate students involved in the project. They are Benjamin Yik-Chi Cheung, Lawrence A. Hegarty, Wai Ming Wong and myself. Each of the students is responsible for the design and implementation of a specific module of the Dee compiler. Benjamin works on the linker and the browser. Lawrence works on the semantic analyser and the code generator. Wai Ming works on the parser. I work on the class interface manager. The work was finished on May of 1992 and the system is up and running on the Department of Computer Science's IPC network.

The development of the project was going smoothly in the design phase. Regular meetings were carried out every three to four weeks to discuss varies design issues and to monitor the project's progress. Informal discussions between individuals were carried out every one or two days. Matter such as interface standards between modules, data structures and the language's syntax were discussed in these meetings.

During the implementation phase, we ran in a lot of trouble with the C compiler. Benjamin reported two bugs in the GCC compiler. There is also incompatibility between the GCC compiler and the CC debuger DBX. The DBX debuger sometimes provides us with out-of-step trace information. In some cases, we have to put a printf() statement to every other line in the suspicious source in order to trace the program. All these problems make the implementation a little inconvenience, but with patience, they were all overcame.

Bibliography

[1] Benjamin Yik-Chi Cheung. CIM for PC-Dee. *Computer Program*, 1991.

[2] Benjamin Yik-Chi Cheung. Dee: An Object Oriented Programming Environment and its Implementation. Master's thesis, Concordia University, Montreal, Canada, May 1992.

[3] Joel M. Crichlow. *An Introduction to Distributed and Parallel Computing*. Prentice-Hall, Inc, 1988.

[4] Charles N. Fischer and Jr. Richard J. LeBlanc. *Crafting a Compiler*. The Benjamin/Cummings Publishing Company, Inc, 1988.

[5] Philip Gilbert. *Software Design and Development*. Science Research Association, Inc, 1983.

[6] A. Goldberg. The influence of an object-oriented language on the programming environment. In D. Barstow, H. Shrobe, and E. Sandewall, editors, *Interactive Programming Environments*, chapter 8, pages 141–174. McGraw Hill, 1984.

[7] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[8] P. Grogono. The Book of Dee. Technical Report OOP 90 3, Department of Computer Science, Concordia University, February 1990.

[9] P. Grogono. The Dee Report. Technical Report OOP 91 2, Department of Computer Science, Concordia University, January 1991.

[10] P. Grogono. Designing a Class Library. Technical Report OOP-91-3. Department of Computer Science, Concordia University. April 1991.

[11] P. Grogono. Issues in the design of an object oriented programming language. *Structured Programming*, 12(1):1-15, January 1991.

[12] P. Grogono and B. Cheung. Database Support for Browsing. Technical Report OOP 91 1, Department of Computer Science, Concordia University, January 1991.

[13] P. Grogono and B. Cheung. A semantic browser for object oriented program development. In *25th Hawaii International Conference on System Sciences*, January 1992. To appear.

[14] Lawrence A. Hegarty. Implementing the Dee System: Issues and Experiences. Master's thesis, Concordia University, Montreal, Canada, June 1992.

[15] Mathai Joseph, V.R. Prasad, and N. Natarajan. *A Multiprocessor Operating System*. Prentice-Hall, Inc. 1984.

[16] James F. Korsh and Leonard J. Garrett. *Data Structures, Algorithms, and Program Style Using C*. PWS-KENT Publishing Company, 1988.

[17] Henry F. Korth and Abraham Silberschatz. *Database System Concept*. McGraw-Hill Book Company, 1986.

[18] M. Linton. Implementing relational views of programs. In P. Henderson, editor, *Proc. ACM Software Engineering Symp. on Practical Software Development Environments*, pages 132-140. ACM, April 1984. Also published in *SIGPLAN Notices*, 19(5), May 1984.

[19] B. Meyer. *Object-oriented Software Construction*. Prentice Hall International, 1988.

[20] B. Meyer. From structured programming to object-oriented design: the road to Eiffel. *Structured Progamming*, 10(1):19-39, 1989.

[21] S. Meyers. Working with object-oriented programs: the view from the trenches is not always pretty. In *Proc. Symp. on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, pages 51–65, September 1990.

[22] M. Missikoff and M. Terranova. *The Architecture of a Relational Database Computer Known as DBMAC*. Prentice- Hall, Inc, 1983.

[23] D. Ungar. *The Design and Evaluation of a High Performance Smalltalk System*. MIT Press, 1987.

Appendix A

Class Example: String

The class *String* is a basic class for strings of ASCII characters. It inherits the classes Monoid and the class Order. The keyword *special*, appearing as the body of a method, indicates that the method is implemented by C code.

```
class String

-- Basic class for strings of ASCII characters

inherits Monoid Order

public method maxstringlen  Int
-- Return the maximum permitted length of a string
  begin
    result := 255
  end

public method = (other. String) Bool
-- Return true iff the receeiver is equal to the argument
  special

public method < (other  String): Bool
-- Return true if the receiver is lexicographically less than the
-- argument
  special

public method show · String
-- Return a string representing the string (') This is the identity
-- function for strings but is provided for compatibility
  begin
    result .= self
  end


public method zero  String
-- Return the empty string
  begin
    result  = ""
  end

public method + (other  String) String
-- Return the concatenation of the receiver and the argument
  special

public method len  Int
-- Return the length of the receiver
  special

public method stoi  Int
-- Return the integer denoted by the string  If there is no such integer,
-- raise a system exception
  special

public method stof  Float
-- Return the floating point number denoted by the string  If there is no
-- such number, raise a system exception
  special

public method ctoi· Int
-- Return the ASCII code of the first character of the receiver  System
-- exception if the receiver is the empty string
  special

public method substr (x  Int  y  Int)  String
-- Return the substring starting at character position x
-- and containing y characters with sensible defaults
  special

public method pat (s  String)  Int
-- Return the index of the first position at which the receiver (viewed
-- as a pattern) occurs in the argument  Return 0 if the receiver does not
-- occur in the argument
  special

public method rep (count  Int): String
-- Returns count copies of the receiver concatenated  For example,
-- (" ") rep(5) returns a string of 5 blanks. Returns the empty string
-- if count is zero or negative  System error 18 if the result is too
-- long
  special

public method left (width  Int)  String
-- Return the receiver left justified in a field of width characters.
  begin
    result  = self + (" ") rep(width - self.len)
  end

public method right (width  Int)  String
-- Return the receiver right justified in a field of width characters.
  begin
    result  = (" ") rep(width - self.len) + self
  end


public method print
-- test routine
-- for the first program
   special

public cons  get
-- test routine
-- for the first program.
   special

public method toByte  Byte
-- convert first characher of the string to byte.
special
```

Appendix B

Class Example: Order

This class provides protocol for any class which is totally ordered. It inherits the class Compare. Notice that the keyword *special* does not appear in the body of any methods. This is due to the fact that the body of these methods can be coded in Dee.

```
class Order

-- This class provides protocol for any class which is totally ordered.
-- A descendant must implement two of the comparison methods
-- The others will then be implemented by the code of this class.

inherits Compare

public method <  (other: Order): Bool
-- Return true iff the receiver is less than the argument under the
-- ordering of the current class.


public method =  (other: Order): Bool
-- Return true iff the receiver is equal to the argument under the
-- ordering of the current class

public method <= (other: Order): Bool
-- Return true iff the receiver is less than or equal to the argument under
 the
-- ordering of the current class.
  begin
    result := (self < other) or (self = other)
  end

public method >  (other: Order): Bool
-- Return true iff the receiver is greater than the argument under the
-- ordering of the current class.
  begin
    result := other < self
  end

public method >= (other: Order): Bool
-- Return true iff the receiver is greater than or equal to the argument un
der the
-- ordering of the current class
  begin
    result := (other < self) or (self = other)
  end

public method min (other: Order): Order
-- Return the smaller of the receiver and the argument under the ordering
-- of the current class
  begin
    if self < other
      then result := self
      else result := other
    fi
  end

public method max (other: Order): Order
-- Return the larger of the receiver and the argument under the ordering
-- of the current class
  begin
    if other < self
```

```
      then result := self
      else result := other
    fi
  end
```

Appendix C

Source Code Listing

A source code listing of the FI layer, FLM layer and the DBM layer are shown in this appendix. It consists of 10 files. The descriptions of these files are as follow:

db_def.h	contains the variable definitions of the DBM layer.
db_fun.h	contains the routine definitions of the FI layer.
db_fun.c	contains the source code of the FI layer.
db_sys.c	contains the source code of the DBM layer which maintains system initialization and high level memory management.
db_class.c	contains the source code of the DBM layer which maintains the CTN subtree.
db_attri.c	contains the source code of the DBM layer which maintains the ATN subtree.
db_attin.c	contains the source code of the DBM layer which maintains the linked list of CLN.
db_attli.c	contains the source code of the DBM layer which maintains the linked list of ALN.
db_lock.h	contains the variable definitions and the routine definitions of the FLM layer.
db_lock.c	contains the source code of the FLM layer.

```
/* Databse of the Dee Class */
/* MAy 92                                    */

#define NO_IMAGE 4


#define DB_ATTRIBUTE_TABLE_SIZE     sizeof (DbAttributeTableType)

#define DB_CLASS_TREE_NODE_SIZE     sizeof (DbClassTreeNodeType)
#define DB_ATTRIBUTE_TREE_NODE_SIZE sizeof (DbAttributeTreeNodeType)

#define DB_CLASS_LIST_NODE_SIZE     sizeof (DbClassListNodeType)
#define DB_ATTRIBUTE_LIST_NODE_SIZE sizeof (DbAttributeListNodeType)

#define FOUND 0
#define NOTFOUND 1

#define REMOVED 0
#define NOTREMOVED 1

#define DONE 0
#define ERROR 1

#define RIGHT 0
#define LEFT 1

#define FALSE 0
#define TRUE 1



/* D E F I N I T I O N   O F   D E E   D A T A B A S E ' S   N O D E */

/* binary tree node for class */
typedef struct DbClassTreeNode
{
  cimAddress Name;             /* Class Name
  cimAddress ClassDef,         /* Class Definition */
  cimAddress ClassComment,     /* Class Comment    */
  cimAddress IEUAList;         /*                  */
  cimAddress InvarList,        /* Invarient List   */

  cimAddress VariableTreePtr, /* pointer to Attribute Tree */
  cimAddress MethodTreePtr;   /* pointer to Attribute Tree */
  char       ClassDeleteFlag, /* Delete Flag */

  cimAddress Left;             /* Pointer to left subtree */
  cimAddress Right,            /* Pointer to Right subtree */
} DbClassTreeNodeType;


/*  binary tree node for attribute */
typedef struct DbAttributeTreeNode
```

```
{
  cimAddress Name,            /* Attribute Name */
  char       AttributeKind,   /* Attribute Kind  V/M */
  char       AttributeType,   /* Attribute Type */
  cimAddress Comment,         /* Attribute Comment */
  cimAddress DefinitionList,  /* Attribute Definition List */

  cimAddress Left,            /* Pointer to the left subtree */
  cimAddress Right,           /* Pointer to the right subtree */
} DbAttributeTreeNodeType,




/* Attribute list use by browser */
typedef struct DbAttributeListNode
{
  cimAddress Name,              /* Attribute Name */
  cimAddress AttributeCount,    /* Count of same attribute in current image */

  cimAddress DbClassListPtr,    /* Pointer to Class List */
  cimAddress Next,
} DbAttributeListNodeType;




/*  Class link list of attribute use by browser */
typedef struct DbClassListNode
{
  cimAddress Name,              /* Class Name */

  cimAddress AttributeTreePtr;  /* Pointer to the Attribute Tree Node */
  cimAddress Next,
} DbClassListNodeType,



/* Head Pointer of Class Root, with index as the corresponding Images */
extern cimAddress *DbClassRoot[NO_IMAGE],

/* Entry Table of the browser's data base */
extern cimAddress *AttributeTable[NO_IMAGE][27];
```

```
/*
   This function check if
   the class *strptr
   exit in the current image

   Return NOTFOUND if Class Name not found in current image,
   return FOUND if Class Name exist in current image
*/
extern int CIM_DB_Check_Class        (/* char *strptr */),



/*
   This function write the
   class record *ptr to
   the current image.

   Return DONE if writer is success,
   return ERROR if write is not success
*/
extern int CIM_DB_WriteClass         (/* cim_ClassIORecord *ptr */),



/*
   This function read the
   record *ptr in the current
   image by giveing the class
   name *strptr

   Return Done if read is success,
   return ERROR if read is not success
*/
extern int CIM_DB_Read_Class         (/* char *strptr, cim_AttributeIORecord *ptr */),




/*
   This function read a list of classe records
   which is in the current image  *ptr gives
   the head of the link list

   Return FOUND if there exist such a list,
   return NOTFOUND if it is an kempty list.
*/
extern int CIM_DB_Read_All_Class     (/* cim_ClassIORecord *ptr */),
/*
   This function return a list of class names
   which is in the current image
*/


extern ClassSlotPtr CIM_DB_Read_Class_List(),


/*
   This function delete the
   class record in the current
   by the name *strptr

   Return DONE if delete is success,
   return ERROR if delete is not success.
*/
extern int CIM_DB_Delete_Class       (/* char *strptr */);




/*
   This function write attribute
   record into the current image
   under the class name *strptr

   Retrun DONE if write is success,
   return ERROR if write is not success
*/
extern int CIM_DB_Write_Attribute    (/* char *strptr, cim_AttributeIORecord *ptr */);




/*
   This function read an attribute
   record *ptr from the current image
   with class name *strptr1 and
   attribute name *strptr2

   Return DONE if read success,
   return ERROR if read not success
*/
extern int CIM_DB_Read_Attribute     (/* char *strptr1, char *strPtr2,
                                         cim_AttributeIORecord *ptr */),




/*
   This function read a list of attribute
   record under the class name *strptr in
   the current image. *ptr gives the head
   of the link list

   Return FOUND if there exist such a list,
   return NOTFOUND if it is an kempty list.
*/
extern int CIM_DB_Read_Attribute_List(/* char *strptr, cim_IORecListType *ptr */),




/*
   This function read a list of attribute
   records under the class name *strptr1 in
   the current image which matches the partial
   attrribute name *strptr. *ptr gives the head
   of the link list

   Return FOUND if there exist such a list,
   return NOTFOUND if it is an kempty list
*/
extern int CIM_DB_Read_Class_PartAtt_List (/* char *strptr1,
                                          char *strptr2,cim_AttributeIORecord *ptr */),




/*
   This function read a list of attribute
   record which match the partial attribute
   name *strptr in the current image. *ptr
   gives the head of the link list

   Return FOUND if there exist such a list,
   return NOTFOUND if it is an kempty list
*/
extern int CIM_DB_Read_PartAtt_List (/* char *strptr, cim_IORecListType *ptr */),




/*
   This function initial the current image
*/
extern CIM_DB_Init_Image        (),



/*
```

```
   This procedure initial the locking mechanism by initializing the
   integer file descriptor
   Given id (image id) and fd (value of file descriptor)
*/
extern CIM_DB_Lock_InitId(/* int id, int fd */),




/*
   This procedure lock the specific database indicated by id,
   Type of lock is indicated by setting value of mode to
   the value READLCK or WRITELCK
   return ERROR the lock is fail
                if previously the same process is having a different
                type of lock on the indicated database and is not released.
   return DONE the lock is successfully applied
                if previously the same process do not have any
                lock on the indicated database or
                if previously the process
                have the same kind of lock on the database.
*/
extern int CIM_DB_Lock(/* int id, int mode*/),


/*
   This procedure release the lock on the specific database indicated by id
*/
extern CIM_DB_Rlse_Lock(/* int id */),



/*
   This procedure release all the lock that the current process have on all

   the databse
*/
extern CIM_DB_Rlse_All_Lock(),
#include <stdio.h>
#include <string h>
#include <stdlib h>
#include <malloc h>
#include <sys/types.h>
#include <unistd h>
#include <fcntl.h>
#include "deecim h"
#include "cim-io.h"
#include "cim-mem h"
#include "db_def.h"
#include "db_lock h"




int CIM_DB_Write_Class(ClassRec)
cim_ClassIORecord  *ClassRec,
{
    return(DbAddClass_ClassNode(DbClassRoot[CIM_mem_getActiveImage()],ClassRec)),
}




int CIM_DB_Check_Class(Name)
char *Name;
{
cimAddress Where,Prev;
int Prev_Dir,
DbClassTreeNodeType *Where_ptr,

   if (DbFindClass(DbClassRoot[CIM_mem_getActiveImage()],Name,
                                         Where,Prev,&Prev_Dir)==NOTFOUND)
      return(ERROR),
   else
      {
      Where_ptr=(DbClassTreeNodeType *)CIM_mem_address(Where),
      if (Where_ptr->ClassDeleteFlag=='N')
         return(DONE);
      else
         return(ERROR),
      }
}




int CIM_DB_Write_Attribute(Name,AttrRec)
cim_AttributeIORecord *AttrRec;
```

```
char *Name,
{
    return(DbAddAttribute_ClassName(Name,AttrRec)),
}







int CIM_DB_Delete_Class(Name)
char *Name,
{
    return(DbMarkDeleteClass(Name)),
}


/****************************************************************************************/



int CIM_DB_Read_Class(Name,Record)
char *Name,
cim_ClassIORecord *Record,
{
   DbClassTreeNodeType *Where_ptr,
   cimAddress Where,
   cimAddress Prev,
   int         Prev_Dir,

  if (DbFindClass(DbClassRoot[CIM_mem_getActiveImage()],Name,Where,
                                                        Prev,&Prev_Dir)==NOTFOUND)
     return(ERROR),
  else
     {
     Where_ptr=(DbClassTreeNodeType *)CIM_mem_address(Where),

     Record->Name=CIM_mem_getString(Where_ptr->Name),
     if (Where_ptr->ClassDef != NULL)
         Record->ClassDef=CIM_mem_getString(Where_ptr->ClassDef),
     else
         Record->ClassDef=NULL,


     if (Where_ptr->ClassComment != NULL)
         Record->ClassComment=CIM_mem_getString(Where_ptr->ClassComment),
     else
         Record->ClassComment=NULL,


     if (Where_ptr->IEUAList != NULL)
         Record->IEUAList=CIM_mem_getString(Where_ptr->IEUAList);
     else
         Record->IEUAList=NULL,


     if (Where_ptr->InvarList != NULL)
         Record->InvarList=CIM_mem_getString(Where_ptr->InvarList),
     else
         Record->InvarList=NULL,



     return(DONE),
     }
}



int CIM_DB_Read_All_Class(ioptr)
cim_ClassIORecListType **ioptr,
{
  *ioptr=NULL,
  CIM_Get_Class(&*ioptr,DbClassRoot[CIM_mem_getActiveImage()]),
  if (*ioptr==NULL)
     return (NOTFOUND),
  else
     return (FOUND),
}




ClassSlotPtr CIM_DB_Read_Class_List()
{
ClassSlotPtr ptr,
ptr=NULL,
CIM_Get_Class_List(&ptr,DbClassRoot[CIM_mem_getActiveImage()]),
return(ptr),
}





CIM_Get_Class_List(ioptr,ptr)
ClassSlotPtr *ioptr,
cimAddress *ptr,
{
DbClassTreeNodeType *ptr_ptr,
ClassSlotPtr lpt,

if (!CIM_me._checkNull(ptr))
  {
    ptr_ptr=(DbClassTreeNodeType *)CIM_mem_address(ptr),
    if (ptr_ptr!=NULL)
      {
```

```
     CIM_Get_Class_List(&*ioptr,ptr_ptr->Left),
     lpt=cim_GetClassSlotNode(CIM_mem_getActiveImage()),
     lpt->Name=CIM_mem_getString(ptr_ptr->Name),
     InsertAllClassList(&*ioptr,&lpt),
      }
    CIM_Get_Class_List(&*ioptr, ptr_ptr->Right ),
  }
}



InsertAllClassList(ioptr,lptr)
ClassSlotPtr *ioptr,
ClassSlotPtr *lptr,
{
ClassSlotPtr tmpio_ptr,


if (*ioptr==NULL)
    *ioptr=*lptr,
else
   {
   tmpio_ptr=*ioptr,
   while (tmpio_ptr->Next'=NULL)
         tmpio_ptr=tmpio_ptr->Next;
   tmpio_ptr->Next=*lptr,
   }
}






CIM_Get_Class(ioptr,ptr)
cim_ClassIORecListType **ioptr,
cimAddress *ptr,
{
DbClassTreeNodeType *ptr_ptr,
cim_ClassIORecord   *lptr,

ptr_ptr=(DbClassTreeNodeType *)CIM_mem_address(ptr),

if (ptr_ptr'=NULL)
   {
   CIM_Get_Class(&*ioptr,ptr_ptr->Left),
   if (ptr_ptr->ClassDeleteFlag == 'N')
      {
       lptr=(cim_ClassIORecord *)calloc(1,cim_CLASSIORECSIZE),
       lptr->Name=CIM_mem_getString(ptr_ptr->Name),

       if (ptr_ptr->ClassDef '= NULL)
          lptr->ClassDef=CIM_mem_getString(ptr_ptr->ClassDef),
       else
          lptr->ClassDef=NULL,
      if (ptr_ptr->ClassComment != NULL)
         lptr->ClassComment=CIM_mem_getString(ptr_ptr->ClassComment),
      else
         lptr->ClassComment=NULL,


      if (ptr_ptr->IEUAList != NULL)
         lptr->IEUAList=CIM_mem_getString(ptr_ptr->IEUAList),
      else
         lptr->IEUAList=NULL,


      if (ptr_ptr->InvarList != NULL)
         lptr->InvarList=CIM_mem_getString(ptr_ptr->InvarList),
      else
         lptr->InvarList=NULL,


      InsertClassIOList(&*ioptr,lptr),
     }

   CIM_Get_Class(&*ioptr,ptr_ptr->Right),
   }


}

InsertClassIOList(ioptr,lptr)
cim_ClassIORecListType **ioptr,
cim_ClassIORecord *lptr,
{
cim_ClassIORecListType *aptr, *tmpio_ptr,

aptr=(cim_ClassIORecListType *) calloc(1,cim_CLASSIORECLISTSIZE),
aptr->RecordIO_ptr=lptr,
aptr->Next=NULL;

if (*ioptr==NULL)
    *ioptr=aptr,
else
   {
   tmpio_ptr=*ioptr;
   while (tmpio_ptr->Next!=NULL)
         tmpio_ptr=tmpio_ptr->Next,
   tmpio_ptr->Next=aptr,
   }
}

/*****************************************************************************************/

int CIM_DB_Read_Attribute(ClassName,AttName,Record)
```

```
char *ClassName,
char *AttName,
cim_AttributeIORecord *Record,
{
  DbClassTreeNodeType *C_Where_ptr,
  DbAttributeTreeNodeType *A_Where_ptr,
  cimAddress            Where,
  cimAddress            Prev,
  int                   Prev_Dir,

  if (DbFindClass(DbClassRoot[CIM_mem_getActiveImage()],ClassName,
                                                     Where,Prev,&Prev_Dir)==NOTFOUND)
     return(ERROR),
  else
     {
      C_Where_ptr=(DbClassTreeNodeType *)CIM_mem_address(Where);
      if (C_Where_ptr->ClassDeleteFlag=='Y')
         return(ERROR),
      else
         {
          if (DbFindAttribute(C_Where_ptr->VariableTreePtr,AttName,
                                                      Where,Prev,&Prev_Dir)==NOTFOUND)
             {
              if (DbFindAttribute(C_Where_ptr->MethodTreePtr,AttName,
                                                        Where,Prev,&Prev_Dir)==NOTFOUND)
                 return(ERROR),
             }

          A_Where_ptr=(DbAttributeTreeNodeType *)CIM_mem_address(Where),
          Record->Name=CIM_mem_getString(A_Where_ptr->Name),
          Record->AttributeKind=A_Where_ptr->AttributeKind;
          Record->AttributeType=A_Where_ptr->AttributeType,

          if (A_Where_ptr->Comment != NULL)
             Record->Comment=CIM_mem_getString(A_Where_ptr->Comment),
          else
             Record->Comment=NULL,

          if (A_Where_ptr->DefinitionList != NULL)
             Record->DefinitionList=CIM_mem_getString(A_Where_ptr->DefinitionList),
          else
             Record->DefinitionList=NULL,


          return(DONE),
         }
     }
}



int CIM_DB_Read_Attribute_List(Name,ioptr)
char               *Name,
cim_IORecListType **ioptr,
{
  DbClassTreeNodeType *Where_ptr,
  cimAddress           Where;
  cimAddress           Prev;
  int                  Prev_Dir,

  *ioptr=NULL,
  if (DbFindClass(DbClassRoot[CIM_mem_getActiveImage()],
                              Name,Where,Prev,&Prev_Dir)==NOTFOUND)
     return(NOTFOUND);
  else
     {
      Where_ptr=(DbClassTreeNodeType *)CIM_mem_address(Where),
      CIM_Get_Attribute(&*ioptr,Where_ptr->VariableTreePtr),
      CIM_Get_Attribute(&*ioptr,Where_ptr->MethodTreePtr),
      if (*ioptr==NULL)
         return (NOTFOUND),
      else
         return (FOUND),
     }
}



CIM_Get_Attribute(ioptr,ptr)
cim_IORecListType **ioptr,
cimAddress         *ptr,
{
DbAttributeTreeNodeType *ptr_ptr,
cim_AttributeIORecord   *lptr,

ptr_ptr=(DbAttributeTreeNodeType *)CIM_mem_address(ptr),

if (ptr_ptr!=NULL)
   {
   CIM_Get_Attribute(&*ioptr,ptr_ptr->Left),

   lptr=(cim_AttributeIORecord *)calloc(1,cim_ATTRIBUTEIORECSIZE),
   lptr->Name=CIM_mem_getString(ptr_ptr->Name),
   lptr->AttributeKind=ptr_ptr->AttributeKind,
   lptr->AttributeType=ptr_ptr->AttributeType,

   if (ptr_ptr->Comment != NULL)
      lptr->Comment=CIM_mem_getString(ptr_ptr->Comment),
   else
      lptr->Comment=NULL;


   if (ptr_ptr->DefinitionList != NULL)
      lptr->DefinitionList=CIM_mem_getString(ptr_ptr->DefinitionList),
   else
      lptr->DefinitionList=NULL;
   lptr->ClassName=NULL,
   InsertIOList(&*ioptr,lptr),

   CIM_Get_Attribute(&*ioptr,ptr_ptr->Right),
   }
}







InsertIOList(ioptr,lptr)
cim_IORecListType **ioptr,
cim_AttributeIORecord *lptr,
{
cim_IORecListType *aptr, *tmpio_ptr,

aptr=(cim_IORecListType *)calloc(1,cim_IORECLISTSIZE),
aptr->RecordIO_ptr=lptr;
aptr->Next=NULL;

if (*ioptr==NULL)
    *ioptr=aptr,
else
   {
   tmpio_ptr=*ioptr,
   while (tmpio_ptr->Next!=NULL)
         tmpio_ptr=tmpio_ptr->Next,
   tmpio_ptr->Next=aptr,
   }
}


/*********************************************************************************/


int CIM_DB_Read_Class_PartAtt_List(ClassName,AttName,ioptr)
char *ClassName,
char *AttName,
cim_IORecListType **ioptr,
{
cimAddress           Where,
                     Prev,
                     Node,
int                  Prev_Dir,
DbClassTreeNodeType *Where_ptr,

    *ioptr=NULL,
    if (DbFindClass(DbClassRoot[CIM_mem_getActiveImage()],
                          ClassName,Where,Prev,&Prev_Dir) == NOTFOUND)
       return(NOTFOUND),
    else
       {
        Where_ptr=(DbClassTreeNodeType *)CIM_mem_address(Where),
        CIM_Get_Class_PartAtt(&*ioptr,Where_ptr->VariableTreePtr,AttName),
        CIM_Get_Class_PartAtt(&*ioptr,Where_ptr->MethodTreePtr,AttName),
        if (*ioptr'=NULL)
           return(FOUND),
        else
           return(NOTFOUND),
       }

}


CIM_Get_Class_PartAtt(ioptr,Ptr,AttName)
cimAddress *Ptr,
char *AttName,
cim_IORecListType **ioptr,
{
DbAttributeTreeNodeType *ptr_ptr,
cim_AttributeIORecord   *lp'r,
ptr_ptr=(DbAttributeTreeNodeType *)CIM_mem_address(Ptr),

if (ptr_ptr!=NULL)
   {
   CIM_Get_Class_PartAtt(&*ioptr,ptr_ptr->Left,AttName),

   if (strncmp(AttName,CIM_mem_getString(ptr_ptr->Name),strlen(AttName))==0)
      {
       lptr=(cim_AttributeIORecord *)calloc(1,cim_ATTRIBUTEIORECSIZE),
       lptr->Name=CIM_mem_getString(ptr_ptr->Name),
       lptr->AttributeKind=ptr_ptr->AttributeKind;
       lptr->AttributeType=ptr_ptr->AttributeType,

       if (ptr_ptr->Comment '= NULL)
          lptr->Comment=CIM_mem_getString(ptr_ptr->Comment),
       else
          lptr->Comment=NULL,


       if (ptr_ptr->DefinitionList '= NULL)
          lptr->DefinitionList=CIM_mem_getString(ptr_ptr->DefinitionList),
       else
          lptr->DefinitionList=NULL,

       lptr->ClassName=NULL,
       InsertIOList(&*ioptr,lptr),
      }

   CIM_Get_Class_PartAtt(&*ioptr,ptr_ptr->Right,AttName),
   }
}
```

```
/****************************************************************************************/


CIM_DB_Lock_InitId(id,fd)
int id,
int fd,
{
token[id]=AVAIL,
Image_fd[id]=fd,
}



CIM_DB_Lock(id,mode)
int id,
int mode,
{
  if (mode == READLCK)
      return(f_read_lock(id)),
  else if (mode == WRITELCK)
      return(f_excl_lock(id)),
  else
      return(ERROR),
}




CIM_DB_Rlse_Lock(id)
int id,
{
  f_unlock(id),
}



CIM_DB_Rlse_All_Lock()
{
int id,
id = 0,
while (id < NO_IMAGE)
       {
            f_unlock(id),
            id=id+1,
       }
}


/****************************************************************************************/
```

```
int CIM_DB_Read_PartAtt_list(AttName,ioptr)
char *AttName,
cim_ClassIORecListType **ioptr;
{
DbAttributeListNodeType *Head_ptr,
DbAttributeTreeNodeType *ptr_ptr,
cim_AttributeIORecord   *lptr,
int key;
*ioptr=NULL;
key=DbListTableHash(AttName),
Head_ptr=(DbAttributeListNodeType *)
          CIM_mem_address(AttributeTable[CIM_mem_getActiveImage()][key]),
      while (Head_ptr!=NULL)
            {
                 if (strncmp(AttName ,CIM_mem_getString(Head_ptr->Name),
                                                    strlen(AttName))==0)
                     CIM_DB_Get_PartAtt_List_AttList(&*ioptr,Head_ptr->DbClassListPtr),
                 Head_ptr=(DbAttributeListNodeType *)CIM_mem_address(Head_ptr->Next),
            }
         if (*ioptr!=NULL)
            return(FOUND);
         else
            return(NOTFOUND),
}

CIM_DB_Get_PartAtt_List_AttList(ioptr,Ptr)
cim_ClassIORecListType **ioptr,
cimAddress *Ptr,
{
cimAddress *AttPtr,
DbClassListNodeType     *Head_ptr,
DbAttributeTreeNodeType *ptr_ptr,
cim_AttributeIORecord   *lptr,

     Head_ptr=(DbClassListNodeType *)CIM_mem_address(Ptr),
     while (Head_ptr!=NULL)
        {
           ptr_ptr=(DbAttributeTreeNodeType *)
                     CIM_mem_address(Head_ptr->AttributeTreePtr),
           lptr=(cim_AttributeIORecord *)calloc(1,cim_ATTRIBUTEIORECSIZE),

           lptr->Name=CIM_mem_getString(ptr_ptr->Name);
           lptr->AttributeKind=ptr_ptr->AttributeKind,
           lptr->AttributeType=ptr_ptr->AttributeType,

        if (CIM_mem_getString(ptr_ptr->Comment) != NULL)
            lptr->Comment=CIM_mem_getString(ptr_ptr->Comment),
        if (CIM_mem_getString(ptr_ptr->DefinitionList) != NULL)
            lptr->DefinitionList=CIM_mem_getString(ptr_ptr->DefinitionList),

           lptr->ClassName=CIM_mem_getString(Head_ptr->Name),
          InsertClassIOList(&*ioptr,lptr);

          Head_ptr=(DbClassListNodeType *)CIM_mem_address(Head_ptr->Next);
       }
}



/*************************************************************************************/



void CIM_DB_Init_Image()
{
     DbInitImage(),
}










/*************************************************************************************/


CIM_DB_Display_Attribute(ioptr)
cim_IORecListType     **ioptr,
{
cim_AttributeIORecord *ptr;


 if (*ioptr!=NULL)
    {
     ptr=(*ioptr)->RecordIO_ptr,
     printf("%s---Attribute Name\n",ptr->Name),
     putchar(ptr->AttributeKind);
     printf("\n-----Attribute Kind\n"),
     putchar(ptr->AttributeType),
     printf("\n-------Attribute Type\n");
     if (ptr->Comment != NULL)
        printf("%s---------Attribute Comment\n",ptr->Comment);
     if (ptr->DefinitionList != NULL)
        printf("%s-----------Attribute DefinitionList\n",ptr->DefinitionList);

     if (ptr->ClassName != NULL)
```

```
        printf("%s-----------ClassName\n\n\n",ptr->ClassName),

     CIM_DB_Display_Attribute(&((*ioptr)->Next)),
    }
}


CIM_DB_Display_Class(ioptr)
cim_ClassIORecListType     **ioptr,
{
cim_ClassIORecord *ptr,


 if (*ioptr!=NULL)
    {
     ptr=(*ioptr)->RecordIO_ptr,

     printf("class Name---->%s\n",ptr->Name);

     if (ptr->ClassDef != NULL)
          printf("class Def----->%s\n",ptr->ClassDef),
        if (ptr->ClassComment != NULL)
          printf("class comment->%s\n",ptr->ClassComment),
        if (ptr->IEUAList != NULL)
          printf("IEUAList------>%s\n",ptr->IEUAList),
        if (ptr->InvarList != NULL)
          printf("InvarList----->%s\n",ptr->InvarList),
        printf("\n"),

     CIM_DB_Display_Class(&((*ioptr)->Next)),
    }
}
```

```
#include <stdio h>
#include <string h>
#include <malloc.h>
#include <math h>
#include <fcntl.h>
#include "deecim h"
#include "cim-io h"
#include "cim-util h"
#include "cim-mem h"
#include "dL_def h"

extern cimAddress *DbClassRoot[NO_IMAGE],
extern cimAddress *AttributeTable[NO_IMAGE][27];

/*
main(argc,argv)
int argc,
char *argv[],
{
int i,j,
CIM_load(),
i=1,
DbInitImage(i),
cim_save_Image(i),
printf("Count before add is %d\n",CIM_mem_getCount()),
CIM_mem_addCount(),
cim_save_Image(i),
printf("Count after add is %d\n",CIM_mem_getCount()),


printf("Here is root point to %s\n",DbClassRoot[i]),
    for (j=0,j<=25,++j)
printf("Here is table %d point to %s\n",j,AttributeTable[i][j]),

}

*/




 /*****************************************************************/
 /* Initialize the current image                                  */
 /* if the current image file exist then read and initialize the  */
 /* DbClassRoot and the AttributeTable by loading in from the file */
 /* and  if it is not exist then this routine will create the     */
 /* corresponmding image file set it's Db_ClassRoot and the       */
 /* AttributeTable point to cim's NULL                            */
 /*****************************************************************/
DbInitImage()
{
int i,
long offsetRoot, offsetTable,
cimAddress tmpRoot,tmpTable;
cimAddress *ptrRoot, *ptrTable,

if (CIM_mem_New()==1)
   {
     DbClassRoot[CIM_mem_getActiveImage()]=(cimAddress *)
                                       CIM_mem_malloc(CIM_AddressSpace),
     CIM_mem_assignOffset(DbClassRoot[CIM_mem_getActiveImage()],0),


     offsetRoot=CIM_mem_offset(DbClassRoot[CIM_mem_getActiveImage()]),
     /* set deletecount equal zero */
     CIM_mem_resetCount();
     CIM_mem_assignOffset(tmpRoot,offsetRoot),
     /* update the node address in the current image file */
     CIM_mem_updapeNodeAddress(tmpRoot);


     AttributeTable[CIM_mem_getActiveImage()][0] =
                  (cimAddress *)CIM_mem_malloc(CIM_AddressSpace),
     CIM_mem_assignOffset(AttributeTable[CIM_mem_getActiveImage()][0],0),

     offsetTable=CIM_mem_offset(AttributeTable[CIM_mem_getActiveImage()][0]),
     CIM_mem_assignOffset(tmpTable,offsetTable),
     /* update the table address in the current image file */
     CIM_mem_updapeTableAddress(tmpTable),

     for (i=1;i<=26,++i)
         {
          AttributeTable[CIM_mem_getActiveImage()][i]=
                        (cimAddress*)CIM_mem_malloc(CIM_AddressSpace),
          CIM_mem_assignOffset(AttributeTable[CIM_mem_getActiveImage()][i],0),
         }
   }
else
    {
     ptrRoot=(cimAddress *)CIM_mem_getNodeAddress(),
     /* initialize the current DbClassRoot     */
     DbClassRoot[CIM_mem_getActiveImage()]=(cimAddress *)CIM_mem_getString(ptrRoot),


     ptrTable=(cimAddress *)CIM_mem_getTableAddress(),
     /* initialize the current AttributeTable, entry 0  */
     AttributeTable[CIM_mem_getActiveImage()][0]=
                  (cimAddress *)CIM_mem_getString(ptrTable),

     offsetTable=CIM_mem_offset((cimAddress *)CIM_mem_getString(ptrTable)),
     for (i=1;i<=26,++i)
         {
         offsetTable=offsetTable+9,
         /* initialize the current AttributeTable, entry 1-25  */
         AttributeTable[CIM_mem_getActiveImage()][i]=
                      (cimAddress *)CIM_mem_Ltoaddress(offsetTable),
        }
    }
}




/* return TRUE if reorganization  of current image is necessary */
int DbImageCheckReOrg()
{
    if (CIM_mem_getCount() > 2)
       return(TRUE),
    else
       return(FALSE),
}




/* reorganization of the current image, main routine  */
void DbImageReOrg()
{
    CIM_createTempImage(),
    CIM_mem_switchImage(CIM_Temp_Image),
    DbInitImage(),
    /* return to current image */
    CIM_mem_restoreImage(),

    DbClassReOrg(DbClassRoot[CIM_mem_getActiveImage()]),   /*Class ReOrg*/

    CIM_copyImage(CIM_mem_getActiveImage(),CIM_Temp_Image),
    CIM_save(),
    CIM_load(),
    DbInitImage(),
}




/* reorganization of Class by breath first traversal */
DbClassReOrg(Class)
cimAddress *Class,
{
cim_ClassIORecord ClassRecPtr,
DbClassTreeNodeType *Class_ptr,

    Class_ptr=(DbClassTreeNodeType *)CIM_mem_address(Class),
    if (Class_ptr!=NULL)
       {
        if (Class_ptr->ClassDeleteFlag == 'N')
           {
            ClassRecPtr Name=CIM_mem_getString(Class_ptr->Name),
            ClassRecPtr ClassDef=CIM_mem_getString(Class_ptr->ClassDef),
            ClassRecPtr ClassComment=CIM_mem_getString(Class_ptr->ClassComment),
            ClassRecPtr IEUAList=CIM_mem_getString(Class_ptr->IEUAList),
            ClassRecPtr InvarList=CIM_mem_getString(Class_ptr->InvarList),

            CIM_mem_switchImage(CIM_Temp_Image);
            CIM_  _Write_Class(&ClassRecPtr),
            CIM_mem_restoreImage(),

            DbAttReOrg(ClassRecPtr Name,Class_ptr->VariableTreePtr),
            DbAttReOrg(ClassRecPtr.Name,Class_ptr->MethodTreePtr),

           }
        /* reorganize next Class in breath first sequence */
        DbClassReOrg(Class_ptr->Left),
        DbClassReOrg(Class_ptr->Right),
       }
}



/* reorgainization of Attribute by breath first traversal */
DbAttReOrg(ClassName,Att)
char *ClassName,
cimAddress *Att,
{
DbAttributeTreeNodeType *Att_ptr,
cim_AttributeIORecord AttributeRecPtr,


    Att_ptr=(DbAttributeTreeNodeType *)CIM_mem_address(Att),
    if ( Att_ptr != NULL)
       {
         AttributeRecPtr Name=CIM_mem_getString(Att_ptr->Name),
         AttributeRecPtr AttributeKind=Att_ptr->AttributeKind,
         AttributeRecPtr AttributeType=Att_ptr->AttributeType,
         AttributeRecPtr Comment=CIM_mem_getString(Att_ptr->Comment),
         AttributeRecPtr DefinitionList=CIM_mem_getString(Att_ptr->DefinitionList),

         CIM_mem_switchImage(CIM_Temp_Image),
         CIM_DB_Write_Attribute(ClassName,&AttributeRecPtr),
         CIM_mem_restoreImage(),

         DbAttReOrg(ClassName,Att_ptr->Left),
            DbAttReOrg(ClassName,Att_ptr->Right),
        }
}
```

```
/* date May 92 */


#include <stdio h>
#include <string h>
#include <stdlib.h>
#include <malloc h>
#include "deecim.h"
#include "cim-io h"
#include "cim-mem.h"
#include "db_def.h"


cimAddress *DbClassRoot[NO_IMAGE],
cimAddress *AttributeTable[NO_IMAGE][27],


/********************************************************************************/

void CopyClassRecordToNode(Node,Record)
cimAddress        *Node,
cim_ClassIORecord *Record;
{
    DbClassTreeNodeType *Class_ptr,

    Class_ptr = (DbClassTreeNodeType *)CIM_mem_address(Node),

    strcpy(CIM_mem_getString(Class_ptr->Name),Record->Name),
    if (Record->ClassDef != NULL)
        strcpy(CIM_mem_getString(Class_ptr->ClassDef),Record->ClassDef),
    if (Record->ClassComment != NULL)
        strcpy(CIM_mem_getString(Class_ptr->ClassComment),Record->ClassComment),
    if (Record->IEUAList != NULL)
        strcpy(CIM_mem_getString(Class_ptr->IEUAList),Record->IEUAList),
    if (Record->InvarList != NULL)
        strcpy(CIM_mem_getString(Class_ptr->InvarList),Record->InvarList),

    Class_ptr->ClassDeleteFlag='N',
}




void AllocateClassNode(Node,Record)
cimAddress        *Node,
cim_ClassIORecord *Record,
{
    char *buf_ptr=CIM_mem_malloc(DB_CLASS_TREE_NODE_SIZE),

    int   LName,
          LClassDef,
          LClassComment,
        LIEUAList,
        LInvarList,
   char *buf_ptrName,
        *buf_ptrClassDef,
        *buf_ptrClassComment,
        *buf_ptrIEUAList,
        *buf_ptrInvarList,
   long  offset,
         offsetName,
         offsetClassDef,
         offsetClassComment,
         offsetIEUAList,
         offsetInvarList,
   DbClassTreeNodeType *Node_ptr,



   Node_ptr=(DbClassTreeNodeType *) buf_ptr,

   LName=strlen(Record->Name);
   buf_ptrName=CIM_mem_malloc(LName+1),
   offsetName=CIM_mem_offset(buf_ptrName),
   CIM_mem_assignOffset(Node_ptr->Name,offsetName),

   if (Record->ClassDef != NULL)
      {
       LClassDef=strlen(Record->ClassDef),
       buf_ptrClassDef=CIM_mem_malloc(LClassDef+1);
       offsetClassDef=CIM_mem_offset(buf_ptrClassDef),
       CIM_mem_assignOffset(Node_ptr->ClassDef,offsetClassDef),
      }
   else
       CIM_mem_assignOffset(Node_ptr->ClassDef,0),


   if (Record->ClassComment != NULL)
      {
       LClassComment=strlen(Record->ClassComment),
       buf_ptrClassComment=CIM_mem_malloc(LClassComment+1),
       offsetClassComment=CIM_mem_offset(buf_ptrClassComment),
       CIM_mem_assignOffset(Node_ptr->ClassComment,offsetClassComment);
      }
   else
       CIM_mem_assignOffset(Node_ptr->ClassComment,0);


   if (Record->IEUAList != NULL)
      {
       LIEUAList=strlen(Record->IEUAList);
       buf_ptrIEUAList=CIM_mem_malloc(LIEUAList+1);
       offsetIEUAList=CIM_mem_offset(buf_ptrIEUAList);
       CIM_mem_assignOffset(Node_ptr->IEUAList,offsetIEUAList);
      }
   else
       CIM_mem_assignOffset(Node_ptr->IEUAList,0),
    if (Record->InvarList != NULL)
       {
        LInvarList=strlen(Record->InvarList),
        buf_ptrInvarList=CIM_mem_malloc(LInvarList+1),
        offsetInvarList=CIM_mem_offset(buf_ptrInvarList),
        CIM_mem_assignOffset(Node_ptr->InvarList,offsetInvarList),
       }
    else
        CIM_mem_assignOffset(Node_ptr->InvarList,0),



    CIM_mem_assignOffset(Node_ptr->VariableTreePtr,0),
    CIM_mem_assignOffset(Node_ptr->MethodTreePtr,0),
    CIM_mem_assignOffset(Node_ptr->Left,0);
    CIM_mem_assignOffset(Node_ptr->Right,0),

    offset=CIM_mem_offset(buf_ptr),   /* Address of the Structure Node */
    CIM_mem_assignOffset(Node,offset),
}



void ReUseNode(Node,Record)
cimAddress        *Node,
cim_ClassIORecord *Record,
{
    int      LName,
             LClassDef,
             LClassComment,
             LIEUAList,
             LInvarList;
    int      LCName,
             LCClassDef,
             LCClassComment,
             LCIEUAList,
             LCInvarList,

    long     offset,
    char    *buf_ptr,

    DbClassTreeNodeType *Node_ptr,

    Node_ptr =(DbClassTreeNodeType *)CIM_mem_address(Node),


    LName=strlen(Record->Name),

    if (Record->ClassDef != NULL)
       LClassDef=strlen(Record->ClassDef),
    else
      LClassDef=0;

    if (Record->ClassComment != NULL )
      LClassComment=strlen(Record->ClassComment);
    else
      LClassComment=0;

    if (Record->IEUAList != NULL )
      LIEUAList=strlen(Record->IEUAList);
    else
      LIEUAList=0;

    if (Record->InvarList != NULL )
      LInvarList=strlen(Record->InvarList);
    else
      LInvarList=0;



    LCName=strlen(CIM_mem_getString(Node_ptr->Name));

    if (CIM_mem_getString(Node_ptr->ClassDef) != NULL)
      LCClassDef=strlen(CIM_mem_getString(Node_ptr->ClassDef));
    else
      LCClassDef=0;

    if (CIM_mem_getString(Node_ptr->ClassComment) != NULL)
      LCClassComment=strlen(CIM_mem_getString(Node_ptr->ClassComment));
    else
      LCClassComment=0;

    if (CIM_mem_getString(Node_ptr->IEUAList) != NULL)
      LCIEUAList=strlen(CIM_mem_getString(Node_ptr->IEUAList));
    else
      LCIEUAList=0;

    if (CIM_mem_getString(Node_ptr->InvarList) != NULL)
      LCInvarList=strlen(CIM_mem_getString(Node_ptr->InvarList));
    else
      LCInvarList=0;



    if ((LCName-LName) < 0)
      {
       buf_ptr=CIM_mem_malloc(LName+1);
       offset=CIM_mem_offset(buf_ptr);
       CIM_mem_assignOffset(Node_ptr->Name,offset);
      }


    if (LClassDef==0)
       CIM_mem_assignOffset(Node_ptr->ClassDef,0);
    else if ((LCClassDef-LClassDef) < 0)
```

```
        {
         buf_ptr=CIM_mem_malloc(LClassDef+1),
         offset=CIM_mem_offset(buf_ptr),
         CIM_mem_assignOffset(Node_ptr->ClassDef,offset),
        }


    if (LClassComment ==0)
         CIM_mem_assignOffset(Node_ptr->ClassComment,0),
    else if ((LCClassComment-LClassComment) < 0)
        {
         buf_ptr=CIM_mem_malloc(LClassComment+1),
         offset=CIM_mem_offset(buf_ptr);
         CIM_mem_assignOffset(Node_ptr->ClassComment,offset),
        }


    if (LIEUAList ==0)
         CIM_mem_assignOffset(Node_ptr->IEUAList,0),
    else if ((LCIEUAList-LIEUAList) < 0)
        {
         buf_ptr=CIM_mem_malloc(LIEUAList+1),
         offset=CIM_mem_offset(buf_ptr),
         CIM_mem_assignOffset(Node_ptr->IEUAList,offset),
        }


    if (LInvarList ==0)
         CIM_mem_assignOffset(Node_ptr->InvarList,0),
    else if ((LCInvarList-LInvarList) < 0)
        {
         buf_ptr=CIM_mem_malloc(LInvarList+1),
         offset=CIM_mem offset(buf_ptr),
         CIM_mem_assignuffset(Node_ptr->InvarList,offset),
        }
}







int DbAddClass_ClassNode(Root,Record)
cimAddress        *Root,
cim_ClassIORecord *Record,
{
    DbClassTreeNodeType *Prev_ptr,
                        *Where_ptr,
    cimAddress           Where,
                         Prev,
                         Node,
    int                  Prev_Dir
    if (DbImageCheckReOrg()==TRUE)
       DbImageReOrg();
    if (DbFindClass(Root,Record->Name,Where,Prev,&Prev_Dir) == NOTFOUND)
      {
       AllocateClassNode(Node,Record);
       CopyClassRecordToNode(Node,Record);
       if (strcmp(Where,Prev)==0)
          strcpy(Root,Node);
       else
      {
       Prev_ptr=(DbClassTreeNodeType *)CIM_mem_address(Prev);
       if (Prev_Dir == LEFT)
          CIM_mem_assignOffset(Prev_ptr->Left,atol(Node));
       else
          CIM_mem_assignOffset(Prev_ptr->Right,atol(Node));
      }
       return(DONE);
      }
   else
      {
       Where_ptr=(DbClassTreeNodeType *)CIM_mem_address(Where);
       if (Where_ptr->ClassDeleteFlag == 'Y')
          {
           strcpy(Node,Where);
           ReUseNode(Node,Record);
           CopyClassRecordToNode(Node,Record);
           return(DONE);
          }
       else
           return(ERROR);
      }
}




int DbMarkDeleteClass(Name)
char *Name;
{
DbClassTreeNodeType *Where_ptr;
int                 Prev_Dir;
cimAddress          Where,
                    Prev;

if (DbFindClass(DbClassRoot[CIM_mem_getActiveImage()],
                          Name,Where,Prev,&Prev_Dir) == NOTFOUND)
    return(ERROR);
else
   {
    Where_ptr =(DbClassTreeNodeType *)CIM_mem_address(Where);
    Where_ptr->ClassDeleteFlag='Y',
    DbDeleteAttributeTree(Name,Where_ptr->VariableTreePtr),
    DbDeleteAttributeTree(Name,Where_ptr->MethodTreePtr),
    CIM_mem_assignOffset(Where_ptr->VariableTreePtr,0),
    CIM_mem_assignOffset(Where_ptr->MethodTreePtr,0),
    CIM_mem_addCount(),
    return(DONE),
   }
}









/****************************************** ******************/
/* Given a class name, where return the location of it's node */
/* prev return the predecessor of it's node and this function */
/* return true if found in the tree, false if not found       */
/************************************************************/

int DbFindClass(Root,Name,Where,Prev,Prev_Dir)
cimAddress *Root,
char       *Name,
cimAddress *Where,
cimAddress *Prev;
int        *Prev_Dir,
{

    DbClassTreeNodeType *Where_ptr,
                        *Prev_ptr;
    int                  found = NOTFOUND,

  CIM_mem_assignOffset(Prev,0),                              /*prev=NULL*/
  Prev_ptr =(DbClassTreeNodeType *)CIM_mem_address(Prev),     /*prev_ptr=NULL*/
  Where_ptr = (DbClassTreeNodeType *)CIM_mem_address(Root),  /*p_ptr=t_ptr*/

  while ((Where_ptr != NULL) && (found == NOTFOUND))
    {
      if (strcmp(Name,CIM_mem_getString(Where_ptr->Name)) < 0)
         {
           Prev_ptr=Where_ptr,
          *Prev_Dir=LEFT;
           Where_ptr = (DbClassTreeNodeType *)CIM_mem_address(Where_ptr->Left),
         }
      else if (strcmp(Name,CIM_mem_getString(Where_ptr->Name)) > 0)
         {
           Prev_ptr=Where_ptr;
           *Prev_Dir=RIGHT,
            Where_ptr = (DbClassTreeNodeType *)CIM_mem_address(Where_ptr->Right),
          }
        else if (strcmp(Name,CIM_mem_getString(Where_ptr->Name)) == 0)
          found=FOUND,
    }


    if (Prev_ptr'=NULL)
       CIM_mem_assignOffset(Prev,CIM_mem_offset(Prev_ptr));
    else
       CIM_mem_assignOffset(Prev,0),

    if (Where_ptr'=NULL)
       CIM_mem_assignOffset(Where,CIM_mem_offset(Where_ptr));
    else
       CIM_mem_assignOffset(Where,0),

    return(found),
}






/*************************************************************************
*/
void DbDisplayClassNode_ClassName(Name)
char *Name,
{
  cimAddress Where,
             Prev,
  int        Prev_Dir,

  if (DbFindClass(DbClassRoot[CIM_mem_getActiveImage()],
                              Name,Where,Prev,&Prev_Dir)==NOTFOUND)
     printf("This class not exit in the tree\n"),
  else
     {
     printf("\nStart Printing All The Class Nodes of Image %d\n",
                                        CIM_mem_getActiveImage());
     DbDisplayClassNode_ClassNode(Where),
     }
}


DbDisplayClassNode_ClassNode(Root)
cimAddress *Root,
{
    DbClassTreeNodeType *Node_ptr,

    if ('CIM_mem_checkNull(Root))
```

```
        {
        Node_ptr=(DbClassTreeNodeType *)CIM_mem_address(Root),

        DbDisplayClassNode_ClassNode(Node_ptr->Left),

        printf("class Name---->%s\n",CIM_mem_getString(Node_ptr->Name)),

        if (CIM_mem_getString(Node_ptr->ClassDef) != NULL)
           printf("class Def----->%s\n",CIM_mem_getString(Node_ptr->ClassDef)),
        if (CIM_mem_getString(Node_ptr->ClassComment) != NULL)
           printf("class comment->%s\n",CIM_mem_getString(Node_ptr->ClassComment)),
        if (CIM_mem_getString(Node_ptr->IEUAList) != NULL)
           printf("IEUAList------>%s\n",CIM_mem_getString(Node_ptr->IEUAList)),
        if (CIM_mem_getString(Node_ptr->InvarList) != NULL)
           printf("InvarList----->%s\n",CIM_mem_getString(Node_ptr->InvarList)),
        printf("Deleted------->");
        putchar(Node_ptr->ClassDeleteFlag),
        printf("\n");
        DbDisplayClassNode_ClassNode(Node_ptr->Right),
        }
}
/****************************************************************************/
```

```
/* Routines to maintain the   tabse of the Dee Class */
/* May 92                                            */

#include <stdio h>
#include <string h>
#include <stdlib h>
#include <malloc h>
#include "deecim h"
#include "cim-io h"
#include "cim-mem h"
#include "db_def h"



extern cimAddress *DbClassRoot[NO_IMAGE],
extern cimAddress *AttributeTable[NO_IMAGE][27],


void CopyAttributeRecordToNode(Node,Record)
cimAddress            *Node,
cim_AttributeIORecord *Record,
{
     DbAttributeTreeNodeType *Node_ptr,

     char *buf_ptrName,
       *buf_ptrComment,
       *buf_ptrDefinitionList,
     int    LName,
        LComment,
        LDefinitionList,
     long  offsetName,
        offsetComment,
        offsetDefinitionList,

     Node_ptr=(DbAttributeTreeNodeType *)CIM_mem_address(Node),

     LName=strlen(Record->Name),
     buf_ptrName=CIM_mem_malloc(LName+1),
     offsetName=CIM_mem_offset(buf_ptrName);
     CIM_mem_assignOffset(Node_ptr->Name,offsetName);
     strcpy(CIM_mem_getString(Node_ptr->Name),Record->Name);

     Node_ptr->AttributeKind = Record->AttributeKind,
     Node_ptr->AttributeType = Record->AttributeType,

     if (Record->Comment != NULL)
     {
      LComment=strlen(Record->Comment),
      buf_ptrComment=CIM_mem_malloc(LComment+1);
      offsetComment=CIM_mem_offset(buf_ptrComment);
      CIM_mem_assignOffset(Node_ptr->Comment,offsetComment),
      strcpy(CIM_mem_getString(Node_ptr->Comment),Record->Comment);
     }
     if (Record->DefinitionList != NULL)
     {
      LDefinitionList=strlen(Record->DefinitionList),
      buf_ptrDefinitionList=CIM_mem_malloc(LDefinitionList+1),
      offsetDefinitionList=CIM_mem_offset(buf_ptrDefinitionList),
      CIM_mem_assignOffset(Node_ptr->DefinitionList,offsetDefinitionList),
    strcpy(CIM_mem_getString(Node_ptr->DefinitionList),Record->DefinitionList),
     }
}




void AllocateAttributeNode(Node)
cimAddress           *Node;
{
     char *buf_ptr=CIM_mem_malloc(DB_ATTRIBUTE_TREE_NODE_SIZE),
     DbAttributeTreeNodeType *Node_ptr,
     long                     offset,

     Node_ptr=(DbAttributeTreeNodeType *) buf_ptr,

     CIM_mem_assignOffset(Node_ptr->Name,0),
     Node_ptr->AttributeKind = '0',
     Node_ptr->AttributeType = '0',
     CIM_mem_assignOffset(Node_ptr->Comment,0),
     CIM_mem_assignOffset(Node_ptr->DefinitionList,0),
     CIM_mem_assignOffset(Node_ptr->Left,0),
     CIM_mem_assignOffset(Node_ptr->Right,0);

     offset=CIM_mem_offset(buf_ptr),   /* Address of the Structure Node */
     CIM_mem_assignOffset(Node,offset),
}




int DbAddAttribute_ClassName(line,Record)
char                 *line,
cim_AttributeIORecord *Record,
{
    DbAttributeTreeNodeType *ClassNode_ptr,
                      *Prev_ptr;
    cimAddress    Where,
                  Prev,
    cimAddress    Fr_Node;
    int           Prev_Dir,

    strcpy(Fr_Node,"0"),
   if (DbFindClass(DbClassRoot[CIM_mem_getActiveImage()],
                              line,Where,Prev,&Prev_Dir) == NOTFOUND)
      return(ERROR);
   else
      {
       if (Record->AttributeKind=='V')
       {
        if (DbAddVariable_ClassNode(Where,Record,Fr_Node) == DONE)
           {
            DbTableUpDate(line,Record->Name,Fr_Node);
            return(DONE);
           }
        return(ERROR);
       }
       else if (Record->AttributeKind=='M')
       {
        if (DbAddMethod_ClassNode(Where,Record,Fr_Node) == DONE)
           {
            DbTableUpDate(line,Record->Name,Fr_Node);
            return(DONE);
           }
        return(ERROR);
       }
       else
        return(ERROR);
      }
}



int DbAddVariable_ClassNode(ClassNode,Record,Fr_Node)
cimAddress           *ClassNode;
cim_AttributeIORecord *Record;
cimAddress           *Fr_Node;
{
    DbClassTreeNodeType     *ClassNode_ptr;
    DbAttributeTreeNodeType *Prev_ptr;
    cimAddress Where,
            Prev;
    int        Prev_Dir;

    ClassNode_ptr =(DbClassTreeNodeType *)CIM_mem_address(ClassNode);
    if (DbFindAttribute(ClassNode_ptr->VariableTreePtr,
                    Record->Name,Where,Prev,&Prev_Dir) == NOTFOUND)
      {
     AllocateAttributeNode(Fr_Node);
     CopyAttributeRecordToNode(Fr_Node,Record);
     if (strcmp(Where,Prev)==0)
        strcpy(ClassNode_ptr->VariableTreePtr,Fr_Node);
     else
        {
         Prev_ptr=(DbAttributeTreeNodeType *)CIM_mem_address(Prev);
         if (Prev_Dir == LEFT)
```

```
            CIM_mem_assignOffset(Prev_ptr->Left,atol(Fr_Node)),
         else
            CIM_mem_assignOffset(Prev_ptr->Right,atol(Fr_Node)),
        }
     return(DONE),
      }
   else
      return(ERROR),
}




int DbAddMethod_ClassNode(ClassNode,Record,Fr_Node)
cimAddress             *ClassNode;
cim_AttributeIORecord *Record,
cimAddress             *Fr_Node,
{
    DbClassTreeNodeType     *ClassNode_ptr,
    DbAttributeTreeNodeType *Prev_ptr;
    cimAddress               Where,
                    Prev,
    int                      Prev_Dir;

    ClassNode_ptr =(DbClassTreeNodeType *)CIM_mem_address(ClassNode),

    if (DbFindAttribute(ClassNode_ptr->MethodTreePtr,
                    Record->Name,Where,Prev,&Prev_Dir) == NOTFOUND)
       {
     AllocateAttributeNode(Fr_Node,Record),
     CopyAttributeRecordToNode(Fr_Node,Record),
     if (strcmp(Where,Prev)==0)
         strcpy(ClassNode_ptr->MethodTreePtr,Fr_Node);
     else
        {
         Prev_ptr=(DbAttributeTreeNodeType *)CIM_mem_address(Prev);
         if (Prev_Dir == LEFT)
          CIM_mem_assignOffset(Prev_ptr->Left,atol(Fr_Node));
         else
          CIM_mem_assignOffset(Prev_ptr->Right,atol(Fr_Node));
        }
     return(DONE)
       }
   else
      return(ERROR),
}
```

```
/**************************************************************/
/* Given a class name, where return the location of it's node */
/* prev return the predecessor of it's node and this function */
/* return true if found in the tree, false if not found       */
/**************************************************************/

int DbFindAttribute(Root,Name,Where,Prev,Prev_Dir)
cimAddress *Root,
char       *Name,
cimAddress *Where;
cimAddress *Prev;
int        *Prev_Dir,
{

    DbAttributeTreeNodeType *Where_ptr,
                     *Prev_ptr,
    int                        found = NOTFOUND;

    CIM_mem_assignOffset(Prev,0),                          /*prev=NULL*/

    Prev_ptr =(DbAttributeTreeNodeType *)CIM_mem_address(Prev),    /*prev_ptr=NULL*/
    Where_ptr = (DbAttributeTreeNodeType *)CIM_mem_address(Root),  /*p_ptr=t_ptr*/

    while ((Where_ptr != NULL) && (found == NOTFOUND))
       {
        if (strcmp(Name,CIM_mem_getString(Where_ptr->Name)) < 0)
           {
            Prev_ptr=Where_ptr,
           *Prev_Dir=LEFT,
          Where_ptr = (DbAttributeTreeNodeType *)CIM_mem_address(Where_ptr->Left),
           }
        else if (strcmp(Name,CIM_mem_getString(Where_ptr->Name)) > 0)
           {
            Prev_ptr=Where_ptr,
           *Prev_Dir=RIGHT,
          Where_ptr = (DbAttributeTreeNodeType *)CIM_mem_address(Where_ptr->Right),
           }
        else
            found=FOUND;
       }

    if (Prev_ptr!=NULL)
```

```
      CIM_mem_assignOffset(Prev,CIM_mem_offset(Prev_ptr));
   else
      CIM_mem_assignOffset(Prev,0);

   if (Where_ptr!=NULL)
      CIM_mem assignOffset(Where,CIM_mem_offset(Where_ptr)),
   else
      CIM_mem_assignOffset(Where,0);

   return(found),
}




DbDeleteAttributeTree(Name,Root)
char        *Name,
cimAddress *Root,
{
   DbAttributeTreeNodeType *Node_ptr,

   if (!CIM_mem_checkNull(Root))
      {
    Node_ptr=(DbAttributeTreeNodeType *)CIM_mem_address(Root);
    DbDeleteAttributeTree(Name,Node_ptr->Left);
    DbTableDelete(Name,CIM_mem_getString(Node_ptr->Name)),
    DbDeleteAttributeTree(Name,Node_ptr->Right);
      }
}



/*****************************************************************************************/



void DbDisplayAttributeNode_AttHead(Root)
cimAddress *Root,
{
   DbAttributeTreeNodeType *Node_ptr,

   if (!CIM_mem_checkNull(Root))
      {
    Node_ptr=(DbAttributeTreeNodeType *)CIM_mem_address(Root);

    DbDisplayAttributeNode_AttHead(Node_ptr->Left);

    printf("Attribute Name----->%s",CIM_mem_getString(Node_ptr->Name));
    printf("Attribute Kind----->");
    putchar(Node_ptr->AttributeKind);
```

```
     printf("\n");
     printf("Attribute Type----->"),
     putchar(Node_ptr->AttributeType);
     printf("\n");
     if (CIM_mem_getString(Node_ptr->Comment) != NULL)
        printf("Attribute Comment-->%s",CIM_mem_getString(Node_ptr->Comment)),
     if (CIM_mem_getString(Node_ptr->DefinitionList) != NULL)
      printf("DefinitionList----->%s",CIM_mem_getString(Node_ptr->DefinitionList)),
     DbDisplayAttributeNode_AttHead(Node_ptr->Right);
       }

}


void DbDisplayAttributeNode_ClassNode(Root)
cimAddress *Root,
{
    DbClassTreeNodeType *Node_ptr;
    if (!CIM_mem_checkNull(Root))
       {
     Node_ptr=(DbClassTreeNodeType *)CIM_mem_address(Root),
     DbDisplayAttributeNode_AttHead(Node_ptr->VariableTreePtr),
     DbDisplayAttributeNode_AttHead(Node_ptr->MethodTreePtr),
       }
    else
     printf("Class Node not exit"),
}



void DbDisplayAttributeNode_ClassName(Name)
char *Name,
{
    cimAddress Where,
            Prev;
    int        Prev_Dir;

    if (DbFindClass(DbClassRoot[CIM_mem_getActiveImage()],
                               Name,Where,Prev,&Prev_Dir) == NOTFOUND)
       printf("Class Name not exist\n");
    else
      DbDisplayAttributeNode_ClassNode(Where),
}

/*************************************************************************************/
```

```
/* date May 92 */

/* Routines to maintain the Databse of the Dee Class */
/*                                                   */

#include <stdio h>
#include <string h>
#include <stdlib h>
#include <malloc h>
#include "deecim.h"
#include "cim-io h"
#include "cim-mem h"
#include "db_def h"


extern cimAddress *DbClassRoot[NO_IMAGE],
extern cimAddress *AttributeTable[NO_IMAGE][27],










int DbListTableHash(Name)
char *Name,
{
     int key_value,
     char key,
     key_value=(int)*Name,
     if (key_value==(int)'_')
        return(26),
     if (key_value>=97 && (key_value<=122))
        return(key_value-(int)'a'),
     else if (key_value>=65 && (key_value<=90))
        return(key_value-(int)'A'),
     else
        return(99),
}







int DbFindAttributeListHead(Head,Name,Where,Prev)
cimAddress *Head;
char       *Name;
cimAddress *Where,
cimAddress *Prev,
{
    DbAttributeListNodeType *Where_ptr, *Prev_ptr,
    int quit=FALSE,
    int found = NOTFOUND,

    CIM_mem_assignOffset(Prev,0);
    Prev_ptr =(DbAttributeListNodeType *)CIM_mem_address(Prev),
    Where_ptr = (DbAttributeListNodeType *)CIM_mem_address(Head),  /*p_ptr = t_ptr*/
    while ((Where_ptr != NULL) && (found == NOTFOUND) && (quit==FALSE))
       {
        if (strcmp(Name,CIM_mem_getString(Where_ptr->Name)) > 0)
           {
            Prev_ptr=Where_ptr;
          Where_ptr = (DbAttributeListNodeType *)CIM_mem_address(Where_ptr->Next),
           }
        else if (strcmp(Name,CIM_mem_getString(Where_ptr->Name)) == 0)
           found=FOUND;
        else
           quit=TRUE;
       }

     if (Prev_ptr!=NULL)
        CIM_mem_assignOffset(Prev,CIM_mem_offset(Prev_ptr)),
     else
        CIM_mem_assignOffset(Prev,0),
     if (Where_ptr!=NULL)
        CIM_mem_assignOffset(Where,CIM_mem_offset(Where_ptr)),
     else
        CIM_mem_assignOffset(Where,0),


     return(found),
}




void DbAddAttributeListHead(line)
char *line;
{
     DbAttributeListNodeType *Prev_ptr,
                    *Node_ptr,
                    *Where_ptr;
     cimAddress Where;
     cimAddress Prev;
     cimAddress Node;
     int        key,
     long       count,


    key=DbListTableHash(line),
    if (DbFindAttributeListHead(AttributeTable[
                     CIM_mem_getActiveImage()][key],line,Where,Prev) == NOTFOUND)
```

```
      {
       AllocateAttributeListHead(Node);
       CopyLineToAttributeListHead(Node,line),

       Node_ptr=(DbAttributeListNodeType *)CIM_mem_address(Node);
       CIM_mem_assignOffset(Node_ptr->AttributeCount,1),

       Prev_ptr=(DbAttributeListNodeType *)CIM_mem_address(Prev),
       if (Prev_ptr == NULL)
      {
     CIM_mem_assignOffset(AttributeTable[CIM_mem_getActiveImage()][key],atol(Node));
       Node_ptr=(DbAttributeListNodeType *)CIM_mem_address(Node);
       CIM_mem_assignOffset(Node_ptr->Next,atol(Where)),
      }
       else
      {
       CIM_mem_assignOffset(Prev_ptr->Next,atol(Node)),
       CIM_mem_assignOffset(Node_ptr->Next,atol(Where));
      }
      }
   else
      {
       Where_ptr=(DbAttributeListNodeType *)CIM_mem_address(Where);
       count=atol(Where_ptr->AttributeCount)+1,
       CIM_mem_assignOffset(Where_ptr->AttributeCount,count),
      }
}




AllocateAttributeListHead(Node)
cimAddress *Node,
{
    char *buf_ptr=CIM_mem_malloc(DB_ATTRIBUTE_LIST_NODE_SIZE);
    DbAttributeListNodeType *ptr;
    long  offset,

    ptr=(DbAttributeListNodeType *) buf_ptr;

    CIM_mem_assignOffset(ptr->Name,0),
    CIM_mem_assignOffset(ptr->AttributeCount,0);
    CIM_mem_assignOffset(ptr->Next,0);
    CIM_mem_assignOffset(ptr->DbClassListPtr,0),

    offset=CIM_mem_offset(buf_ptr);   /* Address of the Structure Node */
    CIM_mem_assignOffset(Node,offset);

}
```

```
CopyLineToAttributeListHead(Node,line)
cimAddress *Node,
char       *line,
{
    DbAttributeListNodeType *tmp1_ptr,
    char *buf_ptrName,
    int   LName;
    long  offsetName;

    tmp1_ptr = (DbAttributeListNodeType *)CIM_mem_address(Node),

    LName=strlen(line);
    buf_ptrName=CIM_mem_malloc(LName+1),
    offsetName=CIM_mem_offset(buf_ptrName);
    CIM_mem_assignOffset(tmp1_ptr->Name,offsetName),
    strcpy(CIM_mem_getString(tmp1_ptr->Name),line),
}




int DbDeleteAttributeListHead(line)
char *line;
{
DbAttributeListNodeType *Prev_ptr,
                *Where_ptr;
cimAddress Where;
cimAddress Prev,
cimAddress Node;
int        key;
long       count,


   key=DbListTableHash(line),
   if (DbFindAttributeListHead(AttributeTable[
                    CIM_mem_getActiveImage()][key],line,Where,Prev) == NOTFOUND)
      return(ERROR),
   else
     {
      Where_ptr=(DbAttributeListNodeType *)CIM_mem_address(Where),
      Prev_ptr=(DbAttributeListNodeType *)CIM_mem_address(Prev),
        count=atol(Where_ptr->AttributeCount)-1,
        if (count != 0)
         CIM_mem_assignOffset(Where_ptr->AttributeCount,count),
        else
        {
        if (Prev_ptr==NULL)
         CIM_mem_assignOffset(AttributeTable[
                    CIM_mem_getActiveImage()][key],atol(Where_ptr->Next));
        else
          {
           Prev_ptr=(DbAttributeListNodeType *)CIM_mem_address(Prev);
           CIM_mem_assignOffset(Prev_ptr->Next,atol(Where_ptr->Next)),
          }
        }
        return(DONE),
       }
}


/*************************************************************************************/


DbDisplayAttributeListHead(Head)
cimAddress *Head,
{
    DbAttributeListNodeType *Head_ptr,

    Head_ptr=(DbAttributeListNodeType *)CIM_mem_address(Head),
    if (Head_ptr==NULL)
        printf("empty list\n\n\n'),
    else
       {
        while (Head_ptr!=NULL)
            {
          printf("Attribute Name--------->%s\n",CIM_mem_getString(Head_ptr->Name)),
            printf("Attribute Count------->|%s|\ ",Head_ptr->AttributeCount),
            DbDisplayAttributeListNode(Head_ptr->DbClassListPtr);
          Head_ptr=(DbAttributeListNodeType *)CIM_mem_address(Head_ptr->Next).
            }
        printf("\n\n\n"),
       }

}
```

```
/* date May 92 */

/* Routines to maintain the Databse of the Dee Class */
/*                                                   */

#include <stdio h>
#include <string.h>
#include <stdlib.h>
#include <malloc h>
#include "deecim.h"
#include "cim-io.h"
#include "cim-mem.h"
#include "db_def.h"



extern cimAddress *AttributeTable[NO_IMAGE][27],
extern cimAddress *DbClassRoot[NO_IMAGE],







int DbFindAttributeListNode(Head,Name,Where,Prev)
cimAddress  *Head,
char        *Name;
cimAddress  *Where,
cimAddress  *Prev,
{
    DbClassListNodeType *Where_ptr,
               *Prev_ptr,
    int quit=FALSE;
    int found = NOTFOUND,

    CIM_mem_assignOffset(Prev,0);
    Prev_ptr =(DbClassListNodeType *)CIM_mem_address(Prev),
    Where_ptr = (DbClassListNodeType *)CIM_mem_address(Head),  /*p_ptr=t_ptr*/

    while ((Where_ptr != NULL) && (found == NOTFOUND) && (quit==FALSE))
       {
        if (strcmp(Name,CIM_mem_getString(Where_ptr->Name)) > 0)
           {
            Prev_ptr=Where_ptr,
          Where_ptr = (DbClassListNodeType *)CIM_mem_address(Where_ptr->Next);
           }
        else if (strcmp(Name,CIM_mem_getString(Where_ptr->Name)) == 0)
           found=FOUND;
        else
           quit=TRUE;
       }
    if (Prev_ptr!=NULL)
       CIM_mem_assignOffset(Prev,CIM_mem_offset(Prev_ptr));
    else
       CIM_mem_assignOffset(Prev,0);

    if (Where_ptr!=NULL)
       CIM_mem_assignOffset(Where,CIM_mem_offset(Where_ptr));
    else
       CIM_mem_assignOffset(Where,0);

    return(found);
}







int DbAddAttributeListNode(ClassName,AttName,All_Node)
char       *ClassName,  /*Classs Name*/
char       *AttName,  /*Attribute Name*/
cimAddress *All_Node,
{
    DbClassListNodeType      *Prev_ptr,
                     *Where_ptr,
                     *Node_ptr,
    DbAttributeListNodeType *Head_ptr,
    cimAddress                 WhereAtt,
    cimAddress                 Where;
    cimAddress                 Prev;
    cimAddress                 Node;
    int                        key,



    key=DbListTableHash(AttName),
    if (DbFindAttributeListHead(AttributeTable[CIM_mem_getActiveImage()][
                                key],AttName,WhereAtt,Prev) == NOTFOUND)
       return(ERROR),
    else
       {
     Head_ptr=(DbAttributeListNodeType *)CIM_mem_address(WhereAtt);
   if (DbFindAttributeListNode(Head_ptr->DbClassListPtr,
                                    ClassName,Where,Prev) == NOTFOUND)
        {
         AllocateAttributeListNode(Node);
         CopyLineToAttributeListNode(Node,ClassName,All_Node);
         Node_ptr=(DbClassListNodeType *)CIM_mem_address(Node);
         Prev_ptr=(DbClassListNodeType *)CIM_mem_address(Prev);
         if (Prev_ptr == NULL)
```

```
            {
           CIM_mem_assignOffset(Head_ptr->DbClassListPtr,atol(Node)),
           Node_ptr=(DbClassListNodeType *)CIM_mem_address(Node),
           CIM_mem_assignOffset(Node_ptr->Next,atol(Where)),
            }
          else
            {
           CIM_mem_assignOffset(Prev_ptr->Next,atol(Node)),
           CIM_mem_assignOffset(Node_ptr->Next,atol(Where)),
            }
          return(DONE),
         }
      else
         return(ERROR),
        }
}




AllocateAttributeListNode(Node)
cimAddress *Node,
{
    DbClassListNodeType *ptr,
    char *buf_ptr=CIM_mem_malloc(DB_CLASS_LIST_NODE_SIZE),
    long  offset;

    ptr=(DbClassListNodeType *) buf_ptr,

    CIM_mem_assignOffset(ptr->Name,0),
    CIM_mem_assignOffset(ptr->AttributeTreePtr,0),
    CIM_mem_assignOffset(ptr->Next,0),

    offset=CIM_mem_offset(buf_ptr),   /* Address of the Structure Node */
    CIM_mem_assignOffset(Node,offset),
}




CopyLineToAttributeListNode(Node,Name,Allocd_Node)
cimAddress *Node,
char       *Name;
cimAddress *Allocd_Node;
{
   DbClassListNodeType *tmp1_ptr;
   char                *buf_ptrName,
   int                  LName,
   long                 offsetName,

   tmp1_ptr = (DbClassListNodeType *)CIM_mem_address(Node),

   LName=strlen(Name);
   buf_ptrName=CIM_mem_malloc(LName+1),
   offsetName=CIM_mem_offset(buf_ptrName);
   CIM_mem_assignOffset(tmp1_ptr->Name,offsetName),
   strcpy(CIM_mem_getString(tmp1_ptr->Name),Name),

   strcpy(tmp1_ptr->AttributeTreePtr,Allocd_Node),
}






int DbDeleteAttributeListNode(ClassName,AttName)
char        *ClassName,
char        *AttName,
{
    DbClassListNodeType      *Prev_ptr,
                   *Where_ptr,
    DbAttributeListNodeType *Head_ptr,
    cimAddress Where,
    cimAddress Prev,
    cimAddress Node,
    cimAddress Head,
    int        key,


    key=DbListTableHash(AttName);
    if (DbFindAttributeListHead(AttributeTable[
                      CIM_mem_getActiveImage()][key],AttName,Where,Prev)==NOTFOUND)
       return(ERROR),
    else
       {
     Head_ptr=(DbAttributeListNodeType *)CIM_mem_address(Where),
   if (DbFindAttributeListNode(Head_ptr->DbClassListPtr,
                                    ClassName,Where,Prev) ==NOTFOUND)
        return(ERROR);
     else
        {
         Where_ptr=(DbClassListNodeType *)CIM_mem_address(Where);
         Prev_ptr=(DbClassListNodeType *)CIM_mem_address(Prev);
         if (Prev_ptr==NULL)
          CIM_mem_assignOffset(Head_ptr->DbClassListPtr,atol(Where_ptr->Next));
         else
            {
          Prev_ptr=(DbClassListNodeType *)CIM_mem_address(Prev),
          CIM_mem_assignOffset(Prev_ptr->Next,atol(Where_ptr->Next)),
            }
         return(DONE),
        }
       }
}




int DbTableUpDate(ClassName,AttName,Node)
char       *ClassName;
char       *AttName;
cimAddress *Node,
{

    DbAddAttributeListHead(AttName),
    if (DbAddAttributeListNode(ClassName,AttName,Node)==ERROR)
       {
     DbDeleteAttributeListHead(AttName),
     return(ERROR),
       }
    else
     return(DONE),
}




int DbTableDelete(ClassName,AttName)
char *ClassName,
char *AttName;
{
    if (DbDeleteAttributeListNode(ClassName,AttName)==DONE)
       {
       DbDeleteAttributeListHead(AttName)·
       return(DONE),
       }
    else
       return(ERROR),
}




/*************************************************************************************/
```

```
void DbDisplayAttributeListNode(Head)
cimAddress *Head,
{
   DbClassListNodeType      *Head_ptr,
   DbAttributeTreeNodeType *ptr,

   Head_ptr=(DbClassListNodeType *)CIM_mem_address(Head);
   if (Head_ptr==NULL)
    printf("empty list\n\n\n");
   else
      {
    while (Head_ptr!=NULL)
     {
      printf("Class Name-------->|%s|\n",CIM_mem_getString(Head_ptr->Name));
    ptr=(DbAttributeTreeNodeType *)CIM_mem_address(Head_ptr->AttributeTreePtr);
      printf("%s---Attribute Name\n",CIM_mem_getString(ptr->Name));
      putchar(ptr->AttributeKind),
      printf("\n-----Attribute Kind\n"),
      putchar(ptr->AttributeType),
      printf("\n-------Attribute Type\n");
      if (CIM_mem_getString(ptr->Comment) != NULL)
        printf("%s---------Attribute Comment\n",CIM_mem_getString(ptr->Comment));
      if (CIM_mem_getString(ptr->DefinitionList) != NULL)
        printf("%s-----------Attribute DefinitionList\n\n\n",
                       CIM_mem_getString(ptr->DefinitionList));
      Head_ptr=(DbClassListNodeType *)CIM_mem_address(Head_ptr->Next),
    printf("**********************************************\n");
     }
    printf("\n\n\n"),

  }
}
```

```
#define READLCK 1
#define WRITELCK 2
#define AVAIL 0
#define RD 1
#define WR 2

struct flock arg[NO_IMAGE];  /* This need to be goble inorder to work */
int Image_fd[NO_IMAGE],

int token[NO_IMAGE],
#include <stdio.h>
#include <string.h>
#include <stdlib h>
#include <malloc.h>
#include <sys/types h>
#include <unistd.h>
#include <fcntl.h>
#include "deecim.h"
#include "cim-io.h"
#include "cim-mem.h"
#include "db_def.h"
#include "db_lock h"


int f_excl_lock(id)
int id,
{

  if ((token[id] == AVAIL) || (token[id] == WR))
     {
      arg[id] l_type=F_WRLCK;
      arg[id] l_whence=SEEK_SET,
      arg[id] l_start=0;
      arg[id].l_len=0,
      while ((fcntl(Image_fd[id],F_SETLK,&arg[id]))'=0)
          {
         /*  printf("%d completing for a exclsive lock \n",Image_fd[id]); */
          }
      token[id]=WR,
      return(DONE),
     }
   else
     return(ERROR),
}


int f_read_lock(id)
int id;
{
  if ((token[id] == AVAIL) || (token[id] == RD))
     {
      arg[id] l_type=F_RDLCK,
      while ((fcntl(Image_fd[id],F_SETLK,&arg[id]))'=0)
        {
/*              printf("%d completing for a read lock.\n",Image_fd[id]);
*/
        }
      token[id]=RD;
      return(DONE);
     }
   else
     return(ERROR);
}
```

```
f_unlock(id)
int id;
{
  if (token[id] != AVAIL)
     {
      arg[id].l_type=F_UNLCK,
      fcntl(Image_fd[id],F_SETLK,&arg[id]);
      token[id]=AVAIL;
     }
}
```