INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

DESIGN OF A FRAMEWORK FOR DATABASE INDEXES

ASHRAF GAFFAR

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

AUGUST 2001

Abstract

Design of a Framework for Database Indexes

Ashraf Gaffar

Database system performance depends greatly on the performance of the indexes used to lookup and update the database, therefore it is important to have efficient indexes to the database. Specialized application indexes developed by experts have "specialized source code" for each kind of database application. The time and cost to develop an index specific to the kind of application could be very high, making it unaffordable or even unavailable in many cases. Object–oriented framework technology has been used to produce index frameworks that can be applied to develop indexes, reducing the development cost. An index framework allows one to adapt to different key / data types, different queries and different access methods.

In this thesis, we focus on balanced tree indexes, and develop a framework in the style of the STL. We focus on the early stages of analysis, architecture, and design in an object-oriented methodology in order to design the framework. We achieve a modular framework with decoupled modules for data, data references, containers, indexes, iterators, and algorithms. This allows for developing new applications by replacing some of these modules and reflecting the changes from one model to the next, without affecting the other modules. This would result in easier developing process with less steep learning curve, and produces applications that have their own "specialized" architecture, design and source code.

Acknowledgements

I would like to express my sincere thanks to my supervisor, Dr. Gregory Butler, for his patience and valuable guidance. I benefited a lot from his deep and wide range of knowledge and experience, I enjoyed discussions with him, and I learned how to be exact and brief.

I would like to thank my parents for their sustained encouragement, which meant everything to me.

I would like to express my gratitude to my wife, who sacrificed a lot and who was there when I needed her.

To my father,

Sometimes feelings are much, much stronger than words.

Contents

LIST OF TABLES

LIST OF FIGURES

Chapter 1 Introduction

A database is not about storing away archive data; it is more about searching and updating live, dynamic data. Database indexes are the search engines for the database, and therefore constitute an important part of any database management system. They are a means for the database management system to deal with the data. From the application perspective, however, the index is invisible.

1.1 The problem

An efficient index is a fundamental requirement for good performance in a database. An accurate, fast index will result in a good quality, quick-responding database. An investment in an efficient index to the database is always a good idea. That is why in many commercial systems, a specialized handcrafted index is the classical choice to achieve the maximum efficiency possible. A specialized index is an index that supports a specific database application in a specific domain using predetermined structures and access methods as well as data types and queries. A specialized index for each new application is generally better for code efficiency and performance. The tradeoff is a lot of development time, and cost, normally affordable only by large corporations. Another choice is to make use of framework technology to develop a framework for a family of indexes, and reuse it to develop different indexes for different applications. This largely reduces the cost of providing a new index. These frameworks produce an application that is less expensive in terms of cost and time investments. Sometimes, however, frameworks provide applications that have less efficient performance than a specialized application. J. M. Hellerstein explains in [HNP95] that the use

of their framework for database indexes does not always provide an efficient lookup.

Early frameworks exist that apply this idea. They generate flexible code with some hot spots that can be easily adjusted to develop different applications but they result in a generic source code that is often complex and not easy to read or modify; characterized by many as having a steep learning curve. The reason is that in one source code, the framework is meant to satisfy all the possible needs of the future members of a family of application at one time. We can see this methodology as concentrating the complexity of future expansions in one stage; the source code. That is one reason why the resultant source code is often less user-friendly than a well-written specialized code. Cost-wise, however, it is a very good alternative.

In some cases. as part of their evolutionary lifecycle, frameworks themselves need some maintenance work. A typical case of modification is by adding new features to the framework to serve a larger family of applications than the original framework was meant to serve. That usually means adding further code even to the cold spots, or frozen spots in the source code, which were not meant to be touched in the original framework. This should lead to another framework that covers more applications than the original one. As a side effect, however, it also often leads to an even more generic code that is bigger and less user-friendly. The reason is the same, namely concentrating the complexity of evolution in one stage; the source code. As this trend progresses, the framework, going in its normal lifecycle, grows bulkier and less user-friendly with more patches added to it. In the end, time comes when it is necessary to recycle some ideas from this framework and use them to develop a new framework to replace it.

The Generalized Index Search Tree, *GiST* [HNP95] is an example. GiST is an existing framework of a generalized index system. It has friendly hot spots that can be adapted to different key types and access methods. The

rest of the source code, however, is meant to be black box so it is not user friendly. GiST tried to address all possibilities in the same piece of code, so it ended up with a large source code that is complex and not easy to follow or comprehend, especially when there is no design documents other than the few pages explaining how to adapt the hot spots. The source code itself, largely influenced by the C programming language, has poor object-oriented style. Despite the efforts to cover all cases, the original GiST did not cover the similarity search trees so it was added later to it [Aok98] as part of the framework evolution cycle. This, however, made the source code even more unreadable as the modification was spread all over the original code. Another addition to visually fine-tune the access method, the amdb [SKH99] was also added later on. This made GiST more useful, but it almost doubled the size of the already large code. In general, we can see that the additions to the framework, albeit useful, had disadvantages with them.

As the framework development methodologies improve, these problems are being recognized and addressed. The load on the source code should be diffused; i.e. the complexity of expansion should be spread to the earlier stages of framework development, following the motto *the earlier the better*. This would help getting closer to the goal of producing frameworks that achieve the advantages of both worlds: the specialized code with efficiency, understandability, and maintainability, and the framework-reusable code with its inherited advantages of efficient time and cost utilization. On the evolutionary side, being able to modify the early stages of the framework, like its architecture and design as well as the code, in a constant process, could mean a better evolution as the framework can be easily redeveloped instead of recycled.

1.2 Our Work

Frameworks have a good potential for improving the process of developing good quality software. They combine the experience of expert

programmers (most likely not domain experts) with that of domain experts (most likely not expert programmers) into a mass production environment for applications. They allow for producing several applications using virtually the same time, money and experience investment, thus the cost per application is cheaper as the total development cost is divided between several applications.

While mass production greatly helps make the application development affordable, it limits the freedom of choice a developer has [BCC+02]. Ford, the father of mass production technique in automobile industry, reduced the cost per car several folds using early, ad hoc mass production methods. Nonetheless, they had their concept of freedom of choice as *you can have any color you want in your new car, as long as it is black.* As the mass production concept was researched and improved, it now has better options, better quality, and better products that rival the custom-made ones at a fraction of the cost per application.

1.2.1 The Domain

Butler is leading a research group in framework methodologies and development for scientific applications. It is aimed at adopting and improving frameworks as a technique to produce good quality software. He is interested in framework development and evolution methodologies that make for better frameworks.

A better framework would:

-Produce good quality software (by combining effort of domain experts with that of expert programmers)

-Allow non-experts to easily develop professional application (most of experience is already in the framework)

-Promote code reuse to reduce time and cost and enhance quality.

-Produce flexible applications that are user-friendly and modification-friendly, not only in final stage (source code) but rather in all development stages.

-Be user-friendly and modification-friendly with well-defined development, documentation, application and evolution lifecycle.

On the domain front, Butler sees that "*The research on software methodology in an academic setting needs a concrete case study in order to evaluate the methodology and models*" [BCC+02].

He is interested in the database domain as a case study to validate the methodology. This resulted in the ongoing development of the Know-It-All (KIA) project, a "*framework for DBMS that support a variety of data models of data and knowledge, the integration of different paradigms and heterogeneous databases*" [BCC+02].

The KIA project involves multiple subframeworks that are integrated together in an adaptable DBMS context. It started by supporting the traditional relational database model, and it is expanding to support other types of database applications using different data models. Eventually, it will be applied to advanced applications in bioinformatics.

1.2.2 The Programming Language

The language of the source code is essential in the development process. In the end, the framework needs to be implemented using a programming language. Designing for the interface rather than for the implementation is always a good choice to follow, making the design implementation independent which allows for more freedom in the design. The programming languages are nonetheless more than just implementations. Concepts and features of some programming languages like inheritance and polymorphism can and should be reflected on the design stages to improve the quality. In other words, selecting a suitable programming language to do the job can certainly affect the quality of the software. Dr. Butler selects the C++ as one of the languages to write the implementations of different subframeworks of the KIA project.

As Stroustrup [Str94] quotes Alexander Stepanov summarizing the experience of writing and using a major library of data structures and algorithms:

> "*C++ is a powerful enough language – the first such language in our experience – to allow the construction of generic programming components that combine mathematical precision, beauty, and abstractness with the efficiency of non-generic hand-crafted code.*"

It is worth mentioning that this *major library of data structures and algorithms* mentioned by Stroustrup was the STL, designed by Alexander Stepanov and Meng Lee. The ANSI/ISO committee later recognized it as part of the standard library. STL is a library of high quality and efficiency with great emphasis on code reuse, certainly good credentials influenced by the language of implementation of that library, the C++.
The ever-evolving programming languages are introducing new features to improve their capabilities. The C++, one of the popular languages is no exception. The recent adoption of an ANSI/ISO standard for the C++ language with the STL library as part of the standard library has many advantages for the programming community. STL makes extensive use of templates, and favoring them over the traditional inheritance, casting and void* has brought advantages with it regarding code modularity and performance efficiency.

Templates

Collins dictionary defines template as a piece of metal or plastic cut into a particular shape and is used to reproduce the same shape many times. Webster's defines it as a mold used as a guide to the form of a piece being made. While this concept can apply to frameworks themselves, being seen as templates for design ruse, templates can also be adopted to the lower stages like source code, allowing for code reuse. Instead of writing the

same code several times with several similar pieces, we can factor out the similarity of these pieces into one *template*, and write the code once only using this template. Later we can use the same code many times by replacing the template with actual pieces of the same shape. This saves on the amount of code to be written (code reuse) and improves run-time performance (efficiency).

As Stroustrup [Str94] explains

"The template mechanism is completely a compile-time and link-time mechanism. No part of the template mechanism needs run-time support".

1.3 Contribution of the Thesis

This thesis introduces a design of a generalized index framework, a subframework of the KIA project. The generalized framework is capable of producing tree-based indexes that are adaptable to different data / key types, different queries, and different database application domains. We produce several models of the index framework; analysis, architecture, design and interface, and we apply them to different index applications and show that the development of each of these applications is reflected in all models. Developing a new application using this framework starts at the earliest stage; the requirements analysis, by reusing the existing analysis to determine the requirements for the new application. This will determine the necessary changes or replacement in the system architecture. We obtain the new architecture from an existing one by applying the necessary changes mapped from the analysis. The new architecture will help us identify the necessary changes in the design. We map these changes to the corresponding design models (structural and behavioral) and obtain a new design. As we get a new design, it will help us map the changes from it to the interface.

To follow these steps, we build a modular framework by applying the STL modularity concept in the analysis, architecture, design and interface stages. We develop a new set of models for an index in different

application domains; linearly ordered domain, general domain, and eventually similarity search domain. This will produce the reusable models we need for all the development stages of the framework, not just for the source code. Using coherent, decoupled building blocks allows us to locate and replace certain blocks in each model to obtain a new model with no major impact on the rest of the model. This makes modifications limited to specific regions of the system. A wealth of off-the-shelf STL modules can be used in the replacement process. The framework also allows for introducing new compatible modules as needed.

In the end we can have a new index (and thus a new source code) for each new application. Each code would be a pseudo-specialized source code that is more adjusted to its specific application than one generic code meant to serve all the possible applications.

1.4 Layout of the Thesis

Chapter 1 gives an introduction with an overview of the work of the research group. Chapter 2 presents a development environment analysis of the STL modularization concept and how to adopt its building blocks approach to decouple system modules. Chapter 3 gives an example of database indexes. Chapter 4 provides a context analysis of using an index system in a DBMS environment, and the relation between an index and both its upper level user; the application, and its lower level server; the database. Chapter 5 includes the design rationale and the general analysis, architecture, and design of the index system. We then provide examples of adapted systems for each of linearly ordered domain (chapter 6), general domain (chapter 7), and similarity search database (chapter 8). Chapter 9 gives a summary and future work.

We assume the reader is familiar with the Unified Modeling Language (UML) [BRI99], and the concepts of object-oriented design [Bch94].

Chapter 2 The STL

The Standard Template Library, STL, is not just another C++ library among many libraries available today; it is a rather important addition to the C++ programming community. One reason is in the word "Standard". The ANSI/ISO standard committee accepted the STL as part of the C++ standard library (ISO/IEC 1998). It is "A particularly carefully constructed library", [Bry00]. Another technical reason is in the word "Template". The STL is using the template mechanism of the C++ language and extends it to new dimensions by using interoperable components concept. As Matthew Austern [Aus99] states, "*Computer programming is largely a matter of algorithms and data structures*". The STL is based on separating algorithms from data structures as two different types of generic building blocks. It also introduces other generic building blocks (e.g. iterators, functors, container adaptors) to work as adaptors or glue to allow for building a large system using a combination of these blocks. This emphasizes code reuse. STL allows for new building blocks to be written and put to work with the existing ones, thus emphasizing flexibility and extendibility. This makes the STL particularly adaptive to different programming contexts including algorithms, data structures, and data types.

Part 2.1 introduces some of these building blocks with brief examples. Part 2.2 shows how to connect them together to build a useful system.

2.1 STL Building Blocks

The STL building blocks are meant to be largely independent of each other. They have different types, and a block of one type can connect and work correctly with other blocks of the same- or different types, giving an endless number of possible correct combinations.

This section introduces some of these blocks with brief examples to demonstrate their context.

2.1.1 Containers

A container is a common data structure that stores a group of similar objects, each of which can be a primitive data type, a class object, or –as in the database domain- a data record.

The container manages its own objects: their storage (see Allocators) and access (see Iterators). The stored objects belong to the container and are accessed through it. Each container provides a set of public type members, data members and member functions to provide information, and facilitate the access to its elements. Different container types have different ways of managing their objects. In other words they offer different sets of functionality to deal with their objects. Storing an object requires it to have some identity to help access it back. In computer science, objects are identified by their physical existence, so two exact copies of the same object are considered as two different objects occupying two different areas in the memory with two different identities. Object database follow the same concept by concentrating on objects as the elements, thus allowing for two similar copies to be stored as two different objects. In relational database, on the other hand, objects are identified by their contents, so two exact copies of the same object are considered as one object occupying one area in the storage with one identity. This is mainly because relational model is concerned with storing data, rather than objects, so it is considered redundant to store two copies of the same data. For simplicity, we will consider the first storage concept only (the object concept) as a general concept. The relational database concept can be considered in a larger, more specialized discussion. Containers can be classified according to the way they store and allow access to their objects into sequential containers and

associative containers. They allow access to their elements either in a sequential or a random access method.

Tree containers are more complex data structures that can be build using some of the STL containers. Tree containers have internal structure that is controlled by the design of the tree, and they are built in multiple levels. Access methods in tree containers are also more complex than the other, one-level containers. Each access request by the user typically involves more than one element of the tree. We can consider this type of access as automatic access since it progresses internally from one tree element to the other without user intervention. We can build database indexes as tree containers. Details of these structures are provided in the design of database index containers, chapter 5.

Table2.1 shows a brief comparison between different access types for the containers

Access Category	Container Category	Number of Levels	Access Mechanism
Sequential access	Sequence container	One	Container knows physical ID of one element. Each element knows physical ID of one other element (next element). No logical ID. User can advance to next element only. Current element translates it to one physical ID.
Indexed random access	Sequence container	One	Container knows physical ID of all elements. Each element is donated logical ID equals its position. Container binds positions with Physical ID's. User can directly go to any position. Container translates it to one physical ID.
Keyed random access	Associative container	One	Container knows physical ID of all elements. Each element has intrinsic logical ID (a key). Container binds keys with physical IDs. User asks for any key. Container translates it to one/more physical ID.
Automatic access	Tree container	Multi level	Each element has logical ID; a key. Container knows physical and logical ID of some elements. Each element knows physical and logical ID of some more elements. User asks for any key. Container translates key to some elements, each of which further translate same key to other elements. This continues until reaching terminal elements.

Table 2.1: Comparison between Different Access Types

2.1.2 Container Adaptors

Containers have some characteristics (associative, or sequential containers with random or sequential access, etc.), which will determine the functionality offered by the container to the user (how to access its elements, how to insert or delete an element, and even how to name an element).

Sometimes the user might be interested in only one part of this functionality; the rest is not to be used at all. In this case we might want to use the original container while leaving the unused functions in it, or we might want to mask them from the user, it accidental use of them could cause problems. In this case we may use a container adaptor to help mask the unused functions. It will have an interface to the user offering only those functions the user is allowed to use; the adaptor then simply translates them to the ones offered by the original container. The rest of the unused functions will never be called, as the adaptor does not allow the user to access them. Another use of container adaptor is when the functions used by the user have different names that those offered by the container. Instead of changing the signature (the name or argument or both) of the functions used by one party (the user or the container) to accommodate the other, an adaptor can be used for this purpose. It takes the user call to a particular function and delegates the call to the suitable function(s) in the container.

2.1.3 Template Parameters

Containers are made independent of the types of the objects they store by using template parameters. Each container type is implemented only once using a general template parameter as the type of objects stored in it, allowing the programmer to completely implement the desired functionality of the container without knowing the type of objects that will be stored in it.

```
template <class T> class vector {
                                    //implementation in terms of template T as
                                    //the generic type for the stored elements
                                      }
```

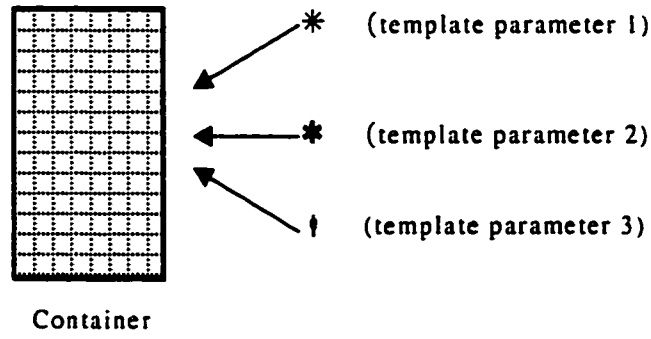


Figure 2.1: The Container and Templates

Later on when the container is to be used, an instance of it is created and passed the type of objects to be stored in, be it an *int*, a *double*, or a complex user defined type. This parameterization allows for code reuse by making the same container code usable many times. Different container object of the same class can be instantiated and passed - as template parameters- different types of objects to store.

```
vector <int> intVec ;
vector <student> studentListVec;
```

The programmer has different container types (with different functionality) to select from, according to the application needs.

2.1.4 Iterators

Containers do not allow direct access to their stored objects, but rather through another class called an Iterator. The Iterator is an object that can reference an element in a container. It allows programmers to visit each element in a sequential or random (indexed) way depending on the type of the container.

X++ ++X X-- -- X	Increment / decrement
X + n X - n	Move n elements forward / backward
X = Y X+=n X - =n	Assign to / from an iterator
X-Y	Distance between two iterators
X = = Y X != Y X>Y X <Y X>= Y X <=Y	Logical comparisons
X [n] *X	Access

Table 2.2: Iterator Operations

This is another technique that adds to the concept of code reuse. Programmers do not need to know the exact interface to different containers to use them. They only need to know one common interface; that of the iterator. Different container types are readily accessible

through the same piece of code, since all containers support iterators that have the same interface. This allows for changing the container type without significantly changing the code that is using the containers.
Iterators support some or all of the following operations, depending on the container they are accessing. Table 2.2 lists these operations associated with two iterators X, Y and an integer n.

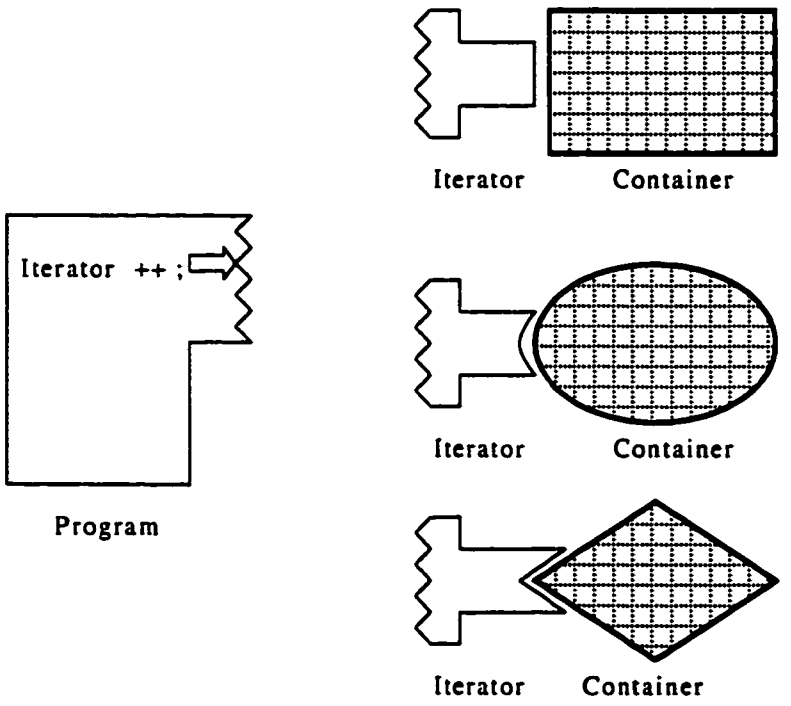


Figure 2.2: The Iterator Interface

2.1.5 Iterator Traits

The STL iterators have become a standard way of dealing with containers. Each iterator type posses a standard set of characteristics that completely describe it. Some of these characteristics are a set of types associated with each specific iterator type. A function, when passed any iterator through its argument, should be able to extract the different type information associated with that particular iterator. For example, an

iterator points to an element, and a function that receives an iterator should be able to find out the type of that element.
Some of these types are shown in figure 2.3.

```
value_type;          //The type of the element the iterator points to.
difference_type;     //A signed value indicating the distance between
                     //two iterators
pointer;             //Pointer to the type_value of the iterator.
reference;           //Reference to the type_value of the iterator.
iterator_category    //A tag telling the category of the iterator.
```

Figure 2.3: Some Standard Iterator Associated Types

2.1.6 Algorithms

Iterators separate the logic from the container types and element types stored in them. The same algorithm, written once, can be applied to different containers with different stored elements by using iterators in the algorithm code. This allows for a complete implementation of generic algorithms without knowing the exact type of the container (its functionality) or the type of elements stored in it.
This is achieved by writing the algorithms to deal with a standard Iterator interface common to all containers. Later in the program, these algorithms can be imported and added to the existing code and used directly without modifications, assuming that the existing code is using the Iterator-Container-Template building concept. This is a further contribution to the code reuse in importing and using already written algorithms in our code.
The algorithm shown in figure 2.4, which is already implemented in STL, will fill any container with any value of type T. All it needs is a standard iterator (of type ForwardIterator) that refers to some container, an object value of type T to use in filling the container, and a compatibility relationship between T and the type stored in the container.

```
template <class ForwardIterator, class T>
void fill (ForwardIterator   first,
          ForwardIterator   last,
          Const T&          value) ;
```

Figure 2.4: An STL fill Algorithm

Later in our code we can use this algorithm to fill a specific container with a suitable type of elements. By suitable we mean that the type T of the value must be assignment-compatible with the type of elements the container stores. This is because the algorithm will perform the assignment operation from value into container elements. For an algorithm to access the elements of a container, it needs to find its beginning and end positions (in fact the beginning and the past-the-end positions, an STL standard, denoted by the [a,b) notation which means the range from a to b including a and excluding b) or more generally any starting and ending position of a sub range within the container where the algorithm will be allowed to work. In the above algorithm, we see that it is allowed to work within the range [first, last). For some data structures, like an array object *Arr* of n elements, it is easy to find its beginning address, which is the array name *Arr* itself, and the past-the-end address, which is the address *Arr+n*. For STL containers, however, these positions cannot, in general, be easily found. Therefore each container must provide two methods begin( ), and end( ) that return an iterator to locate its first and past-the-end positions respectively. They can then be assigned to other iterators or passed directly to the algorithm when using it as in the following code.

```
vector<int> myVector(100); //creates a vector of 100 integers
fill (myVector.begin ( ), myVector.end( ), 1);
```

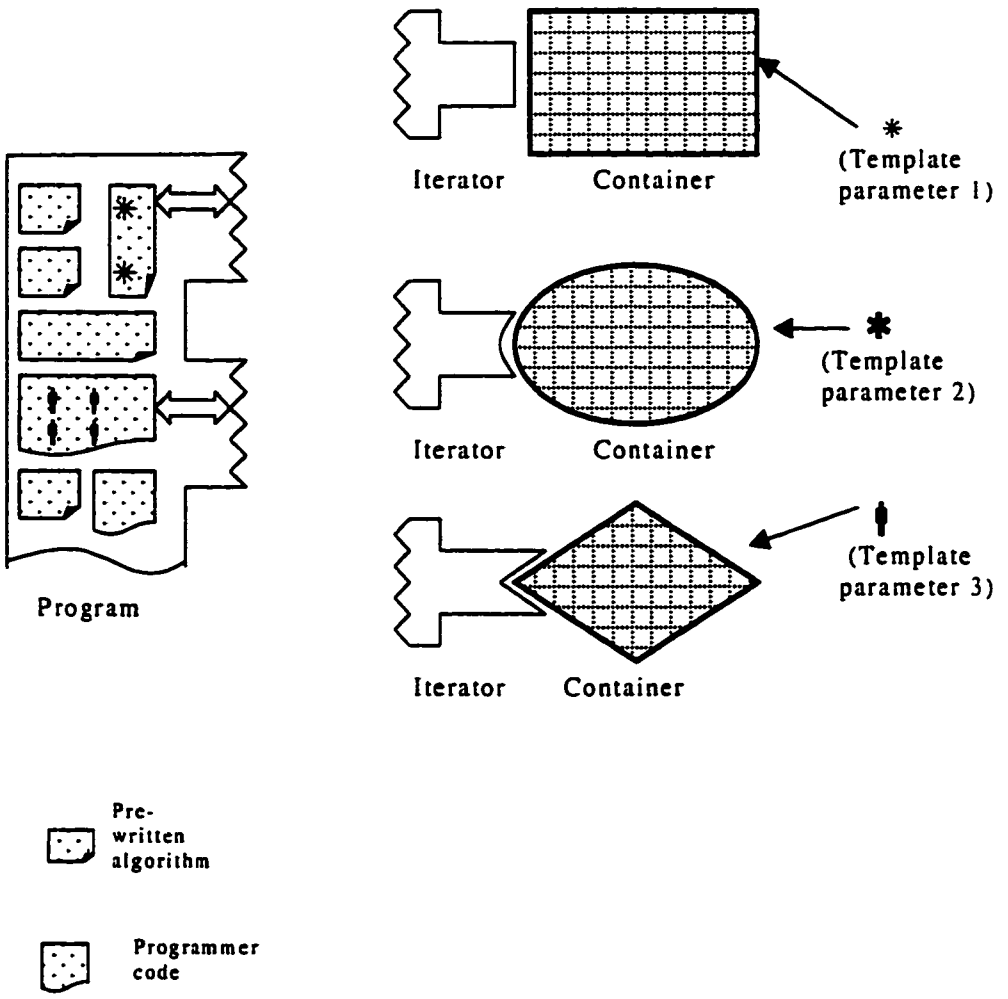


Figure 2.5: Using Same Algorithm for Different Containers

By following the STL standard process for templates, containers, iterators, and algorithms when building a database index, it will be easy to extend the code to support new data types, new queries, new access methods and produce a readable standard code with some off-the-shelf modules easily integrated into the code. The STL comes with a large number of generic algorithms that can be used in implementing an index to provide the needed functionality for searching or updating the database. More algorithms can then be added to increase its capabilities. Figure 2.5

sketches the concept of using different algorithms with different containers.

2.1.7 Functors

Function objects, or *functors*, are generalization of function pointers [Rob00], the same way as iterators are generalization of pointers. They add more functionality to them, making them much more powerful and conformant to STL specifications for building blocks. Function pointers can be bound to one of many functions at compile- or runtime depending on a condition (like user input for example). They are then used to perform the action they were bound to by overloading the *operator ( )* (the call operator) on them.

2.1.8 Function Adaptors

Some functions accepted two parameters in their argument and a compatible binary functor (that represent a binary operation). The function will call the functor, pass its two parameters to it as operands, and do something with the returned value from the functor. For a function that accepts a functor of type BinaryPredicate, an implementation would look like the one shown in figure 2.6.

There are other types of functors defined in the STL, like UnaryFunction, (which take one parameter as input and return a result) and UnaryPredicate (actually called "Predicate", and takes one parameter as input and return a Boolean). What can we do if we have a function that takes only one parameter in its argument and we want to pass to it a binary functor, like less ( ). How can the function obtain the second operand to pass it to the functor? Also assume that we want to pass a binary functor to a function that is expecting a unary functor in its argument instead?

```
less <double> P;
plus <int> Q;

int Fn1 (int a, int b, Q ( ) ) { return  Q (a, b) ;};
//Fn1 expects a binary functor that returns an int,
//a functor of type BinaryFunction.

bool Fn2 (double a, double b, P ( ) ){return P (a, b) ;} ;
// Fn2 expects a binary functor that returns a bool,
// a functor of type BinaryPredicate.
```

Figure 2.6: Functions with Different Functor Types

The STL provide adapters that can adjust this kind of irregularity when combining the building blocks, so that we can still put some blocks together that seem incompatible at first sight.

Passing sequences as input parameters.

The main advantage of the functors and adapters is that they can be applied repeatedly on a sequence of elements; a container. We can write (or reuse) our functions and pass them a container (or more) and functors with adapters if needed.

The function F4 ( ) can be passed a complete sequence of elements (a container) to test for the elements that are less than 5 , and do something to those elements (e.g. replace them with value 10). So we pass to F4 ( ) a container of elements to test (by passing an iterators to first- and past-the-end positions) and a functor less ( ) which is bound to 5 to be compatible with F4( )

```
F4 (iterator first, iterator last, bind2nd ( less <int> ( ), 5 );
```

F5 accepts two sequences and a binary functor that applies a binary operation to them, element by element and produces a new sequence of the results. Note that we need to pass three containers to the function: the two input containers and the output container to be filled with the results. Note also that we need the size of one input container only (the smaller of the two, passed before the larger one, if they were not equal in size) since the algorithm will stop as soon as any input container is exhausted. We pass the beginning and past-the-end positions of first container, and the beginning of second input container, and the beginning of the output container

```
Template <class InputIterator, class OutputIterator, class BinaryOperation >
OutputIterator manipulateContainers (InputIterator     begin1,
                                     InputIterator     end1,
                                     InputIterator     begin2,
                                     OutputIterator    result,
                                     BinaryOperation   bin);
```

Figure 2.7: Passing Multiple Containers to a Function

And we can use the function as shown in figure 2.8. Note that the container list provides two functions begin( ) and end( ) that returns iterators to the first and past-the-end locations of the container respectively. STL containers must provide these standard functions. What the function does is, it selects every two corresponding elements from the two input containers, passes them to whatever binary functor object it has in its argument, gets the output from the functor, and puts it in the output container. The function repeats the process until the first input container is exhausted. STL has allowed the function to be independent of the container, the container elements, and the operation applied to them.

```
void main ( )
{
int in1[10], in2[12], out[10];
int* in1_end = in1+10   //need the location of past-the-end element of in1

//... code to fill the two input containers in1 and in2

manipulateContainers (in1, in1_end, in2, out, plus<int> ( ) );
//the array name is a pointer to its first element

// ... other code

list<double> list1, list2, list3, list4;
//we use same function on a different container with a different element type and
//different functors

// ... code to fill list1 and list2

//list3 will have element-by-element subtraction of list1 and list2

manipulateContainers        (list1.begin( ), list1.end( ),
                             list2.begin( ),
                             list3.begin( ),
                             minus<double> ( ) );


//list4 will have element-by-element multiplication of list1 and list2
manipulateContainers        (list1.begin( ), list1.end( ),
                             list2.begin( ),
                             list4.begin( ),
                             multiplies<double> ( ) );
}
```

Figure 2.8: Using the Function manipulateContainers

2.1.9 Allocators

Containers are responsible for managing the access and storage of their elements. The access is allowed by providing suitable iterators and a set of functions to provide information about the elements, return a reference to- or add/ delete an element. The storage is an essential part to the container, allocating memory to new elements, and freeing memory from

the deleted elements. Normally the user need not worry about storage management when using a container. This is done behind the scene without any user intervention and this greatly simplifies the use of containers. On the other hand, some special applications require a special kind of storage management and cannot simply work with the standard one. This would mean that those special applications will have to discard the STL containers altogether and write their own special containers that have a special storage management. This means a lot of efforts, most likely without the efficiency and elegance of STL code. To be able to serve those special cases as well, while still keeping the containers simple for everyone else, the STL made a further separation of building blocks. The Container is not exactly responsible for the storage. An allocator class was made responsible for the storage of the container elements, and each container handles storage by simply asking an allocator object to do that. To keep things behind the scene for users, each container uses a default allocator for its storage needs, unless given a specialized allocator to handle the job. For example, the container "vector" has the format:

```
template <class T, class Allocator = allocator<T> > class vector ;
```

We can see that a vector of integers can be instantiated by simply writing

```
vector <int> intVec ;
```

and a default allocator object will be used, without user intervention. On the other hand, the same container can be passed a specialize allocator, written by the user, or imported from another library as follows:

```
vector <int, my_allocator> myVec ;
```

In this case we made full use of the standard container class in a special application without writing a special container. In writing a specialized allocator, they must have the standard interface of the STL allocators to be integrated seamlessly into any STL container. This is very useful in database, where we need a special storage on the hard disk, where files are too large to fit in memory. We can now put all the building blocks together on one diagram. These building blocks satisfy the standard interface provided by STL for components. They are compatible with each other, and we can replace on or more of these blocks with a specialized block provided that it has the same interface of the corresponding STL block. Figure 2.9 shows a general system Layout for this system.

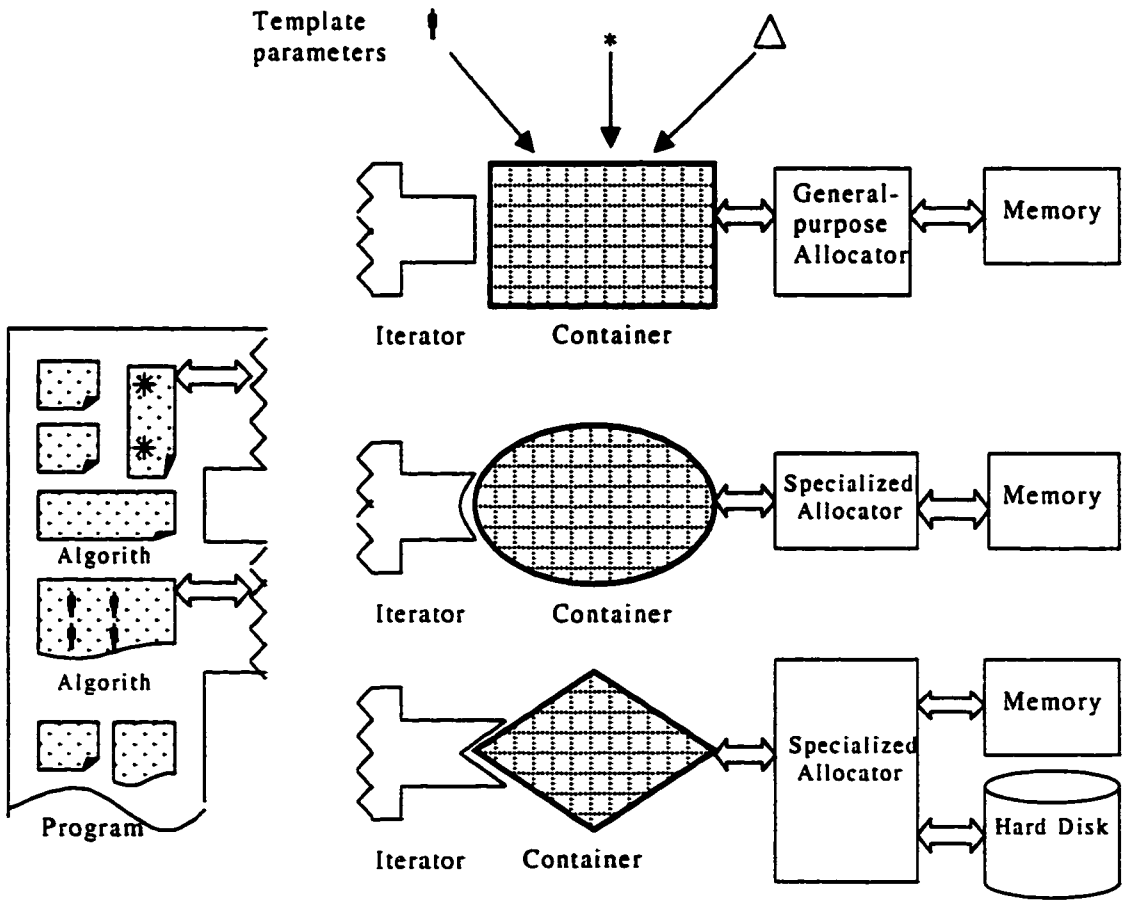


Figure 2.9: The General System Layout

2.2 Building Blocks Connectivity

The STL provides us with a complete set of compatible building blocks that can be connected together to build complex systems. Using these blocks, we can produce a blueprint design of the system, which can be drawn as a directed graph. The nodes would be STL blocks and the edges would be the connections between them. Semantically, a connection between two nodes would represent a use relationship between the two STL blocks represented by the nodes. Connecting these blocks is not, however, done at random. We cannot simply connect any one block of our choice to another and put them to work. There are certain rules that apply when trying to connect two blocks together. In order to show these rules easily, we will divide the three fundamental building blocks, Containers, Iterators and Algorithms, into categories. Then we can show how these categories connect together properly.

2.2.1 Iterator Categories

We can categorize the iterators according to their movement ability. Some iterators have more movement freedom than others. Figure 2.10 shows these categories.

Input Iterator

Input iterators have the least amount of movement; they can provide only one step forward. They are single-pass iterators, meaning that they cannot be used to access the same container twice; they might give different values each pass. They are used as rvalue to take input into program. Output iterators are similar to them, only they work as lvalue, taking output from the program to a container.

Forward Iterator

Forward iterators have the ability to move forward. They support the operator++ to step forward incrementally. They support multipass

concept, where they give the same values every time they pass through the container. They do not support backward movement. To access the previous element, we increment the iterator all the way to the end of container, and start a new pass from the beginning until we get to that element.

Bidirectional Iterator

Bidirectional iterators have all the abilities of the previous two categories. They also support the backward (reverse) movement using the operator --.

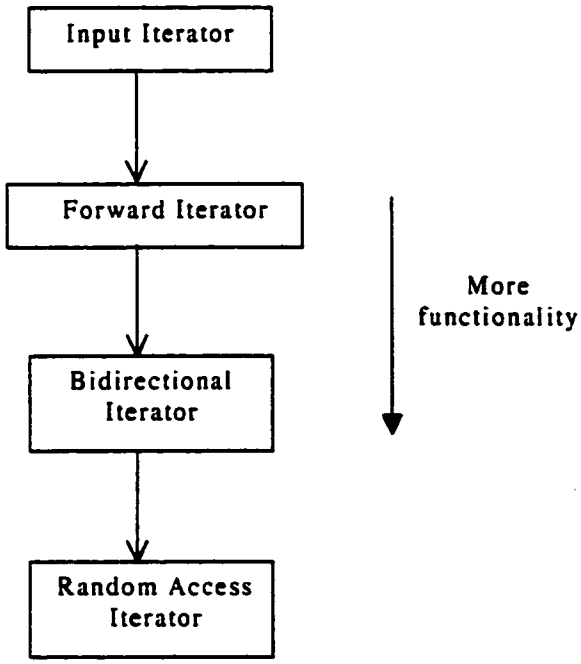


Figure 2.10: Iterator Categories

Random Access Iterator

Random Access iterators are the iterators with the highest degree of freedom. They can move forward, backward and also allow direct access to an arbitrary element using operator [ ] to specify the desired element by its index.

2.2.2 Container Categories

We can categorize the containers according to the movement freedom they offer to iterators trying to access their elements. Each Container can allow iterators to iterate through its element with certain degree of freedom, depending on how the container organized and stored its elements. Figure 2.11 shows these categories.

Compatibility between Iterators and Containers

If the iterator has more movement ability than what the container can afford, the extra iterator abilities cannot be used unless we upgrade the container. Using this extra functionality without upgrading the container category is incorrect and dangerous. It can crash the program, or at least provide incorrect information

Also if the iterator has less ability than what the container can offer, the extra container abilities cannot be used, unless we upgrade the iterator. This is, however, not a serious practice, and will cause no harm since we use iterators to access the container so we will have no access to any incorrect functionality; we will just lose some.

Internal and External Iterators

Iterator can communicate with internal and external iterators. For external iterators, they are built outside the container, but for a specific type of containers; meaning that the internal structure of the container must be known to the iterator. They can be used with any container of the same type by passing to them a particular element of the container and using them to move in the container. For internal iterators, the container provides them as methods that return an iterator. Each container can return iterators of different categories. The container is wise enough not to provide an internal iterator with higher movement ability than the container can afford, yet they can typically return a lower category iterator if that is all we need.

Input Container

Input Containers are conceptual containers capable of providing only input iterator. They would only provide input to the program in a single pass concept, like input stream object. An input container is conceptual as it does not really belong to the program itself, but it is more of an external component that provides input to the program.

The keyboard can be seen as an input container that provides input to the program. A second pass through this container (in the form of running the program code twice and accepting a set of input variables by a set of cin statements each time) may produce different elements as typed in by the user in every run. This set of cin statements can be seen as receiving their values from the "conceptual" input container; the keyboard.

Forward Container

Forward Containers are containers that are organized such that they can be accessed in a forward incremental direction; like single-linked-list data structure. No backward movement is allowed. This type of containers can only provide a forward iterator or an iterator with less functionality.

Bidirectional Container

Bidirectional Containers store their elements in a sequential way and allows access to them in a forward or backward direction, in incremental steps only. This is similar to the concept of doubly linked list data structure. They can therefore provide Bidirectional iterators (or less-capable iterators if needed). Also any external Bidirectional iterator can access this type of containers with no problems.

Random Access Container

Random Access containers store their elements such that they can be accessed in both directions, and can also support direct access to an

arbitrary element without the need to incrementally access all elements before it. Obviously they can support a random access iterator or less.

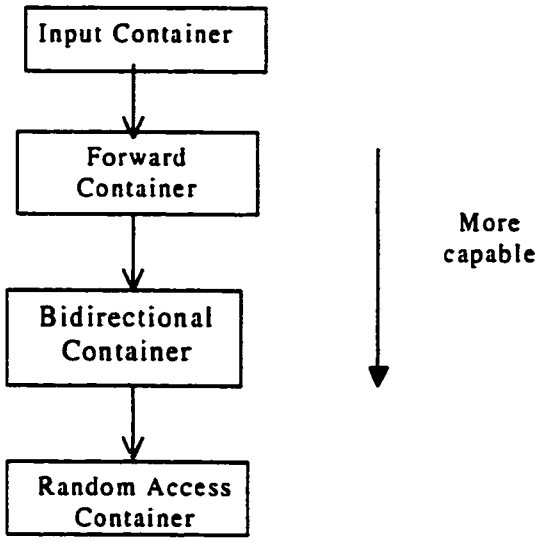


Figure 2.11: Container Categories

2.2.3 Algorithm Categories

A fundamental STL concept was to use iterators as an interface between algorithms and containers, allowing the same algorithm to be used on different containers and vise versa, promoting the cut-and-past concept on code; or code reuse. It might seem that we could say: "iterator allows the use of any algorithm with any container". This is not exactly correct. A more exact statement would be: "iterator allows the use of many algorithm with many container"; hinting that some algorithms do not work with some containers. To make it clear, we can categorize algorithm using the same criterion. An algorithm will be categorized according to the iterator it uses. Figure2.12 shows these categories.

Compatibility between Iterators and Algorithms

For simplicity, we will consider the category of iterator needed for the algorithm, and ignore the category of the container as it can be inferred

from iterator category. After all, this is why we use iterators: to make algorithms container-independent. Obviously the algorithm can connect to a higher-category iterator without any problems; it only will not make any use of the extra set of functionality provided by that iterator. Each algorithm will therefore be written with the minimum iterator category required to work correctly. Connecting an algorithm with a lower category iterator is, however, dangerous. The algorithm will be using some functionality that is not provided by the iterator, which will result in an incorrect performance, or even a program crash.

Input Algorithm

Input Algorithms require an input iterator to work properly. Their functionality would obviously be to input some elements to a container in an incremental unidirectional way, in a single pass concept.
Apparently this type of algorithms works fine with the lowest category of the containers (input container) and needs only the lowest category of iterators (input iterator).
For example the algorithm *accumulate* in figure 2.12 needs an input iterator to work correctly.

Forward Algorithm

Forward Algorithms need forward iterators to work correctly
For example, the algorithm *remove* in figure 2.12 needs a forward iterator to work correctly.

Bidirectional Algorithm

Bidirectional Algorithms needs at least a bidirectional iterator. This kind of algorithms will need to move forward and backward to work correctly.
For example, the algorithm *reverse* in figure 2.12 needs a bidirectional iterator to work correctly.

```
template < class InputIterator, class T >
T                       accumulate (InputIterator first,
                                    InputIterator last,
                                    T init) ;

template < class ForwardIterator, class T >
ForwardIterator         remove (ForwardIterator first,
                                ForwardIterator  last,
                                const T&  value ) ;

template  < class BidirectionalIterator >
void                    reverse (BidirectionalIterator first,
                                 BidirectionalIterator last);

template  < class RandomAccessIterator >
void                    sort (RandomAccessIterator first,
                              RandomAccessIterator last);
```

Figure 2.12: Some Algorithms

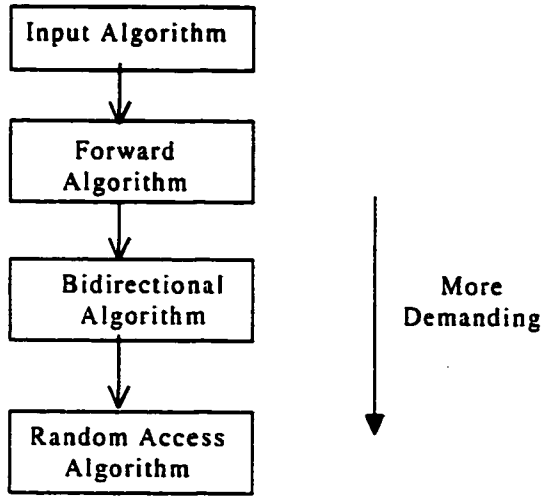


Figure 2.13: Algorithm Categories

Random Access Algorithm

Random Access Algorithms are the most demanding algorithms of all. They need the highest iterator category, a random access iterator. This kind of algorithms will need to move forward and backward and access any arbitrary element in the container to work correctly.
For example, the algorithm *sort* in figure 2.12 needs a random Access iterator to work correctly.

Downgrading an Algorithm

It is worth mentioning here that most algorithms can be downgraded to a lower category and still work correctly. This will, however, not be an efficient algorithm anymore. The minimum category for an algorithm will guarantee that it will have the best performance in term of time complexity as guaranteed by STL. Downgrading an algorithm by rewriting it such that it can be used with a lower category iterator could mean a much less efficient performance. For example, an algorithm that requires direct access iterator can be rewritten to use only Bidirectional iterator. Each time the algorithm will need to jump to another element, it will have to start from the beginning or current position and access all elements lying before the desired one. Such an algorithm could take very long time to complete.

2.2.4 A Blue Print for the Connectivity

Figure 2.14 shows how we can connect the categories of STL building blocks correctly. An arrow means that the two blocks are compatible.
We can divide the components into other categories using different criteria. For example, we can divide iterators according to their mutability; read-only iterators allow algorithms to take a copy of container elements without being able to change it (a const iterator). The container element itself can be constant for all algorithms, or can be

mutable by other algorithms. Containers can also be divided into sequence and associative container, regardless of the above categories.

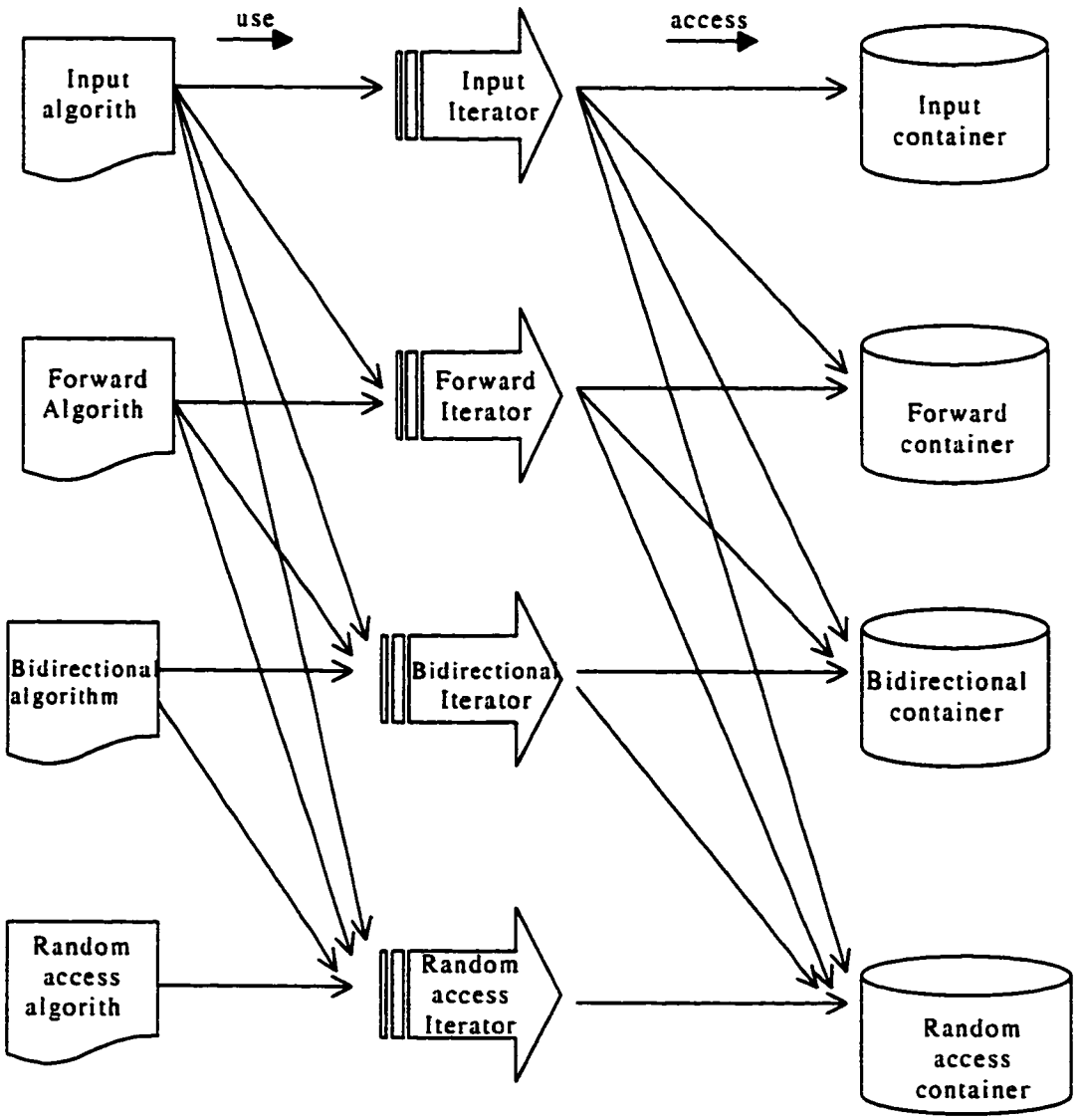


Figure 2.14: A Blue Print for Connectivity

Chapter 3 Database Indexes

3.1 Introduction

Indexes help access information. To make a large amount of data useful, it has to be organized and classified into logical parts, then an index added. The index tells us where to find the data of interest in a data collection. Using a specific search method, we can go through an index and get a location where the data is to be found. A simple example is the table of contents of a book. We use the table of contents to find information in the book using the topic name as a search method through the index.

To build and use such an index we perform the basic steps:

1- We divided the book data into Partitions (in this case chapters, sections and paragraphs).

2- We found a key for each partition to give us some clue about the contents of the partition (in this case a header or a caption for each part relevant to its content).

3- We gave a physical address; a reference to each partition, marking its exact location (in this case unique page numbers).

4- We decided on a certain access method; a criterion of how to search for information (in this case since the number of entries is relatively small, the index would be limited in size to few pages, we will search the index sequentially by scanning the keys from the first one until we find a topic of interest. This would take O(n) time to search where n is the number of entries. Since n is small, an O(n) algorithm performs well enough. Sequential listing of contents will also help to give an overview of the book, but that is not one of our concerns for an index.

5- We built the index by collecting all the keys and references in one part – the table of contents.

Another example of an index is the keyword index that can be found at the end of the book. This index is built to locate data in the book using keyword as the search criterion. We follow the same four steps to design and build the new index:

1- Partitions are the keywords we want to highlight in the data and include in index.

2- The best key for the keyword is the whole keyword (or part of it).

3- Each keyword has a page number (or numbers) where it is mentioned in the book

4- Since this index will have many more entries than the previous one (n, the number of keywords, is now much larger, O(n) might be unacceptably high. Searching a table that contains a few thousand unsorted keywords can be annoying. We thus decide to sort the keys alphabetically and use binary search as an access method. We go to the middle of the alphabetical index and check the word and then decide if the keyword we are searching was before or after the middle, and so on. This will have a time complexity of O(log n) on the average, giving an improved performance

5- We then build the index and add it to the book.

Saving time is clearly an advantage in using an index. We could scan sequentially through all the book headers (chapters, sections and paragraphs) to find the topic we are looking for, but scanning through the index gives the same results in a much faster time. With keywords it is even clearer. To search for a keyword in a book without a keyword index, we need to search every word in the book sequentially which is a lot of work.

3.2 Index Tree Applications

Almost any kind of data can be stored in a database. Numbers, text data, images, maps, fingerprints and even audio and video files are typical examples. A lot of work has been made on the concept of index tree, resulting in many different types of trees suitable for this variety of applications. The idea is to be able to categorize data in such a way that we can retrieve it fast, and then build an index that allows us to locate data of interest in an efficient way. Similarly, different index concepts were developed to be able to deal with the variety of application data. In the next pages, we will briefly present some of these index trees, and show how to use them with database applications.

3.3 B+ tree Index Example

Figure 3.1 shows a simplified B+ tree index layout. It contains pages that store pairs of keys and pointers. Pages are laid out in a tree-like structure. The top level contains only one page, the root page. The lowest level contains the data (or references to the data). Each page in that level is called a leaf page. Data is included at the leaf level only. No data is allowed to exist in other levels. The tree is balanced; all leaves are at the same level. The number of pointers, R in a page determines the number of children of that page. This number, called the fan out of the tree, has a maximum allowed value, and is fixed for the tree. It shows the multiplication factor from one level to the next. Each page is allowed to be totally or partially filled with a factor called fill factor. The minimum fill factor is typically 50% and the maximum is 100%, except for the root page, which can have as little as two children or as much as any other page. The example shows the basic search operation for a value 202 in the leaf level, starting from the root. Other operations involve writing to the index, or modifying its contents, like insertion and deletion operations.

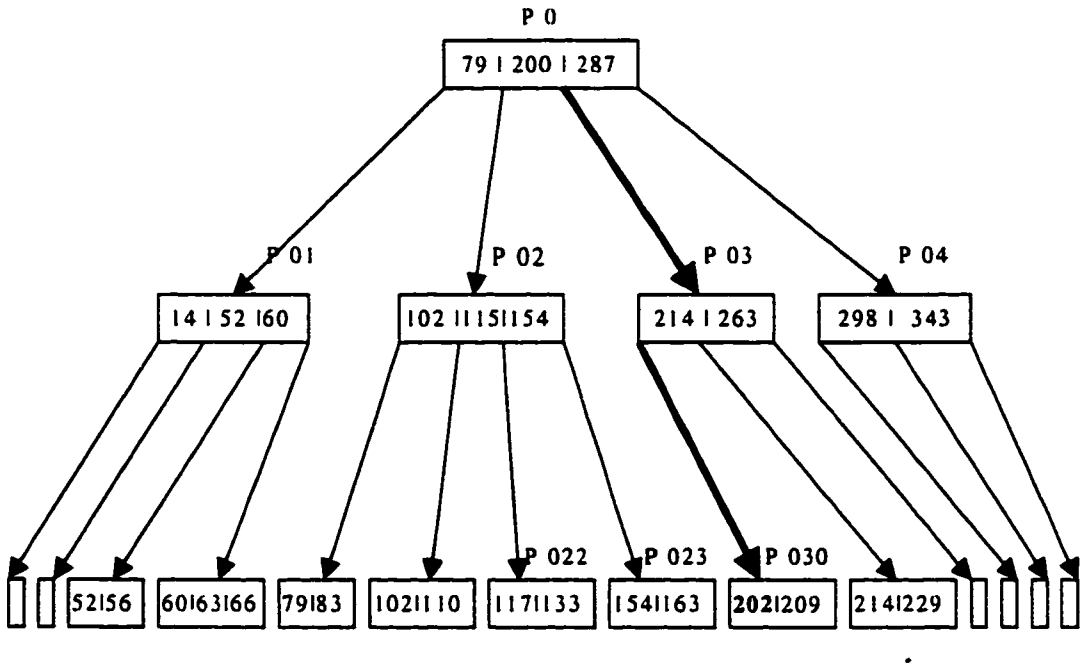


Figure 3.1: B+ Tree

These operations are far more complex in their algorithms and procedures than a simple search operation. A single insertion or a deleteion operation can result in one simple step, or it can trigger a chain reaction leading to a large number of related steps spread over multiple index pages with multiple sub operations of splitting pages, merging pages or moving some contents between pages before it comes to an end. The algorithm for these operations and their related sub operations must guarantee the integrity and correctness of both data and indexes after any number of operations with any combination of data.

3.4 General Domain Applications

The previous example showed an index using integers (or comparably strings of characters) as its data type, and equality as its search criteria (query). This works well for traditional relational databases. It can also be

extended to serve other queries (range query for example). For some applications, however, that example does not quite serve their needs.

One domain of those applications is the spatial database, with a multi dimensional data type. The data in this domain can be represented in the Cartesian space as multi-dimensional objects. For examples geographical information about earth surface, cities layout in a country, or detailed city maps for buildings, streets, etc. Some index trees have emerged to deal with these database applications including K-D-B-tree [Rbn81], R-tree [Gut84], R+-tree [SRF87], and R*-tree [BKS90].

As an example, the R-tree is a height-balanced tree similar to the B+-tree, but it considers the data type to be an n-dimensional polygon in Cartesian space, composed of a group of lines related together. Figure 3.2 shows a collection of polygons representing some data. This could be any geometric figure, a city map, districts map, etc. Arcs can also be considered by approximating them to a number of lines.

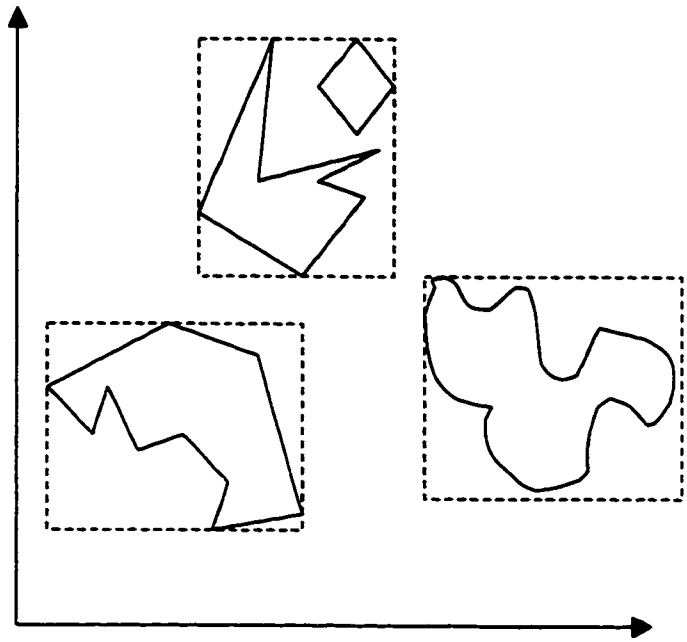


Figure 3.2: Finding the Key for Partitions of Polygons

To build an index, we need to partition data, by considering each polygon or a group of polygons as one partition. We then need to find a key for each polygon (or group of polygons). The key is the bounding rectangle to the spatial partition, covering all the partition (of one or more polygons) while reducing the data size from many points of (x, y) describing each polygon to just four points. To represent a rectangle we just need two points; the op left, and the bottom right corners. Now this key covers all the partition with much less data size; the desired nature of any key. Note that partitions might be allowed to overlap depending on the application. This will not affect data integrity, as for overlapping keys their associated data can still be easily distinguished.

Searching will typically have a point or an area (another polygon) with the search key as the smallest rectangle to cover this area. The search mechanism will try to find those polygons in the database that have something in common with it, by finding the database index keys (rectangles) that intersect with it. Figure 3.3 shows an example of searching a key and we can see the three data keys that match that key (by intersecting with it).

The figure shows that some rectangles might intersect with the key rectangle we search, giving positive search result while the two actual data parts inside the polygons do not intersect, making the data found by searching the database useless. This is a side effect of using keys for the polygons and is not harmful in being unable to locate correct data inside the database; It will just impose some acceptable performance burden by accessing some useless data once in a while.

Similarly for an index, we need to merge partitions by finding one key to cover all merged partitions (for merging two pages, or for building a higher level in the index where each key will cover many keys below it). Other applications use sets as their data types where data entities are sets of related or unrelated elements. Keys for sets can be found using

different techniques and algorithms to extract a summary for each set. Set theory with its notations like union, intersection, subset... is used here to combine keys, search for a key, etc.

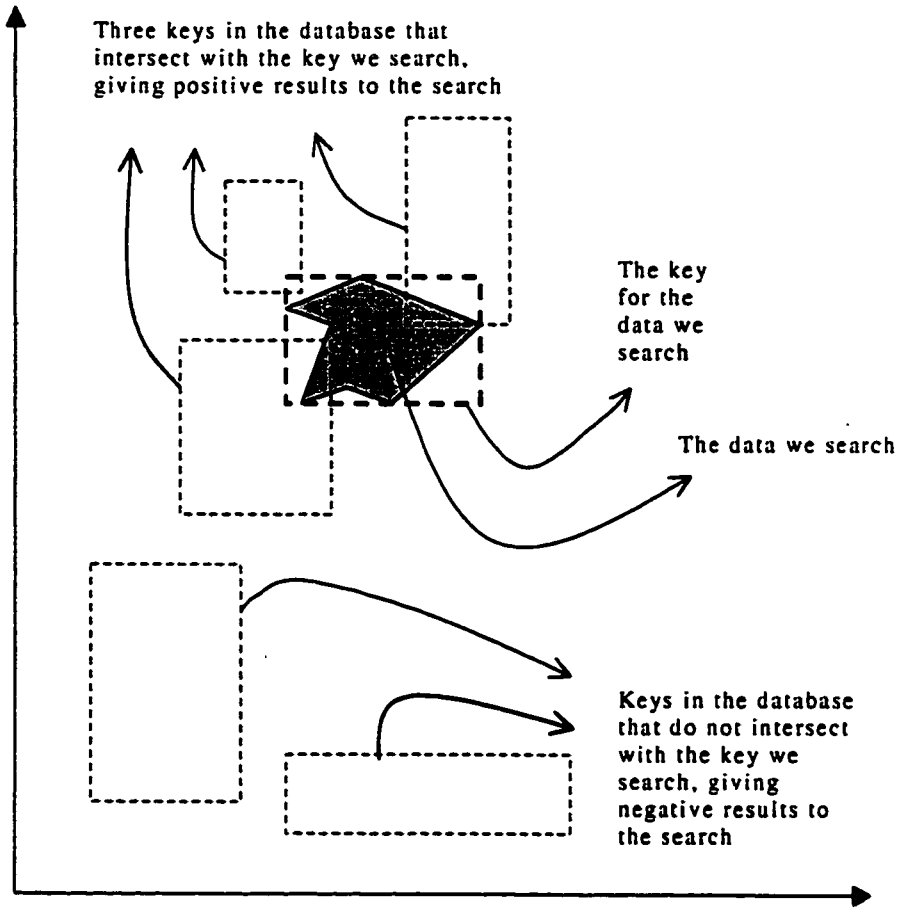


Figure 3.3: Searching Data Using a Key to Find Matching Keys

3.5 Similarity Search Applications

The general domain database and the linearly ordered domain have concrete results for a match, either true or false. We can see them as equality search applications. In similarity search, on the other hand, we do not look for an exact match, we are looking for data that is similar to what we have. This is useful in domains that need to build a database of

images, like multimedia, journalism, art, astronomy, image and voice recognition, traditional medicine (e.g. 2D X-ray and 3D CATSCAN medical imaging) as well as new genetics and protein databases.

One main difference in these applications is that data is not directly represented by concrete numbers. Another difference is that even after we have translated the images to numbers (they have to, to be stored in digital medium), we cannot simply compare these numbers to decide on matching.

Considered a simple case of scanning and storing the same image multiple times as pixels with digital values, then comparing them with a similar image to retrieve back these stored occurrences of the image. Apart from the great inefficiency due to the large amount of data to be compared, a slight shift or rotation, or even a rounding error might give a different number for few pixels, giving a negative search result for the same image, which is unacceptable. If those images were compressed to increase storage- and transfer efficiency, chances are all occurrences of the image will be different.

Seidl and Kriegel [SK01] show how to add flexibility to large image databases including pixel-based shape similarity to tolerate and compensate for such errors and locate similar images.

This example shows that we are looking for the same contents with similar digital representation. Similarity search can have much broader meaning when we want to look for similar contents. This makes the subject more complicated than just numbers. Deciding whether objects are similar or not is very subjective, and could be different for different persons, or even for the same person under different circumstances. We will show this with an example.

Consider three objects: A goldfish, a white shark, and a dolphin and we want to select the two most similar ones out of them. For a biologist, the gold fish and the shark are similar, both are cold-blooded, fish, while a dolphin is a worm-blooded mammal with lungs to breathe. Now we change

the person. For an ordinary person swimming in water, the gold fish and the dolphin are similar, both are peaceful water creatures, while a white shark is a terrible monster. Now we change the circumstance of the same person. If this person is looking for a house pet to put in a glass bowel, the dolphin and the white shark are similar and must be avoided, both are too large and have enormous appetite and not sold in pet stores, while a gold fish is perfect for the purpose.

Due to this subjectivity, content similarity research developed different criteria to manage and decide on the similarity decision. We can recognize two main parts: criteria for extracting a key, and criteria for comparing keys to decide on similarity between them.

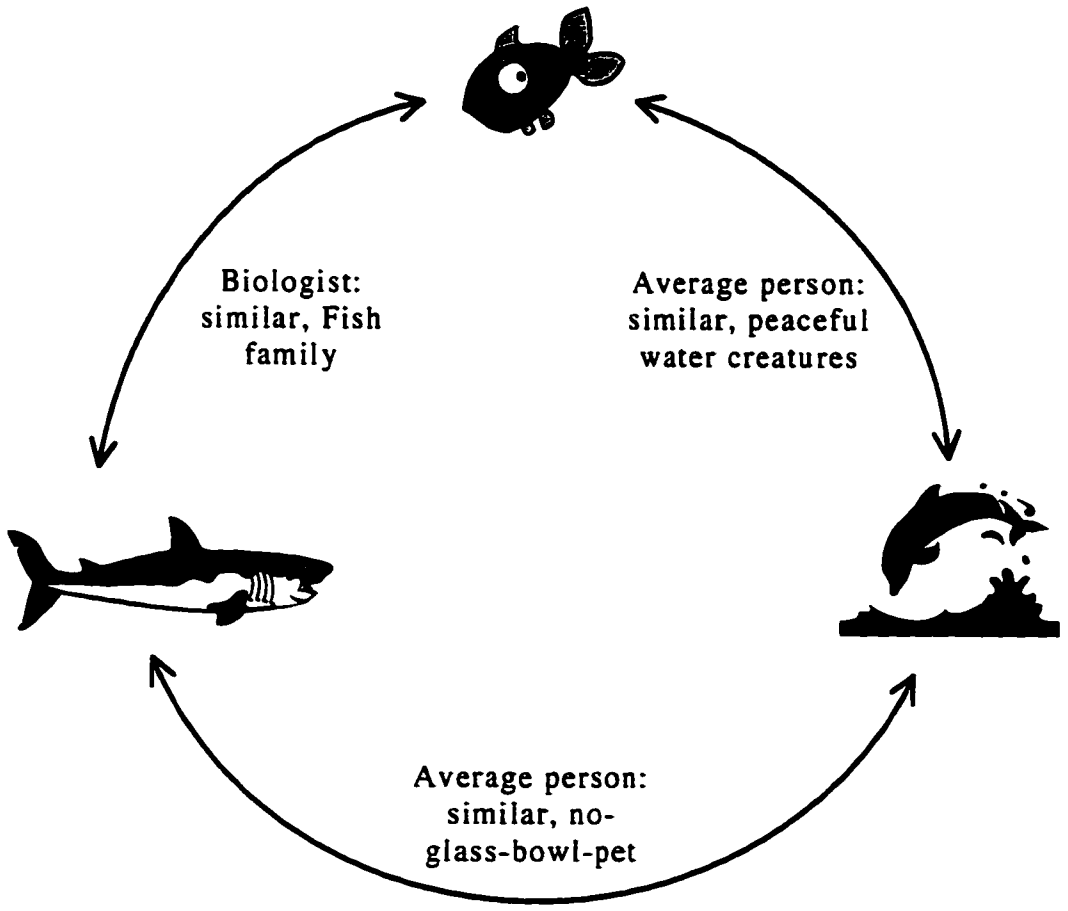


Figure 3.4: The Subjectivity in Similarity Decisions

3.5.1 Extracting keys: Feature Vectors

Images of 2D or 3D objects could be digitally represented and stored as a mathematical representation using a multidimensional matrix or CAD tables or other digital forms. However we want to extract a key for this large object to satisfy the main properties of a database key: smaller size while still having information about the contents. We will consider two examples for 2D and 3D images.

For 2D color images, for example, the SR-tree [KS97] develops a criterion to extract a color histogram to be used as an efficient key. It divides an image into 4 regions from upper left to lower right. For each region, munsell color space (hue, saturation, intensity) are measured and quantified into nine basic colors, giving a color histogram for the region. The four histograms are concatenated to give a 36-dimensional feature vector, which is then reduced to a 20-dimensional feature vector (since higher dimensions dramatically increase search time, making queries slower). Each image will have a 20-dimensional key (feature vector) representing its contents. The key is much smaller than the image itself.

For 3D images, Examples of 3D-protein database [AKKS99], [KKS98] divide 3D images of complex-shaped protein molecules into 3D cells (concentric shells of a sphere, sectors of a sphere, or a combination of both). Assuming that protein images are given as sets of points in 3D space [SK95], the shape histograms are determined by counting the number of points within each cell. This gives a feature vector for each protein image that represents its shape. This feature vector, again, is much smaller than the original protein object, and represents its contents, so we can use it a key when building a database for the 3D protein images.

Feature vectors can also be text or keywords about the images. For example in journalism, we can generate a feature vector for the thousands of pictures taken daily by giving them numbers to represent the date and place they were taken, the contents (people, objects, landscape) and the subject (politics, celebrity, fashion, sports) and so on. This could generate

the feature model for pictures. We can also build a hierarchy of these keys (feature models) by defining criteria to combine groups of keys into a super key that represent them. Algorithms can then be written to generate these keys.

3.5.2 Comparing Keys: Distance Functions

Feature vectors give compact digital representations of images. Nevertheless comparing two feature vectors as keys is not a straightforward application. As we can see, we do not look for exact match because of the nature of digitizing images errors, and contents differences. Different criteria have been developed to decide how similar two n-dimensional feature vectors are, such as the Euclidean distance in SR-tree [KS97] or the quadratic distance function explained in [AKKS99].

3.5.3 Searching the Database: Query Processing

After developing a criterion to extract keys and a criterion to compare them, we can use them to query the database to find similar objects. Goals can differ from one application to another, or within the same application. For example in biomolecular databases (like 3D protein images), a basic task is classification of new molecules [AKKS99]. In molecular biology, there are already defined classes for molecules. So after a new molecule has been discovered, we need to determine its class. First we look up the database and find the closest neighbors to it, then domain experts can decide on the exact class by comparing the new molecule against the similar ones in the database with more complex classification criteria than just the 3D shape. In this case, efficient classification algorithms can be used as fast filters for further investigations by the experts.

In other 2D, 3D, or color image applications for example, similarity search can use other algorithms to lookup similar pictures for fingerprints, or for news archive image search.

3.5.4 Building an Index to Speed up Query Processing

Comparing feature vectors to determine the distance typically include a large amount of calculations, which can be a heavy load on the processor. Some search sessions can take overnight to finish, so research has been done in the area of reduction of dimensionality of feature vector.

Searching a database in parallel can some times speed up the search.

As the dimensionality of the feature vector increases, or the complexity of distance function increases, or both, the time for an evaluation of two vectors can significantly increase. We can avoid many unnecessary feature vector distance evaluations by using an index as a pre-filter. Multi-step keys can be used to eliminate large number of objects without the expensive full-evaluation of their feature vectors, thus building a multi-level hierarchical index to the database. The idea is to cluster a group of similar keys (feature vectors) together into one super key working as a pre key (pre filter) to all members of the group, and apply this concept recursively to build a hierarchical index. [BBBK00] explains the difference between hierarchical algorithms and partitioning algorithms. Hierarchical algorithms decompose the database into several levels of nested partitions (clusterings), whereas partitioning algorithms construct a flat (one-level) partitions, each of which contains similar database objects. In this paper, a high performance clustering concept based on the similarity join is introduced to help build the key of keys by efficiently clustering a group of similar keys.

Another approach to speed up the search is to use a parallel similarity search [BEK+98], [BBB+97]. In parallel search, data is declustered (by uniformly distributing it over different disks) and thus the time for accessing and processing can overlap. [BBB+97] explains the importance of efficiently distributing data among the available disks and shows how this can speed up the search and introduces some efficient declustering algorithms.

3.5.5 Traversing an Index

Access methods used to traverse a multi-dimensional database in similarity search domain have many more variations than the equality-based database domains. [GG98] gives an extensive overview of multidimensional access methods. As explained before, indexes are built by recursively clustering data together and creating keys for the clusters. The higher level keys cover larger clusters and are thus coarse-grained compared to the lower ones. Traversing the index will produce different similarity distance at different levels, which may result in moving up and down the index structure as these values change, in order to locate the nearest neighbors. P. M. Aoki [Aok98] gives an example obtained from the SS-tree [WJ96] to demonstrate this feature. The SS-tree organizes records in hierarchical clusters. Each cluster is represented by a centroid and a bounding sphere with minimum radius to cover the elements within a cluster as shown in figure 3.5 in centers c11, c12, c13, c21, c22. Each group of spheres can be further combined into a higher level cluster (sphere) of one centroid representing the weighted center of mass and a radius to cover the elements in the group. This is shown in figure 3.5 as c1 and c2. As we search a key using a query, we can represent the search process as searching the closest (nearest) neighbor data (shown as black spheres) to our element key (shown as x in the figure). Since this is a multilevel index hierarchy, data elements (black spheres) are not directly exposed until we reach data levels. At higher levels of the index, only hierarchical keys (the hollow spheres) are exposed. In the first step, we compare our query to the high level clusters centered at c1 and c2. c2 looks closer so we descend to the right subtree (link b) at second level in the index. We retrieve the c21 and c22 clusters. At this point, both c21 and c22 are further than c1, i.e. c1 is now the nearest to our query of all c1, c21, c22, so we abandon the subtree of c2 (link b) and jump back to the subtree of c1 using link a. Descending on the subtree of c1 yields c11, c12, c13. As we measure the distance, we find that c13 is the nearest

neighbor. We can now access the three data components of c13 and find the nearest of them to our query x. From this example it is clear that we may need to go back and forth within the tree levels to get to the nearest neighbor. The example also depicts that as we get to a lower levels (like c21, and c22 level) the closeness (proximity) of clusters differ as they become more exact that the coarse-grained keys at higher levels (c2). That is why c21 and c22 appeared further than their super key c2 promised when comparing it against c1.

Similarity search algorithms use different distance evaluation techniques to determine the distance between two objects, which will indicate how similar they are. Less values for distance will imply higher percentage of similarity, with zero distance considered as 100% similar.

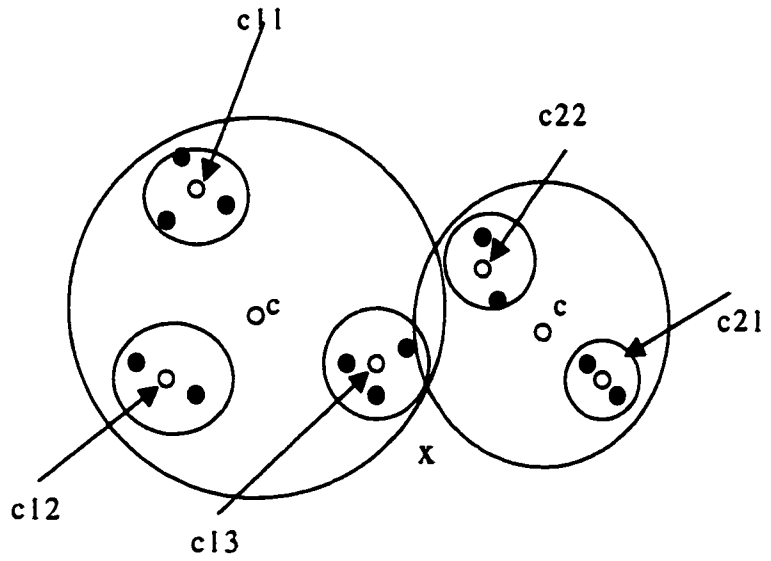


Figure 3.5: The SS-tree in Spatial Presentation

Chapter 4 Context Analysis

Database is manipulated in different ways depending on the user. In this chapter we concentrate on the different users and the context of using a database system.

4.1 Context Use Cases

Figure 4.1 shows the main actors of a DBMS.

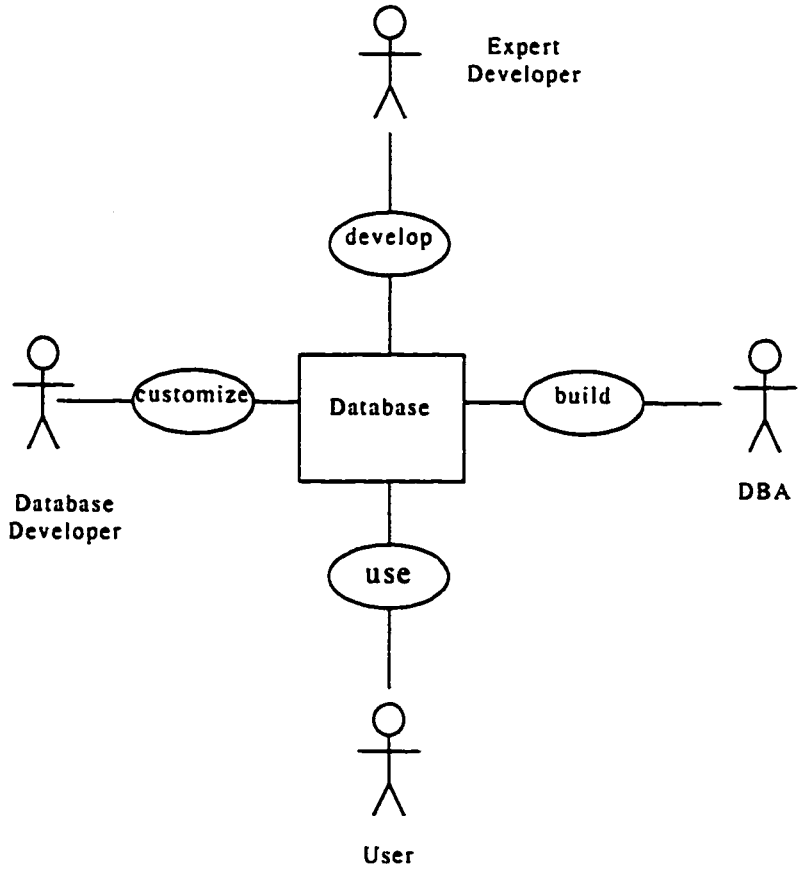


Figure 4.1: Context Use Case Diagram

4.1.1 Actors

Expert developer

The expert developer is an experienced programmer who designs the database system to satisfy domain experts needs and then implements the design using a programming language.

Database developer

Database developer is a knowledge domain expert, responsible for customizing an existing database system by modifying some of its parts (components) or by replacing some parts with others to be able to support a new data or query type, or support a completely new index structure with new access method. Database developer is also responsible for setting the parameters to suit the system platform and the application variables.

Database administrator

Database administrator is the person responsible for building the Database system, which include building the scheme, the physical tables and the indexes used to access them.

User

User is the software that is using the system to search for, insert or delete data from the database.

4.1.2 Use Cases

Develop Database

The expert developer puts a complete design for a database system and implements it to run on a platform.

Customize Database

From the extension programmer's point of view, part of the database can be customized to support new data types and/or query types. New index structures supporting new access methods can also be produced by modifying few parts of an existing index that has a different structure.

Build Database

The DBA is responsible for building the database by using a DDL language to define the scheme, and by defining the physical data tables, storage formats, and data partitioning, and building the indexes needed to access them in appropriate index files.

Use Database

This is the main use case that concerns the application, while the previous use cases are invisible to it. It involves connecting to an existing database and either looking up some data or doing some transaction (insert/delete) on that data using a query language.

4.2 Detailed Use Cases

In this part we will consider the index as a part of the database with the following properties:

- It is built as a balanced tree structure
- It can be customized to suit different applications
- It can be used to lookup data in the database (read-only access)
- It can also be used to insert or delete data in the database (read-write access)

Figure 4.2 shows the diagram for detailed use cases.

For more details on the index, see section 2.4, "Database indexes".

Design database

The expert developer puts a complete design for a database system to satisfy the functional need of database domain experts as well as the non-functional needs like storage, and retrieval issues, platform mounting, etc.

Implement database

The expert developer implements the design including all details such as the physical access details and the platform-dependent details.

Provide new query

The database developer defines a new query and passes it to the existing database to be used in searching the existing data.

Provide new data type

The database developer defines a new data type by writing an index implementation to produce a new index capable of handling the data type used by a new application. This includes defining suitable keys to describe the data partitions, and defining ways to compare them, setting a suitable layout for the node pages and a page policy for adding/deleting this new data type keys inside them.

Define new access method

The database developer defines a new method to traverse the index by replacing some parts of an existing index in order to produce a new index that is using a new access method

Set index parameters

The database developer sets the index parameters, like the tree order, the page minimum fill factor, and the page size, depending on the system environment, hardware profile, and application variables.

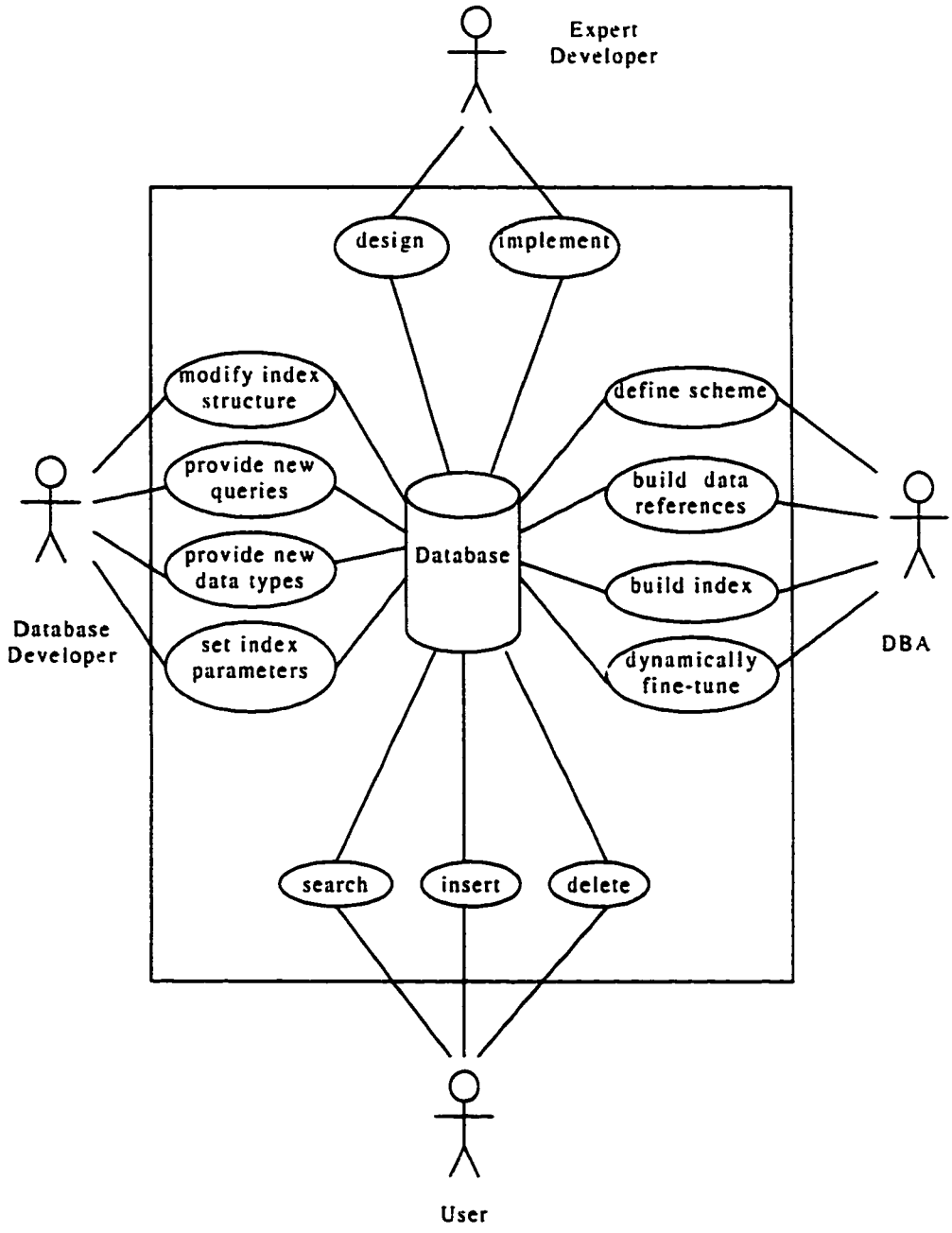


Figure 4.2: Detailed Use cases

Define scheme

The DBA will need to write the scheme describing the data tables.

Build data references

The data, once available, is prepared for the indexes by extracting suitable attributes to be used as keys in each index (primary / secondary) along with the corresponding physical location references. They are then sorted by their key value and in case of a secondary index further sorted by their physical location or by their primary key values, then put in the reference file as pairs of key, data reference.

Build Index

The DBA will build the index by successive insertions of the pairs of key, data reference into an empty index file (bulk loading the data reference file). A successful bulk loading operation will yield a complete index to the data.

Dynamically fine-tune

After the database system is up and running, the DBA needs to dynamically fine-tune it by slightly changing / adjusting its parameters to achieve the optimal performance under typical workloads.

Search

The application will connect to the database and use some query or a key to search for some data. An index will be used to lookup data. The index will return the matching data in the form of references to their physical locations. The references will then be used to access the data.

Insert

For an insertion, the search use case runs first to find a suitable insertion position. The position is then used to insert the data, and the index structure is adjusted if necessary to reflect the changes to the data.

Delete

For a delete use case, the search use case runs first to locate the matching data to be deleted. Then the candidate data is deleted. The index structure is adjusted if necessary.

Appendix A lists the flow of events.

4.3 The Database Data Model

The index is a major data component. Typically we have one primary index and multiple secondary indices, all built on the same data set. They help access the data using different keys/ queries for the search.

Each index is built on the physical data indirectly, through a data reference file. This reference file is composed of a key, data reference pair, and constitutes the data level of each index (see figure 4.3). This separation between physical data and data references allows for building multiple indexes on the same data set (data is not included in the index) and for the ability to change physical data formats without affecting the existing indexes.

In dealing with an index, we provide a data reference as an input (for insert / delete) or obtain one as an output (search). The data access component will then take the responsibility to bind the references used by the indices to the data tuples on the physical storage.

Depending on some system characteristics (like access time constraints, and system volatility) different binding mechanisms are applied (see book). The mechanism used must guarantee that changes in the tuple physical locations will still allow for all associated indices to access them correctly.

4.3.1 The Index Data Model

The index does not provide us directly with data, but with information about where the data is. This mean that the data stored in the index is

different from the typical database information (tables, records, etc). Figure 4.4 shows a graphical presentation of the relation between the index data and the database data.

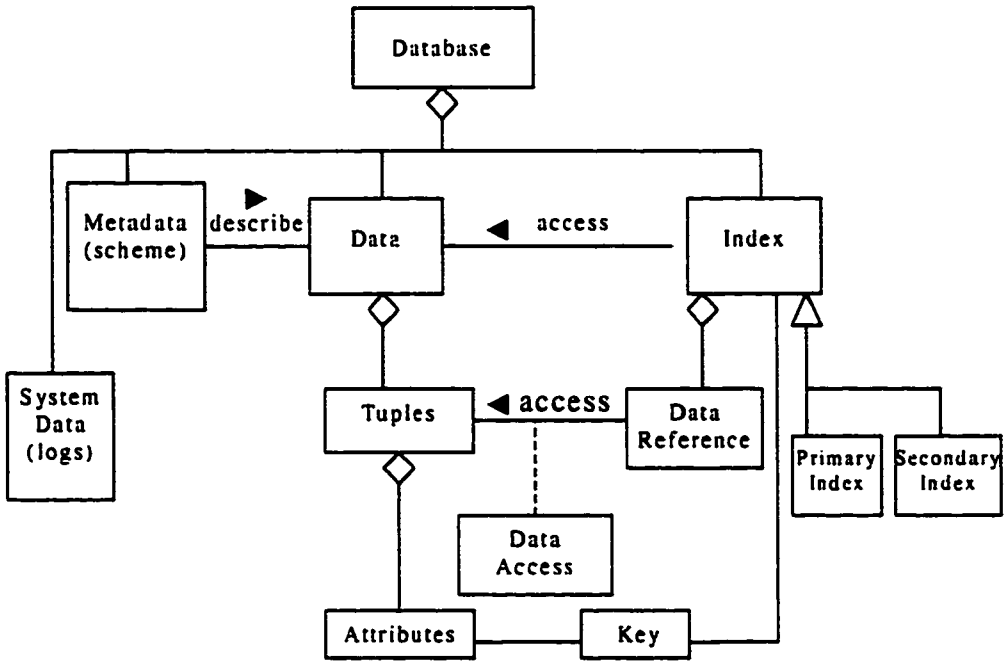


Figure 4.3: The Data Model

From this figure we can see that index data is references to physical data. We start querying the index using a certain key. With some comparison criteria, we try to find our way through the index down to the index leaf level, where the index data is stored. This will bring the index responsibility to an end. We should then refer to the physical database access mechanism to get the real data we are looking for.
Implementing an index goes the opposite way. We start with physical database data that we want to build an index for, build the reference file, then build the index using the reference file.

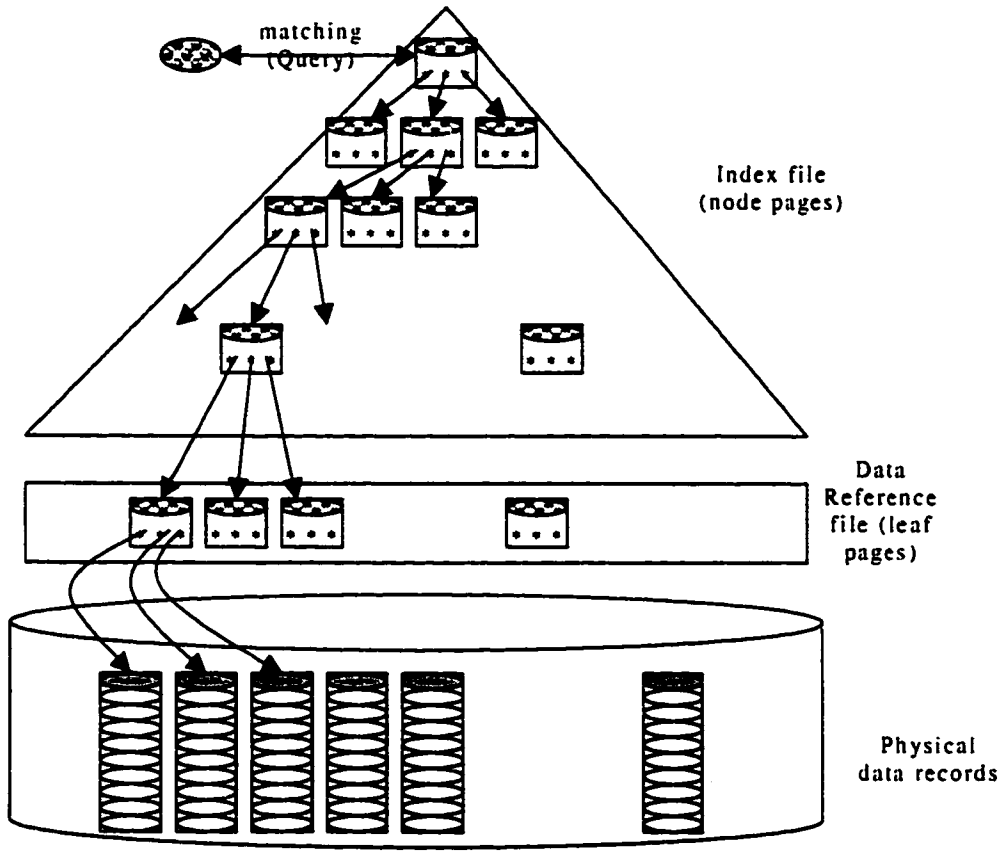


Figure 4.4: The Reference File Model

Chapter 5 The General Design

This chapter presents the design of a generic index system that can be adapted to different applications.

5.1 Design Rationale

System Modularization

The system is designed as an integrated set of modules. These modules provide all the necessary parts needed for implementing a complete system. Each of these modules, referred to as a building block, is well defined and has a clear set of functionality that is meant to carry out a specific task. This provides highly cohesive building blocks, a fundamental requirement for a good Object-Oriented design. Building blocks are also completely independent of each other. Each block can perform all its functions regardless of the types of the other blocks it is connected to, providing a very low coupling between these blocks, another fundamental requirement for a good Object-Oriented design.

Adopting the STL Concept

The STL, is a recent addition to the C++ language (1998) that supports good programming practices and provides a wealth of building blocks that can satisfy the needs of many sophisticated modern systems. We can build complete systems with the majority of its building blocks obtained from the STL library.

Code Reuse

A large number of building blocks already exist with a complete implementation on hand. This dramatically reduces the time needed for the implementation for many large systems where a great percentage of the code is simply imported from the STL.

Interface Reuse, and Combinatorial Composition

Each building block belongs to a group such as container group, iterator group, ...etc and these different groups are designed to carry out a different part of a system so that in the end we can have a complete system from these groups. The interface of these blocks is carefully designed to seamlessly connect to other blocks of the same or a different group, giving an endless number of possibilities of arrangements to achieve the needed system design. Some of these blocks do not perfectly fit together. This would imply that they cannot be connected or that their interface needs to be modified. This would, however violate the concept of generality and implementation-independence. Some extra building blocks are specifically designed to solve this problem in an elegant way. Without changing the standard interface of any block, some incompatible blocks can be made compatible using adaptors, a type of building blocks designed to smooth out some interface incompatibilities, thus increasing the number of possible combinations between the blocks.

Implementation Independence

The system design was made regarding only the interface of all the building blocks without any implementation details. This implementation-independent design frees the system from any unnecessary constraints, needed only for a later implementation, producing a simple abstract design.

Efficiency

All the STL building blocks used are written with the most efficient implementation possible, therefore allowing the system itself to be efficient. The efficiency, as a fundamental issue in any new system, is addressed as a design goal of STL.

System Flexibility

The framework has a flexible design by adopting a complete building block replacement policy. No building block is made mandatory to the design. In other words all the blocks that make up the system are replaceable. This gives the designer the freedom to isolate and replace any one or more of these blocks with no- or minimal impact on the system. The design can be adapted to the needs of any application by simply changing some of the building blocks. This was made easy since the building blocks are highly decoupled.

Wide Rage of Complexity

Even that the available generic blocks cover a wide range of applications; other specialized systems can still be implemented when the existing blocks do not cover all their functionality. The user can add any specialized building block to the system to replace an existing one, provided that the new block has the same interface. This allows for a minimal impact on the system since the user-defined block will seamlessly integrate with the other blocks eliminating the need to do a completely new system.

Default Values for Simplicity

In order to keep the system simple, only the necessary building blocks are exposed to the designer. Default building blocks are placed automatically in the system to do some of the necessary work without exposing them to the designer. Only when the designer wishes, are these default blocks replaced with other generic or even user-defined ones.

Favoring Templates over Inheritance

Inheritance is a relatively old concept in C++, compared to templates. Inheritance produces what can be seen as vertically-built or deep system,

which can, depending on the design and implementation, compromise the system run-time performance, a fundamental measure in database system. Templates, on the other hand, produce a horizontally-built or shallow system, which would, in general, have a better run-time performance than a comparable system designed with inheritance. Our design was built using templates extensively with minimal inheritance. This does not mean that the use of further inheritance is excluded from the system. It is always possible to inherit from the existing building blocks to produce new blocks with added functionality.

User-friendly, Readable Code

The system was made up of standard blocks. Each of these blocks has a clear standard interface and a well-defined functionality. This makes all the blocks easy to understand. Also new blocks may be added with the same look as standard ones. The code will then add up these blocks with some extra lines of code working as glue between the blocks, without masking their interface. This will make the code more readable that a specialized code that does not use standard building blocks.

Ease of Modification

For programmers, modifying a code written by someone else (or even by themselves after some time had elapsed) can be a nightmare, especially if the original code was poorly designed and/or insufficiently commented.
We opt for ease of modification for any design using the framework by having a completely modular system design. This allows any programmer modifying the system to focus on the modules to be changed without the need to understand the details of the whole system.

Sustained Code Quality

A neatly designed and implemented code can quickly deteriorate in quality after few patches and modifications.

The system modularity allows for an endless number of modifications and patching by simply replacing some blocks in the system without affecting the system quality. The system quality will not deteriorate by adding scattered patches all over the code because modifications are mainly done by replacing modules, keeping code quality the same during its lifecycle.

5.2 System Structure

5.2.1 The Basic Components

The system will be built using STL components. These components will provide the necessary index structure, and manage the access and storage of the index.

The index is a container that provides an iterator to its contents. The only way to interact with the container is through its iterator, which provides controlled access to the elements of the container.

The index container will be an index of pages, so the elements that the index stores are of type page. They are passed to the index as template type in the implementation phase. This makes the index independent of the page design and can work with any page type.

The index iterator is therefore iterating through pages, one page at a time. This is exactly the database concept of indexes: *paged indexes* to facilitate access and improve performance.

The index uses an index allocator to manage the storage of its own elements, the pages. In practical application the index exists permanently on non-volatile storage, like a hard disk or other mass storage media since it is normally too large to fit entirely in memory. This means that the container will use a specialized allocator that takes the responsibility of retrieving the page from the storage into memory for access and controlling the different objects accessing the same page simultaneously. This will ensure data integrity by applying a suitable access and locking policy (pinning the page). After the pages have been modified, they also

need to be updated in the physical storage (flushing the page) by the allocator. A default, in-memory allocator is provided for simple applications where the whole index can fit in memory at one time. It is up to the system designer to use it or override it by providing a storage-dependent specialized allocator.

.

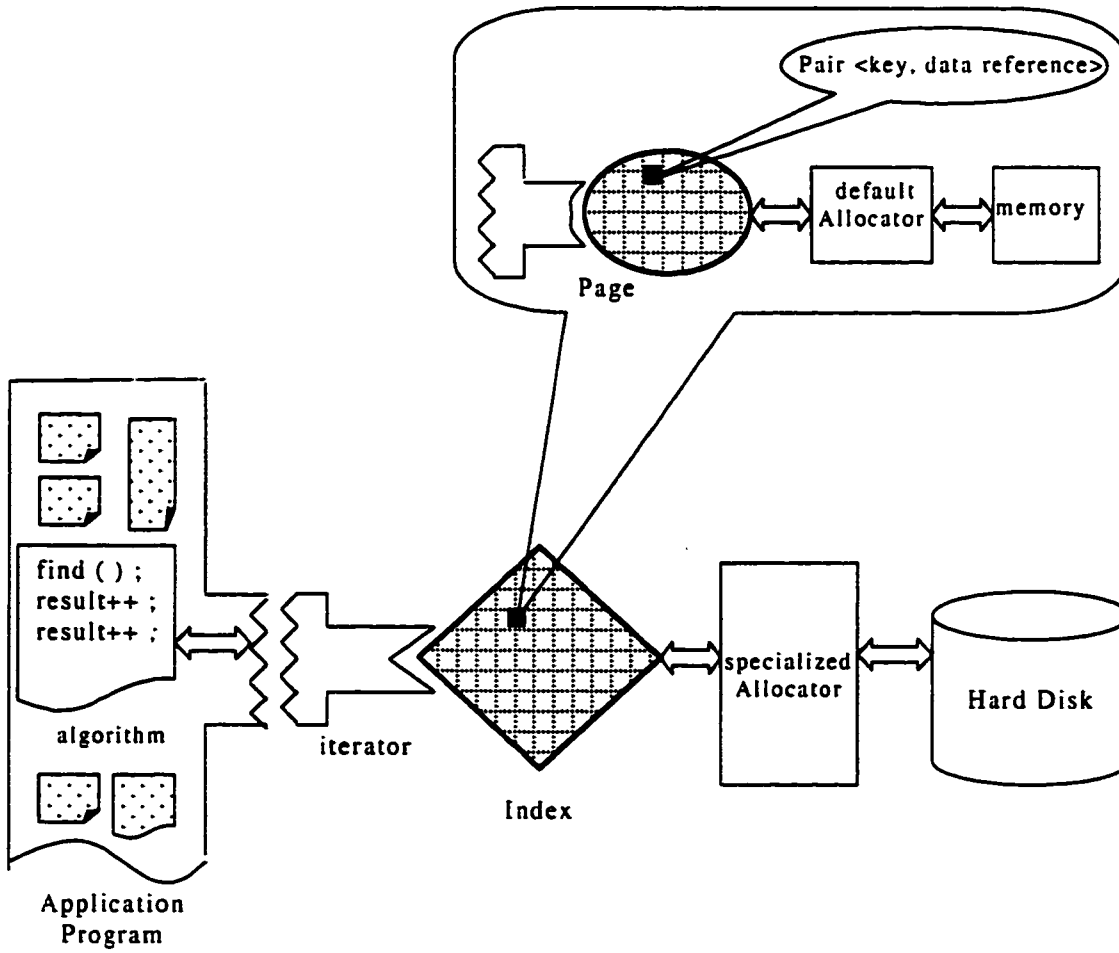


Figure 5.1: The Components Layout

This separation between the container and the allocator allows for the system to be completely independent of the physical storage of the index. The container uses the standard allocator interface to ask for storage services without any knowledge of the real storage dynamics

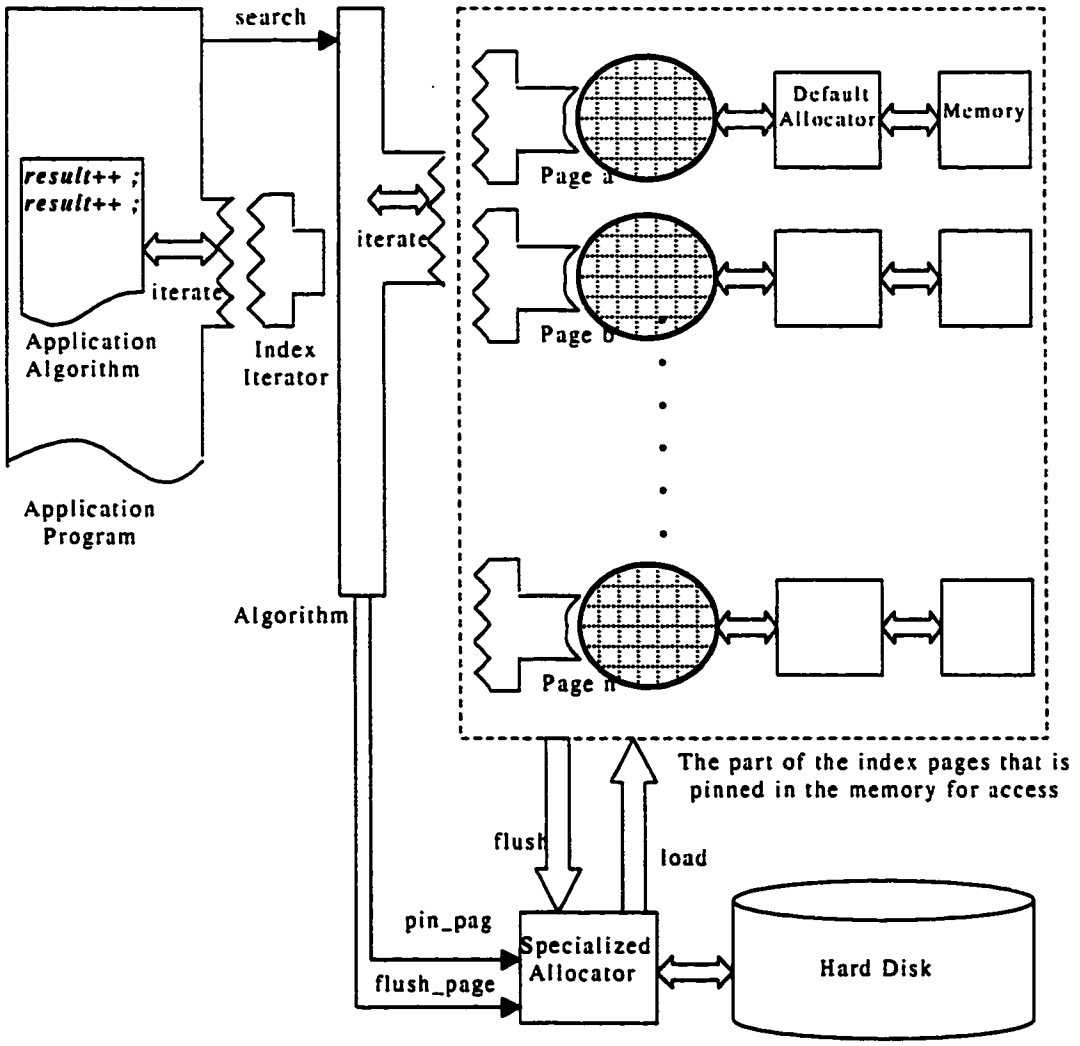


Figure 5.2: Detailed System Layout

The page is also a container on a smaller scale. While the index is a container of pages that typically resides on hard disk, the page itself is a container of pairs of key and data references. Again the page is made independent of the key and data reference type by having them as two templates passed to the page in implementation phase. Therefore the same page can work with any key type and data reference type that are passed to it. The page typically resides in memory for use. So whenever a page is needed, it is retrieved from the hard disk into memory. At this point the

page can perform its tasks of searching for a key in its contents, accepting new entries, deleting some existing ones, etc. This design produces a simple system structure without affecting its complexity level or extensibility. The basic modules are shown in fig. 5.2.

5.2.2 The Class Diagram

Figure 5.3 shows the class diagram of the index system

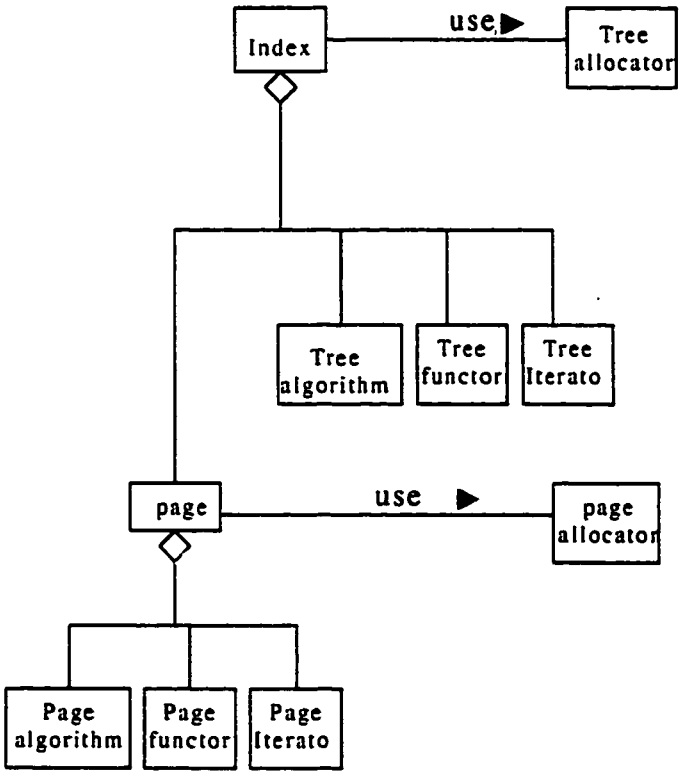


Figure 5.3: The System Main Classes

5.3 The Design Dynamism: Modules Replacement Scheme

The framework provides a modular design that can be adapted to different applications requirements. The resultant design is an index that helps an application access storage to get some useful information. Depending on the application domain and the type of storage, the system can vary in design. A simple layout will generate a system that uses its internal

components (the built-in set of functions) to operate. The application uses the internal algorithms (like find ( ) and insert ( )), which accesses the container elements via its internal iterator and apply the internal functor to them to decide on the candidate elements to a key provided by the application. The container uses a default allocator to manage data storage and retrieval.

Figure 5.4 shows this layout. As we can see all the components of the container class can be replaced by other external components.

In order to achieve this flexibility of replacement of objects without affecting the run time performance, an interface is provided for each of the system classes as in figure 5.5. The built-in classes are then an implementation of these standard interfaces. Any new class that is meant to replace an existing one must satisfy its exact interface, so it should inherit the necessary functionality of the original object by simply inheriting its interface. This is just an abstract inheritance that has no effect on the run time performance.

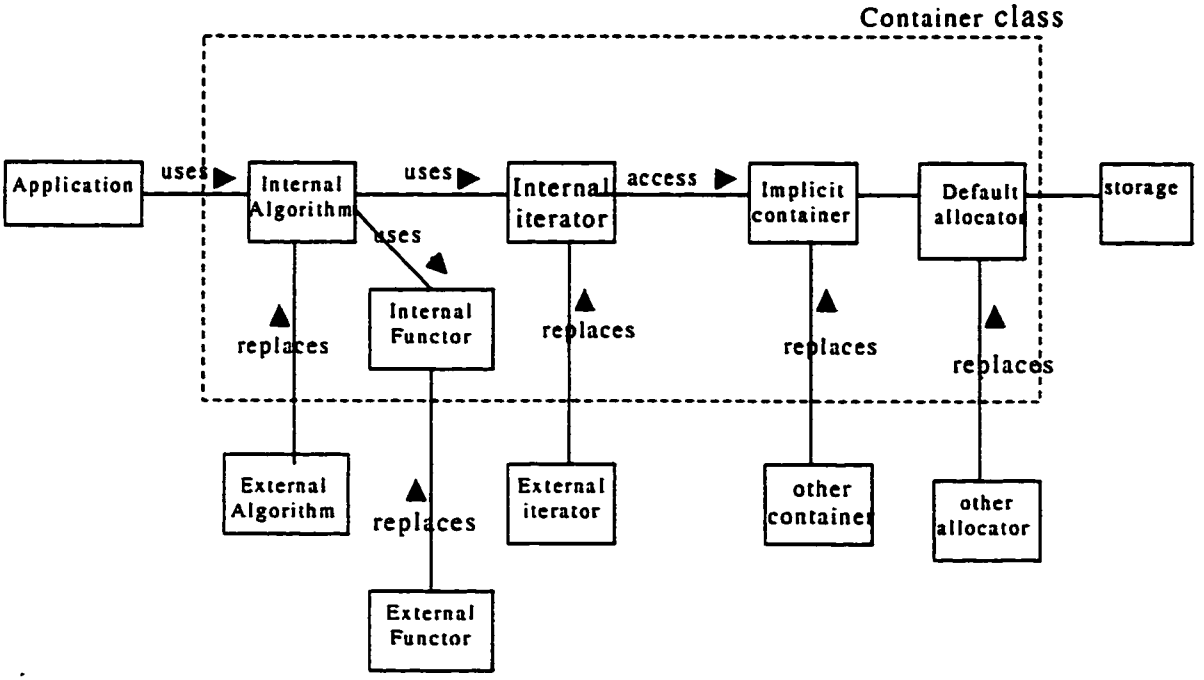


Figure 5.4: Components Replaceability

The basic interfaces of the container, iterator, and algorithm modules are shown in figure 5.5.

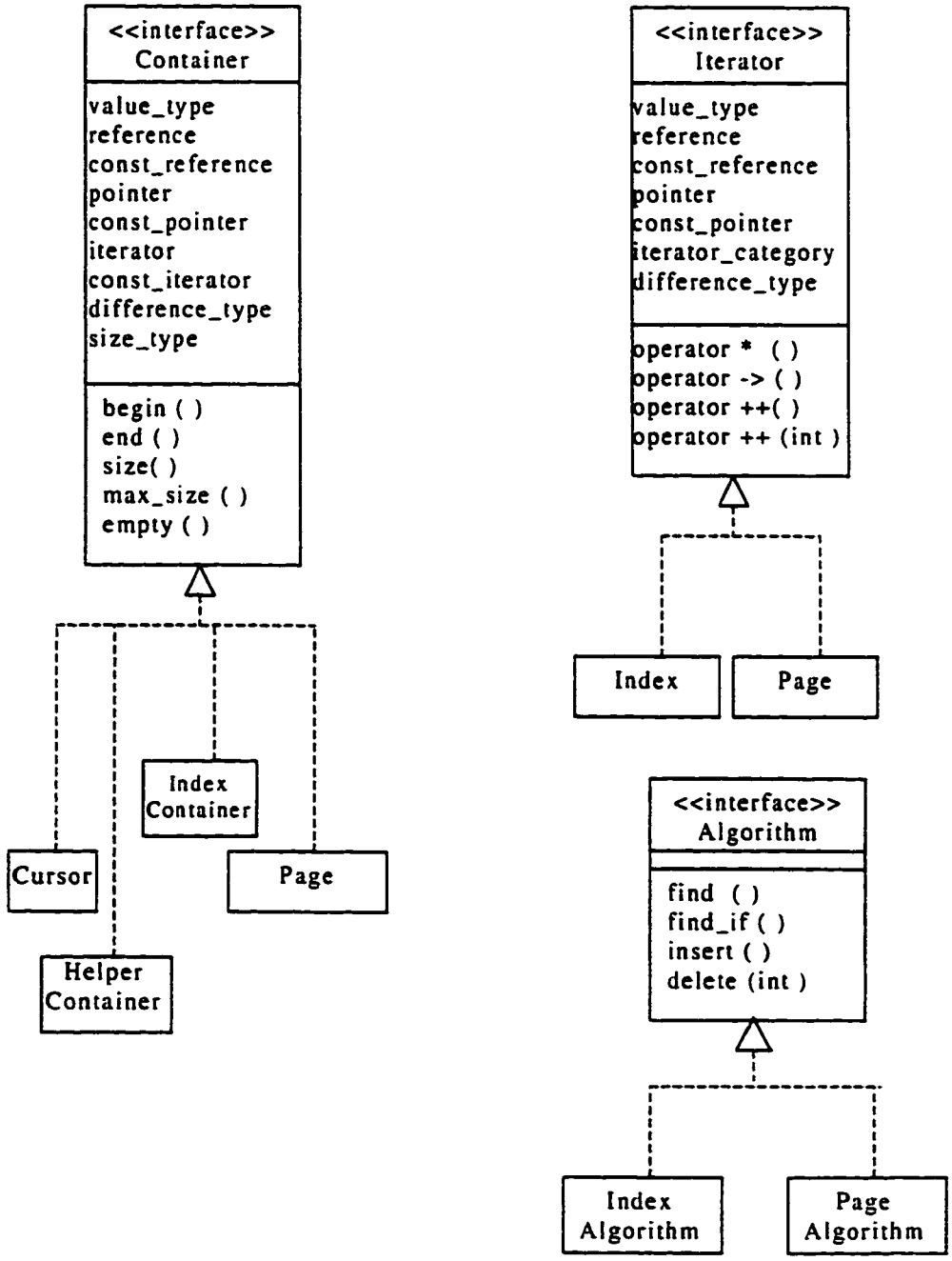


Figure 5.5: Detailed View of the Main interfaces

5.4 Exception Handling

The system manages part of its functionality through exceptions. An ordinary insertion operation would result in adding the new entry to the page and probably adjusting its key with no further impact on the system. Considering the fact that the index starts with its pages partially empty, the scenario of adding new entries will continue to the point where the page becomes full and thus it becomes necessary to split it. Assuming a comparable amount of deletion along with the insertions, the page will take much longer time before becoming full. Once the page becomes full during an insertion operation, the operation will throw an exception page_full as a signal. The index class will catch this signal and start the procedures of splitting that page.

A comparable situation can happen during a deletion operation. Typically a page starts only partially full, and can support some extra deletion operations with no impact other than deleting the entry and probably readjusting the page key. Adding to that the counteraction of adding some entries to the same page, the page size will fluctuate within the allowed range without impact outside the page. Once, however, the net amount of deletion to a page becomes dominant, the page size can shrink to below the minimum allowed fill factor (typically 50 %). At this point, during the critical deletion operation, the page will throw an exception page_sparse. The index will catch that signal and start the procedures to fix the sparse page.

One of the first steps to fix a sparse page is to try to borrow some entries from one of its siblings. Redistributing the entries of two pages equally between them will solve the problem with a minimal impact on the index. Only the two pages involved will be modified and possibly their keys at the parent nodes as well. If the two siblings of the sparse node were not suitable for borrowing, the operation check_to_borrow ( ) will fail and throw an exception none_to_borrow. The index will catch this signal and start the procedures of merging the sparse node with one of its siblings.

Figure 5.6 shows the signals received by the index class and the hierarchy of signals sent by the page class.

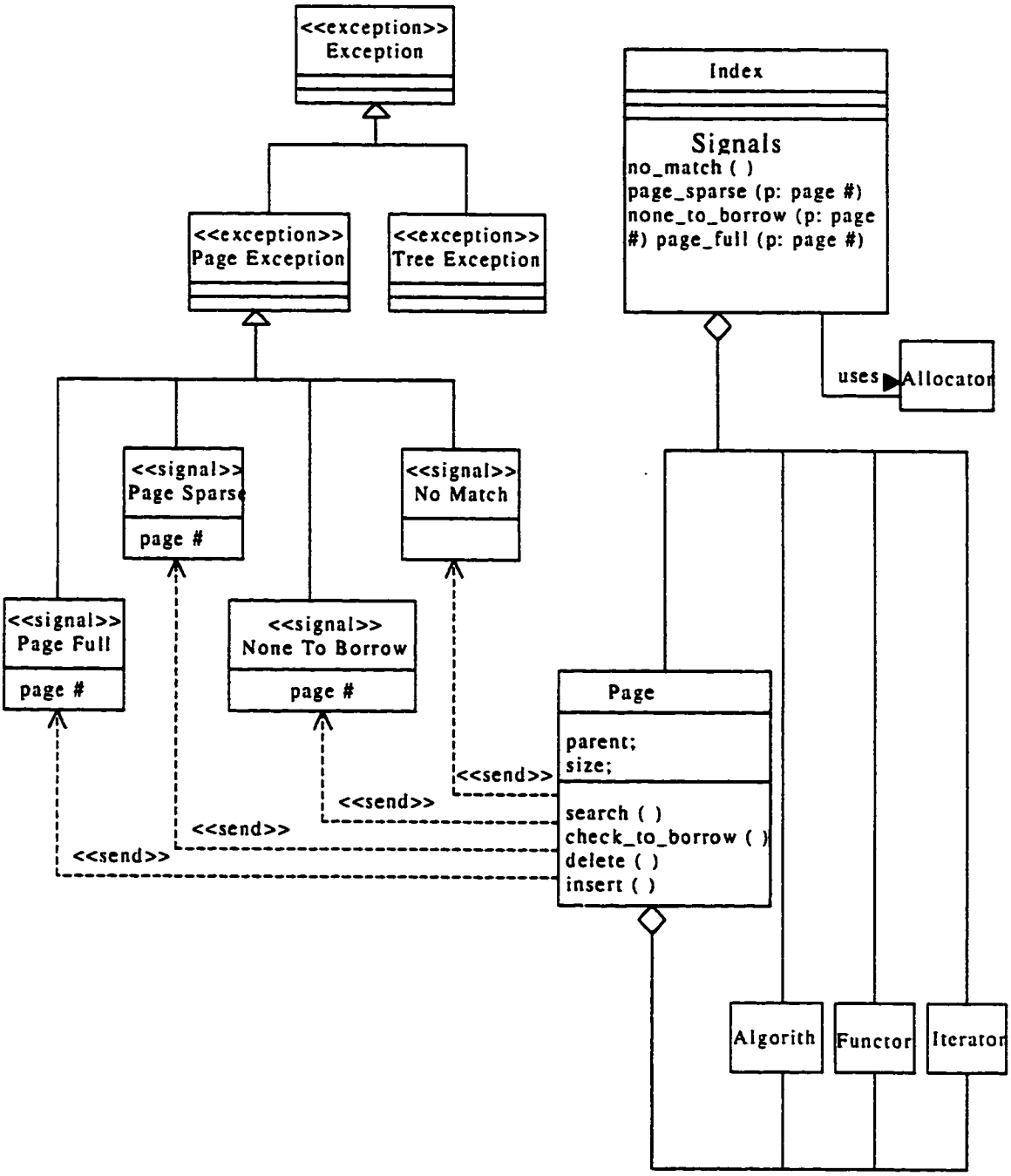


Figure 5.6: The Exception Class Diagram

5.5 The General System Interface

Page is an associative container that manages its data (page entries), which are pairs of key and data reference and allow for searching these pairs for a given key. Page is an STL-like container that must conform to all STL interface characteristics. The page container can be implemented using one of two implementation criteria: Interface realization or Implicit container inclusion.

Interface realization

Interface realization is implemented by inheriting the public container interface from STL container class and implementing the class methods completely. This will provide a more specialized code that is still conformant with the design. This is particularly useful when special time or space restrictions might apply.

Implicit container inclusion

For efficient implementation, the page container can be built on top of an existing STL container. The page will work as a wrapper class that has an STL container class as one of its data members (an implicit container). This will allow for masking the unneeded member functions in the STL class, and will save a lot of implementation time. The page flexibility will depend on the container used. This will give a more generic code.

In both cases we then need to add the needed functionality of the page. Many STL containers can be used as an implicit container, but we consider map to be the most suitable one for the following reasons:

i- It supports any user-defined keys (sequential containers can only support integers as keys)

ii- It supports unique keys (multimaps support multiple keys, if needed)

iii-It stores data entries as key and data (sets store keys only, if needed)

iv-It has efficient storage and search time

All containers provide their own public functions (built-in algorithms like find ( )). They also provide public iterators and type definitions to allow for interaction with external STL algorithms like find_if ( ) or any new user_defined algorithm. Depending on the search algorithm of the index, we can adopt any of these searching policies.

-Each page keeps a reference to its parent page if depth first with no stack were used as traversal method or for tree maintenance purpose.

-If tree supported sequential access (like B+ tree), leaf pages will have references to both its left and right siblings as well.

-Each page has a method to extract its key (the page_key( )) and – possibly- to save it as a data member for efficient retrieval of it with some storage overhead. As with other computer science issues, it is a compromise between efficient dynamic behavior and efficient storage (time and space).

-The page will receive pairs of <Key, page &> as entries to store within the page.

-The page is capable of adjusting its page_key after an entry insertion.

5.5.1 The Page Container

The page container is invisible to the application. It is created and managed by the index container. The page stores a pair of (Key, T). Class T can be any user-defined type, typically a reference to another page. Class T will be referred to as the mapped object of type mapped type. The pair (Key, T) will be referred to as the value object of type value type.

Key is used to distinguish the different pairs. It can be any user-defined type. Key does not have to be part of the mapped object T, extracted from its contents, it can be added to it. Keys are unique (other containers allow for duplicate keys).

Compare is the functor used when searching objects to decide on the matching keys. The default values allow for simply using the built-in comparison methods as built-in queries, or adding external functors as new user-defined queries new matching criteria. The page provides its built-in set of functions (as standard algorithms), type definitions and iterators to its contents to interact with external algorithms and external iterators. Insertion / retrieval takes logarithmic time when using the STL map as an implicit container in the implementation because objects will be stored in the page in a tree structure. This has, however, no big impact on performance since the whole page container is meant to fit in memory as one page and is typically small (on block size) so the retrieval time is insignificant compared to disk access time. Figure 5.7 shows the page class. Appendix B shows some public and private members of class page.

```
template <class Key,
          class T,
          class Compare = less <Key> ,
          class Allocator = allocator <pair <const Key, T> >>
class page;
```

Figure 5.7: Page Class Signature

5.5.2 The Leaf Page Interface

Figure 5.8 shows the leaf page class. Leaf pages are different in the data type stored inside them (class T). They also have a true value for the private flag is_leaf, which is accessible through the constant public function is_leaf ( ).

```
template <class Key, class T,
          class Compare = less <Key>,
          class Allocator = allocator <pair <const Key, T> > >
class leaf_page: public page<Key, T, Compare, Allocator>;
```

Figure 5.8: The Leaf Page

5.5.3 The Index Tree Container

The index is an abstract data type that can use different implicit data types in its implementation. As with the page design, the index container functionality can be either inherited from an STL container interface and undergo a complete implementation, or can be implemented by using an STL container as an implicit container in a new index container class. We follow the second approach. The index container provides methods that use some of the implicit container methods, but this is transparent to the user. The page container is meant to be small enough to easily fit into memory (a fundamental concept in the index page). The index container, on the other hand, does not have to fit completely in the memory. In typical cases it will not. The characteristics of the index container will decide on the implicit container used. Since index exists mainly on hard disk, inserting new pages can only be made at the end of the index file, or by using a recycled page (for efficiency reasons). No support for insertion anywhere else is needed from the implicit container. For the same reason, the index uses a special allocator to manage the storage on the hard disk (pinning the page). The access of pages is required to be a direct access. Given a page number (or offset) we expect to be able to read it directly without the need to access all pages before it. Therefore the implicit container must support random access using operator [ ]. Sequential access to the index pages would result in unacceptable performance. This means that we have two choices. We can use a sequential container that supports random access to its elements (pages) using some index number to identify each page (by translating it to the physical page address). Alternatively, we an use an associative container that supports random access to its elements (pages) using a unique key for each page, (again by associating it to a unique physical page address).

The Interface

The incrementing operator implementation in class cursor might have different implementations depending on the index type

1- It can be as easy as reading the next entry from the private vector component if all results were deposited into the cursor in one batch as in general domain batch search (breadth first policy).

2- It might include sending messages to an index page to iterate to its next entry as in linear domain sequential access.

3- It might include sending a message to the index to continue searching and locating the next entry as in general domain depth first policy.

For class index, it is more likely to use the default container than the default allocator, so allocator comes first in the template argument.

Figure 5.9 shows the signature of the query, cursor, and index classes, all included in the index class.

Public type members and data members:

Each user-defined container must provide a set of standard STL types that are internally translated to the equivalent container-specific types with typedef statements. This allow for the container to be easily integrated with other STL components.

This can be accomplished directly as public type members or by providing a struct traits to define these types. We follow the first approach.

```
template <class T>
class query : public Unary_function <T >
{
typedef  page <Key, T, Compare, Allocator> :: key_type   key_type;
//the query includes a page key in it.

T  operator ( ) (T a, key_type k ) const;
// return value must be convertible to boolean

}

template <class T>
        class cursor
        {
        public:
                T  operator = ( );        // for index to deposit results
                T  operator * ( );        // for application to dereference
                                          // results
                T  operator ++ ( );       // for application to pre increment
                T  operator ++ (int ); // post increment
                begin ( );
                end ( );


        private

                vector <T> V

        } //end class cursor

template <class T,
            class Allocator = allocator <T>,
            class Container = vector <T> >
class index;
```

Figure 5.9: The query, cursor, and index Classes

Chapter 6 Design for Linearly Ordered Domain

Before describing details of system design, we will explore some properties of linearly ordered domain. This will clarify the design decisions made for a linearly ordered domain index.

6.1 The Concept of Linearly Ordered Domain

Data in this domain is stored in a sequential order. The data is at least Less-Than Comparable, meaning that it must be possible to compare two objects of that type using the operator < and the operator < must define a consistent order [Aus99], so it can be at least partially ordered. For example, for two elements A and B, we can decide which one is less than the other, and store them in their order. If we cannot decide, they are said to be equivalent and we store them in the same order as one group. So ordering here can be done only partially as for two equivalent elements, there is no specific order even that they are two different elements. But at least between groups, there must be a complete order. No two groups can be equivalent unless they are the same group.

An example is when storing information about a number of cars. We can order them according to their model year. For every two cars, we can say if one is older than the other and put them in order. If two cars have the same year model, we cannot order them according to this criterion. They are equivalent, even that they are two different cars, and we put them in one group.

Data can also satisfy the more stringent Strict Weakly Comparable concept, generating a total ordering for the data. This means that for two elements A, and B, we must be able to decide their order. They cannot be equivalent anymore as long as they are two different elements. Each element is equivalent to itself only, so we reduce the number of elements

per group to one and the equivalent concept becomes equal. An element is equal to only itself.

In the previous car example, ordering the cars by their license plate will give a total ordering. No two different cars can have the same license plate, and for any two cars, we must be able to decide which one is less than the other. For every group of license plate, we have only one car.

This implies that the search for a specific key will find a specific element (or a specific group of elements) that satisfies the search comparison criteria. This element (or the first element in case of a group result) will be returned as the result for the search.

In case of a group result, the beginning of the group is sufficient to give the necessary data. Since data is ordered, the rest of the group can be easily located. This mandates a small addition to the generic page concept: Each leaf page should be physically connected to the next leaf page. Even that pages are not necessarily physically ordered, it is easy to find the logical order of leaf pages regarding their contents. Then it would be easy to link each page with the page that has the immediately next sequence of data. In the design, each leaf page will therefore have a reference to the page that follows it in the sequence. Acting like a linked list data structure, inserting a new page between two pages requires some modifications to keep the logical sequence in order.

Linking each page to the previous page as well can make a further addition. It will then act like a doubly- linked list.

We not that leaf pages act like linked lists when it comes to sequentially accessing the data, But, unlike linked lists, they also provide the functionality of direct access when performing a search.

It is clear that the search technique can be composed of one single dive downward through the index, with no stack needed for the lookup process.

6.2 Class Diagram

For a linearly ordered domain, the index will have an index algorithm class to provide the index functionality (search, insert, delete, etc). This algorithm will be used by the application to demand services from the index. The index will have two types of iterations, implicit internal iteration, done without a real internal iterator object, and explicit external iterator; the cursor.

Implicit internal iteration

This iteration will provide a special kind of internal tree traversal. It will generate a page reference and simply access the page. No iterator range (begin, end) or functionality is needed. Generating a page reference will be based on one of the following tree iteration concepts:

Concept 1: Iterating using a key:

The iteration will start from the root page as the *begin ( )* position. The incrementing process *operator* ++ must be based on a search key; no arbitrary incrimination is supported since there is no specific next element without a search key. The next page in the index will be decided by interrogating the index current page using the key. This interrogation will provide the next element (a page reference), where the iteration can proceed. This will make it the new current element.

Concept 2: Iterating using a page reference:

The page reference can be obtained by different ways.

One way is by interrogating another page using a certain key. This case will be invoked internally by a search operation that uses concept #1 iteration without interaction with the application.

The page reference can also be obtained by interrogating a leaf page for the next page reference (in case of sequential access). This case will be called externally by a cursor iterator that is accessing the result data sequentially.

The page reference can also be obtained by interrogating any page for a parent page reference. This case will be called internally by an insert or delete operation that needs to fix a parent to a modified page.

All these iterations are invisible to the application. The only iteration done by the application is iterating through the result data using a cursor. The index provides the results to an external iterator (the cursor), which will then be able to iterate through the index leaf pages contents as well as iterating from one leaf page to the next to access successive results.

Explicit external iteration; the cursor:

The cursor is a special kind of external iterator. It is created by the application when searching for some data. As the index searches and locates the beginning of that data, the cursor can iterate through the successive entries (on request from the application). Here, unlike internal iterator, the next element is sequentially defined; it is the next page entry (a pair of key, reference). As the leaf page is depleted, the cursor is capable of jumping to the next leaf page to continue iteration. Here the next element is also clearly defined; it is the first page entry in the following page. The cursor will use the page iterator to go within a leaf page, and will use the index iterator to go from one leaf page to the next. This is invisible to the application. The application will be able to iterate through resultant data continuously as if they were inside a sequential container. The index will have pages as its stored elements. The index will use an index allocator to store the pages on an external storage medium that holds the index pages (typically the index size is too large to fit in memory in one shot). An in-memory allocator can be used if the index can fit in memory. The index will use an index functor (query) that will be passed to the pages to check for matching entries. This query is external to the page, but built in within the index and passed to the page to use it. This query will carry out the equality

search. So the linear domain index comes with a built in query. Any other query can be passed to the index by the application and be used by the page. The page will have page algorithms to decide on the policy of searching, inserting, and deleting within the page. An in-memory page allocator will carry out the memory allocation and deallocation operations. The page will have a page iterator to iterate through the contents of the page depending on the contents layout. The application that uses the index will use the algorithms provided by the index to build, search, insert, or delete entries. The application will have a cursor, which the index will be made aware of. The index will pass the results to the cursor, and the application will then access them from the cursor.

Note that some components are missing, like the page functor class for example, because they are not needed in this design. This does not, however, exclude the possibility of adding it to the design in a later modification.

6.3 System Behavior

In order to easily track the behavior of the system using sequence diagrams, we will use a specific concept for numbering the pages of the index. For simplicity, we will assume each page to have a maximum of 10 children (the same concept can apply to any number of children). The root page, at first level, has the number 0. The children of root page have numbers 00, 01, 02, 03, ... up to 09, where the first digit (the 0) is the parent number, and the second digit is the child number (0 through 9). Each of these pages also has children numbered by post fixing the child number to the parent number. For example, page 03 has children numbered 030, 031, 032, ... up to 039. This also means that the number of digits in a page number gives the level at which the page is. Of course, it is possible for a page to have less than 10 children, so for example if page 044 has 8 children (0440 to 0447), then 0448 and 0449 will simply not

exist. We will not use these numbers to implement a real index, so we will not consider unneeded details like a missing child in the middle between two children (for example as if 0355 was missing, the next page to 0354 would be 0356). We will simply assume that no missing pages exist in the middle. Each page will have a reference to it carrying its number and a key with the same number as well. For example page 0338 has reference P0388 to it and its key is referred to as key0388. From the page number we can get all the information we need; its immediate parent, its ancestors all the way back to the root, its siblings (if existed), and the possible children. For example page 05380 is the first child of page 0538, which is in return the 9^{th} child of page 053. Page 04256 has the two siblings 04255 (previous), and 04257 (next, if existing)

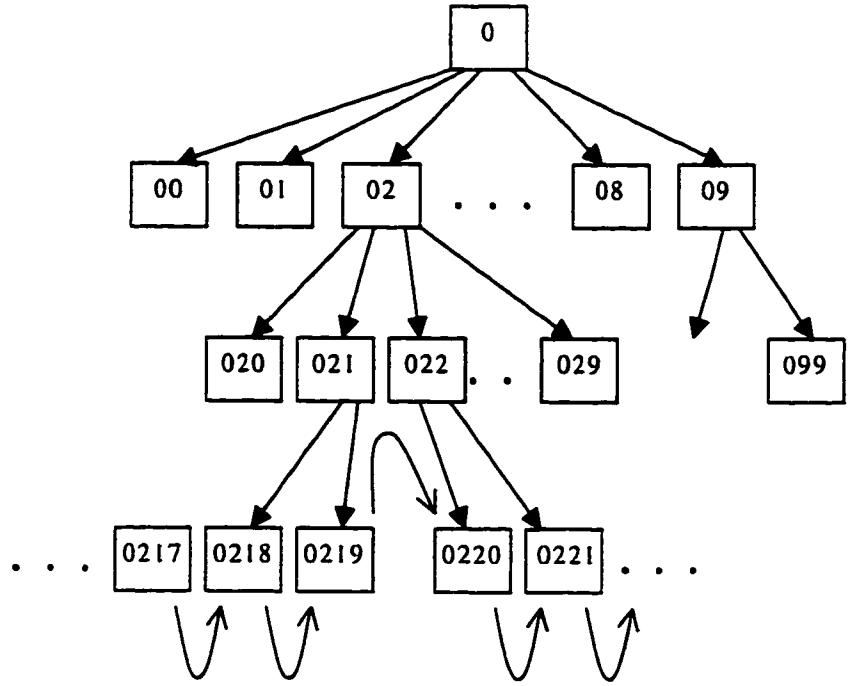


Figure 6.1: The Numbering Convention

Page 0219 has the left sibling 0218, while it has no immediate right sibling under the same parent (the parent would be page 021 with 10 children from 0210 to 0219). However, we can access the next sibling by

finding the next sibling of the parent (page 022) and access the first child of it (0220). We followed the tree paths, and did not just add (0219+1=0220). This concept, albeit pure theoretical, can simplify our tracking of the system behavior across the index pages. Note that all page numbers will start with 0 as the index tree has one root only, page 0.

6.3.1 Scenario for "search" Using an Internal Query

Search is the fundamental operation in using the index. The application uses the index to locate data by searching its contents to find where the physical data resides. As mentioned before, the index does not provide the data we are looking for, but tells us where it is stored, so the data returned by the index is typically a reference to a location. The other index use cases insert and delete also depend on the search operation. In order to delete an element, we must find its location first. Also to insert an element, we must search for the best location to insert it. For unique elements, we must search before inserting an element to make sure it does not already exist.

The search function comes in two fundamental forms: using a key or using a query. Search by a key is the basic form. It uses the internal comparison operator (the built-in functor) inside the index to find the first data entry whose key matches the searched key. As shown in figure 6.2, the application starts by (1) creating a cursor object to receive the search result in it. Then it (2) sends a search request to the index algorithm providing information about the key to search for, and the cursor to put the results in. The Index starts the search operation by creating a page algorithm object; this object has the semantics of searching a particular page for a key. From the STL concept, this object needs to be passed three things: a key, a page iterator, and a functor object. The tree creates a functor from the built-in functor class, and an iterator to the root page and passes them along with the key to the page algorithm object. This latter object will use the iterator to go through the root page entries, one at a

time. With each entry retrieved, the page algorithm will extract the key, and check it against the searched key. If the check returned false, the page algorithm discard this entry and repeat the same process on the next entry. Once an entry's key evaluates to true, the page algorithm will extract the data part of the entry (a page reference) and send it back to the index algorithm. The index algorithm will remove the root page iterator if it is not needed any more, create another page iterator to the page returned by the first page search. The Page algorithm object will then be passed the three parameters: the searched key, the iterator to the new page, and the built-in functor to check the retrieved keys for a match. The same process is repeated for all page entries until an entry that satisfies the key is found and its reference part is returned to the index. The index will keep retrieving the pages and interrogating them for further pages down the tree until a leaf page is reached.

As the leaf is interrogated for the key, and the data part for a matching key is found, the data part is (2.1) returned to the cursor object. The search carried out by the index comes to an end.

Now the result for the search is in the cursor (in the form of a page iterator pointing to the page entry that matches the key) and the application can (3) retrieve this result by (3.1) instantiating the iterator object using (*result) and incrementing the iterator to the next result using (iterator ++). As the cursor is incremented, it will (3.2) iterate to the next entry in the index. The next entry is (3.3) sent to the cursor and then (3.4) to the application. As the application iterates to the next result, another iteration cycle (3.5), (3.6), (3.7) takes place.

Figure 6.3 shows a high-level sequence diagram of this interaction. Figure 6.4 shows a detailed sequence diagram.

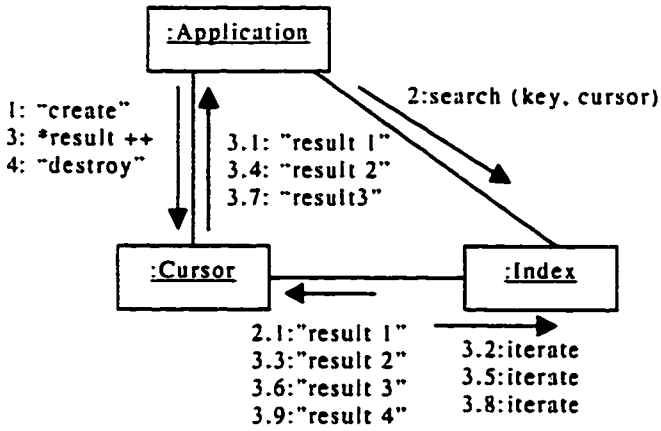


Figure 6.2: Hi-level Collaboration Diagram

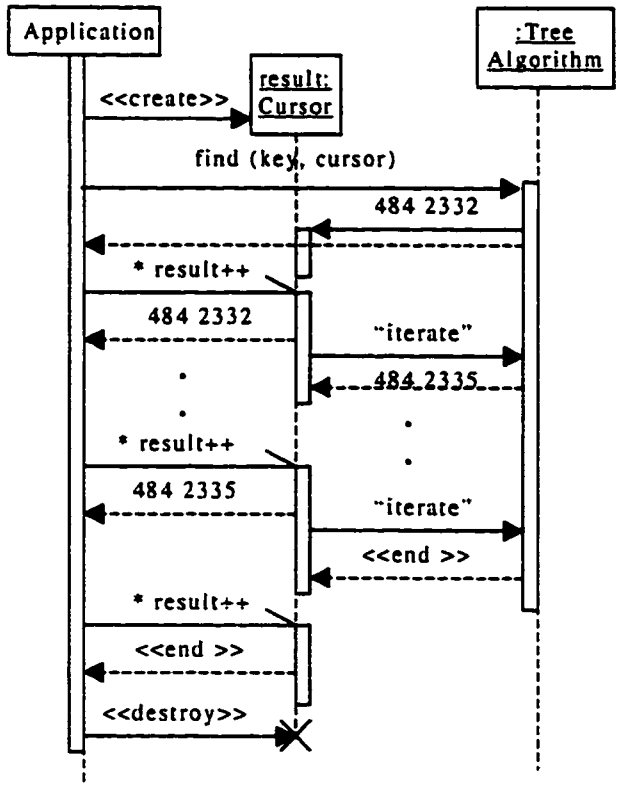


Figure 6.3: Hi-level Sequence Diagram

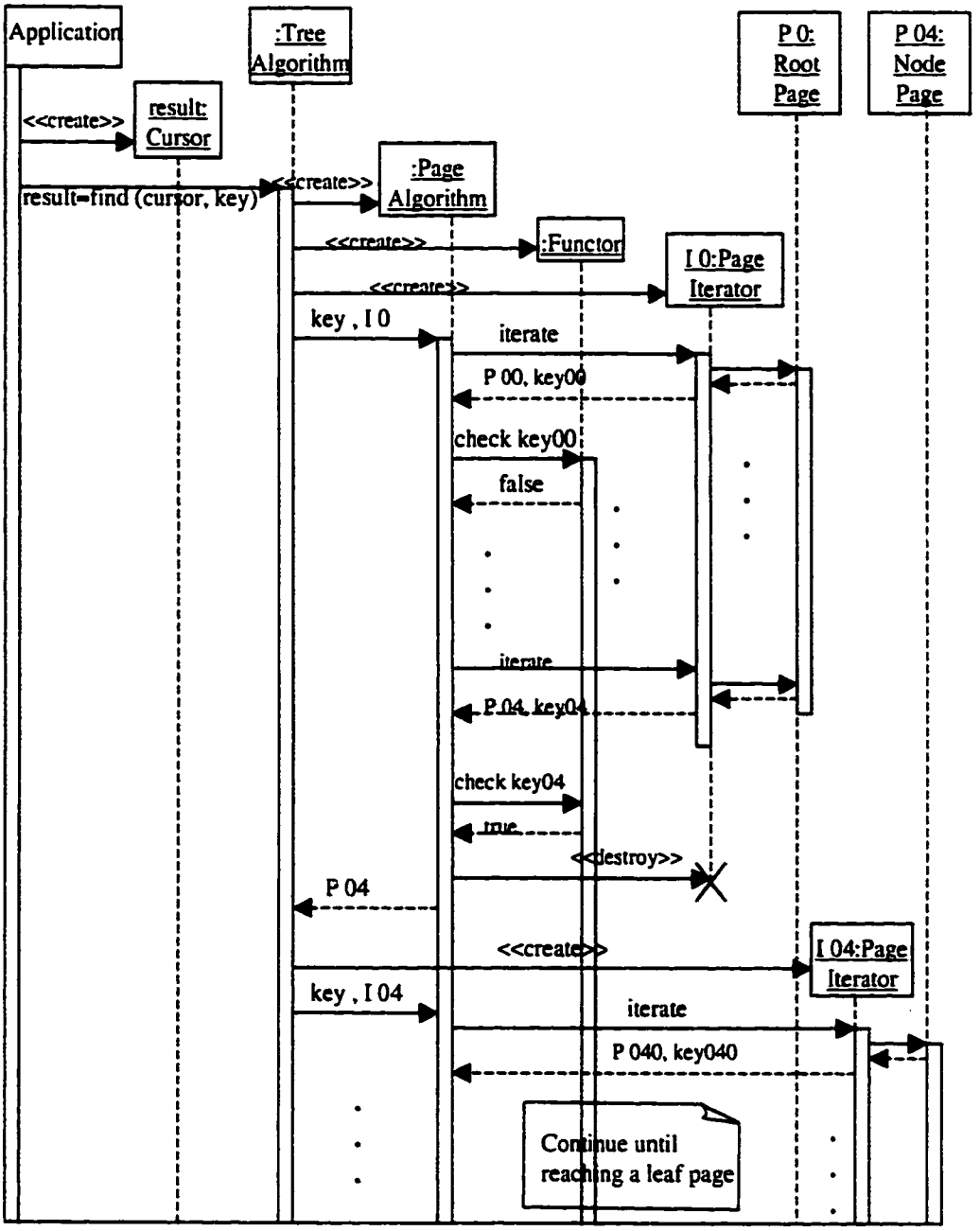


Figure 6.4: Using Internal Query

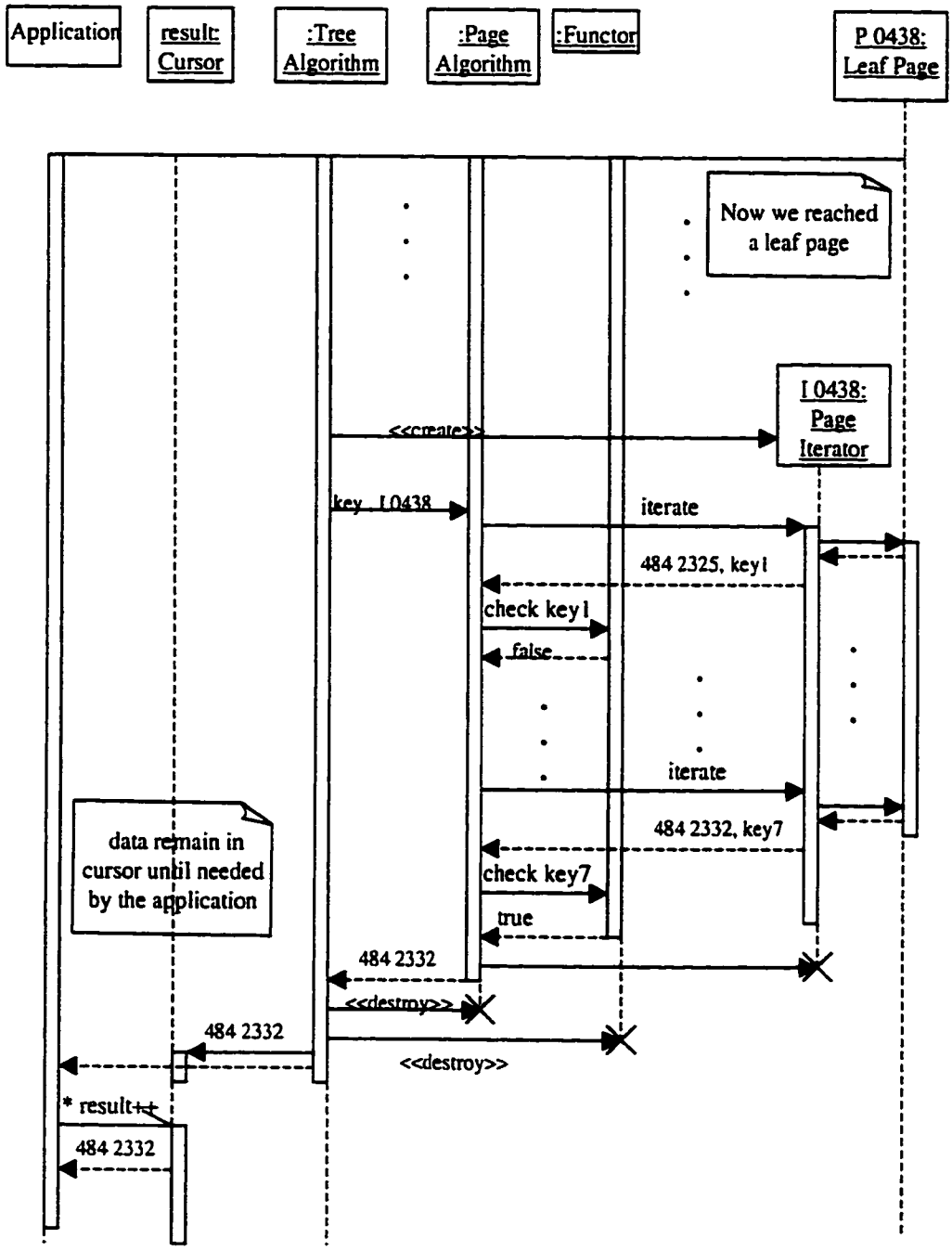


Figure 6.4 (cont): Using Internal Query

6.3.2 Scenario for "search" Using an External Query

We can use the index to search for a key using an external query instead of the internal one. No major change is needed. In database concept, search using a query is a search for certain elements that satisfy the query. All we need to do to the system is simple: pass to the page algorithm object an external functor that resembles the query, so that when the page algorithm retrieves page entries, it will check them against this external functor instead of the internal one. From the page algorithm's perspective, nothing much has changed. Only the functor used for the check is different. The scenario with these change reflected on it is shown in figure 6.5. Now the application creates the query object of its choice (it must, of course, conform to the standard functor interface by inheriting the abstract interface of a functor). The query object will contain the searched key inside it as well as the comparison criteria. It can check any key passed to it. It will be able to check the keys sent by the page algorithm and return true or false to the page algorithm. The index algorithm will make the necessary adaptation. As it receives a search request with a query instead of a key, it will create the suitable page algorithm object and passes to it the two objects: the root page iterator and the query object. The page algorithm will retrieve page entries as before, and check their keys against the external query, and select the matching ones. We need to notice one subtle difference between the algorithm of internal and external queries. For internal query case, the algorithm receives two parameters: the searched key along with the page iterator. It will then use the internal functor (the default one), extract the keys from page entries and send the two objects extracted key and searched key to be checked by the internal functor. For external query case, the page algorithm receives two parameters: the page iterator and the external query object. No key is sent as the information about it is now inside the query object.

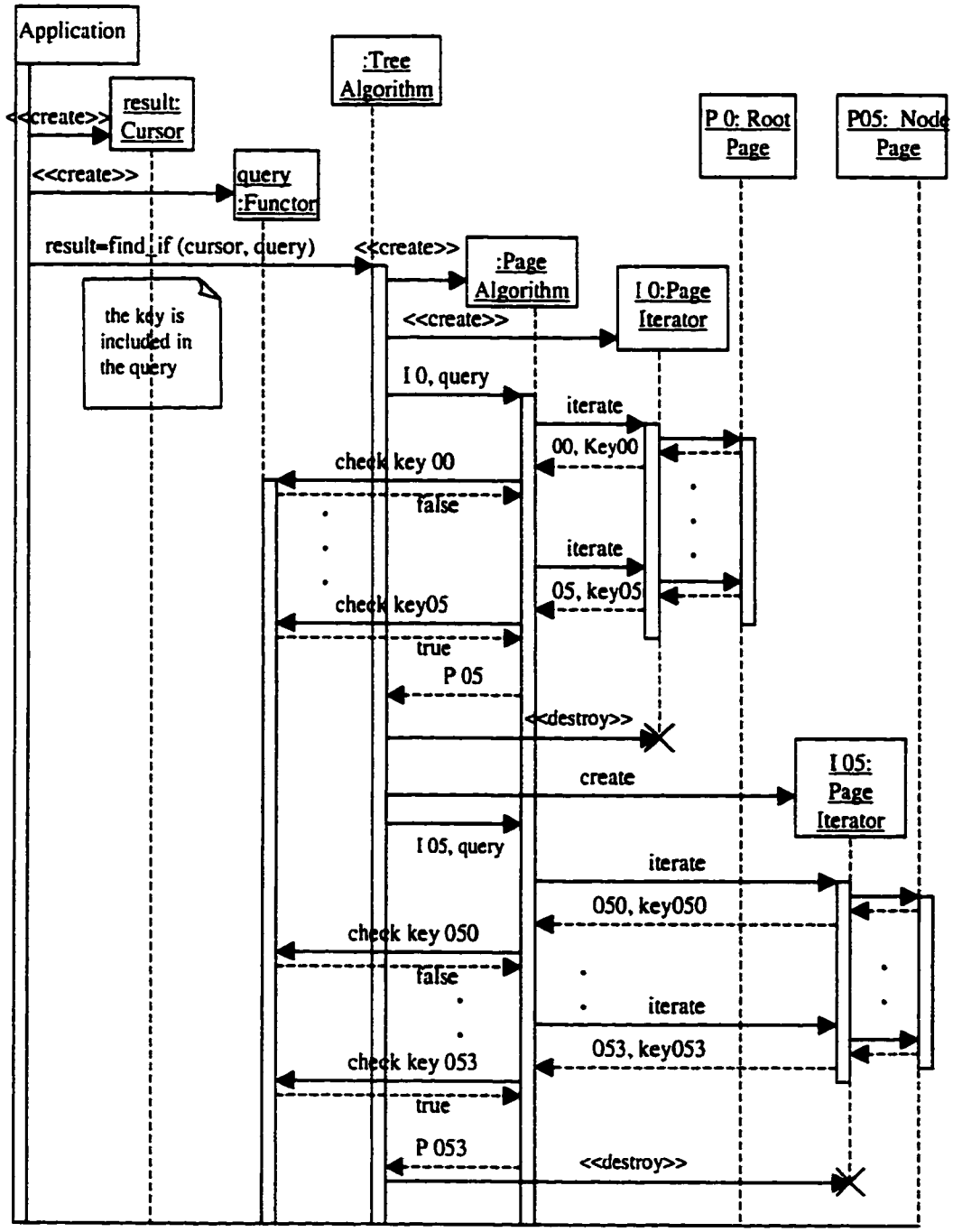


Figure 6.5: Using External Query

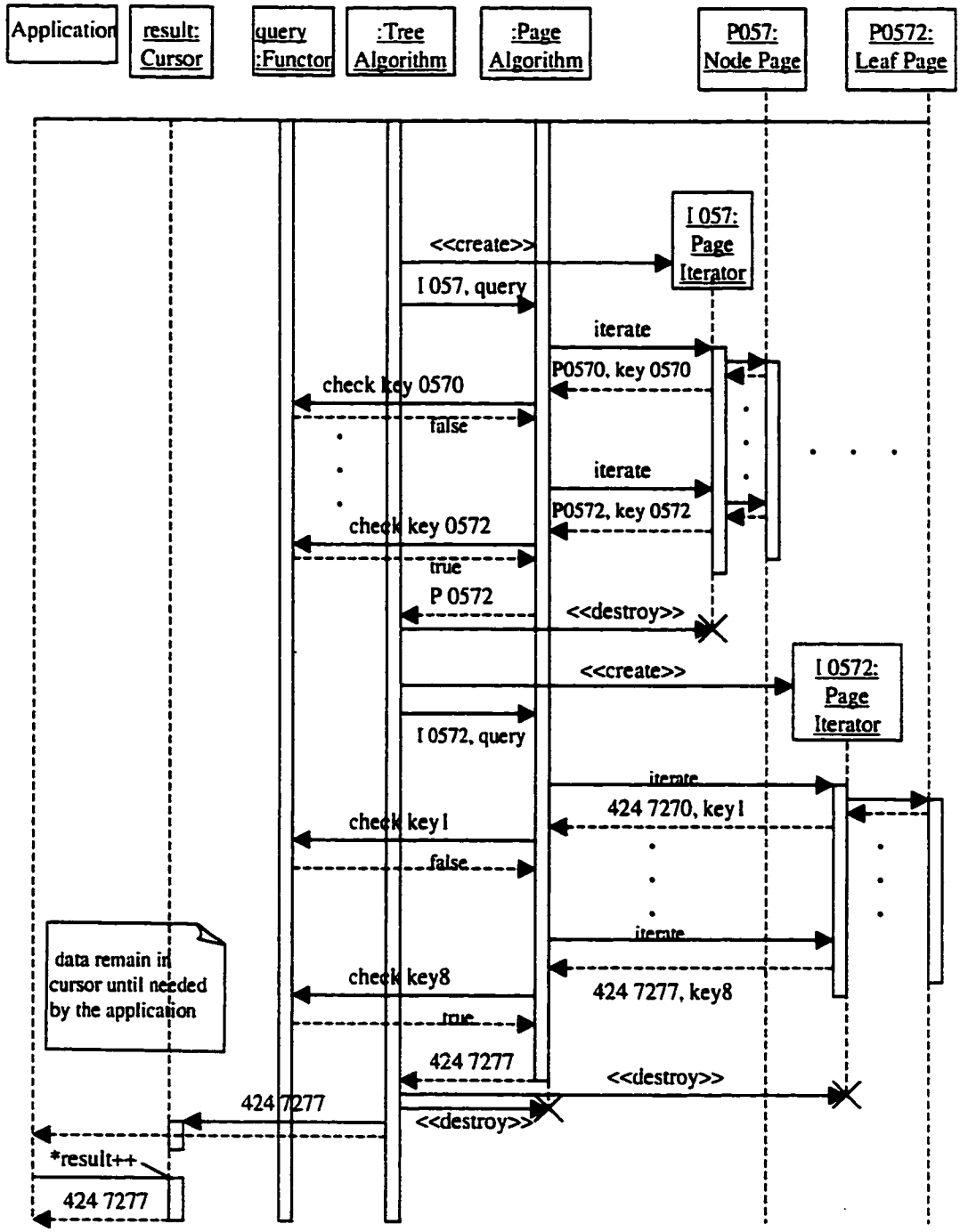


Figure 6.5 (cont): Using External Query

6.3.3 The Cursor Behavior: Iterating Through the Search Result

The cursor object is an iterator object that has the capability of iterating through an index page and being dereferenced to give access to the object it references. As the index returns the candidate leaf page having the needed information, it will in fact return an iterator to the location of interest inside it. The cursor will therefore be referring to a location in a particular page where the information of interest starts. The application can then dereference the iterator and get the information.

Having the same capabilities as a page iterator, the cursor allows the application to iterate through the page sequentially using the operator ++. This is no done directly from the application to the data without asking the index. At a certain point, the cursor might come to the situation where the page contents are depleted and the application is still asking for the next element. Here we can see the difference between the page iterator and the cursor. A page iterator will return null value as the last entry of the page is accessed. The cursor might want to continue to the next page if data was sequentially ordered. For the cursor, data is a linear sequence and the end of a leaf page is followed by the beginning of the next leaf page. This mandates a slight change to the page iterator: as the page contents are depleted, the page will not return a null value but rather a reference to the next leaf page. As the cursor receives this result while trying to access the next page entry, it will be understood and the cursor will go to the next leaf page using that information and start accessing its contents by iterating sequentially through it upon further requests from the application.

This scenario continues until the application ceases asking for further iterations or the last entry in the last leaf page was accessed.

We should notice that this scenario applies for linearly ordered domains only, where data is sorted and stored in a sequential form.

For non linear data, other scenarios apply, as will be shown later.

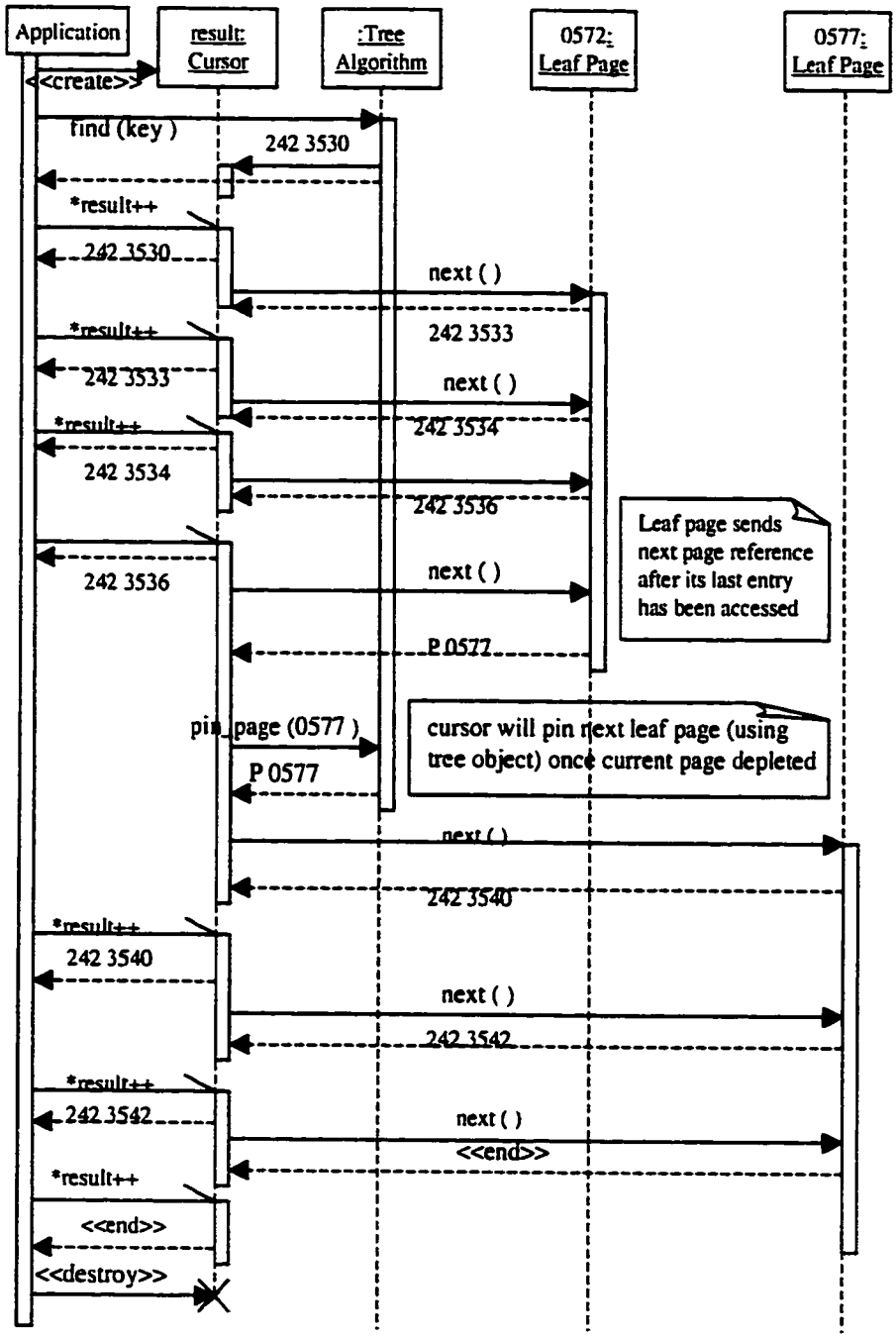


Figure 6.6: The Cursor Iterating through Results

6.3.4 The Activity Diagram for Insertion

After an insertion, the page could become full. In this case the index will need to split it in two. Figure 6.7 shows the activity diagram for insertion.

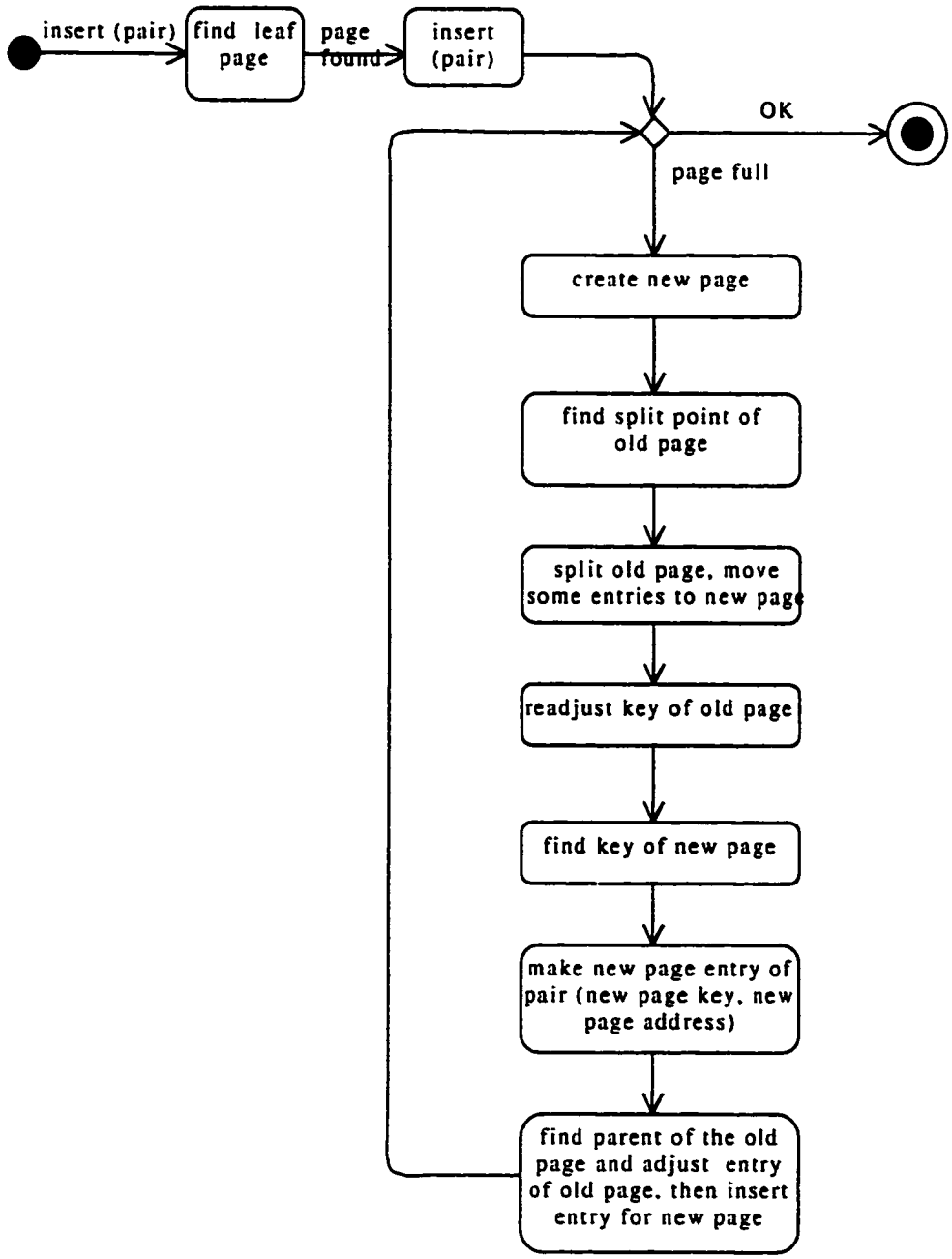


Figure 6.7: Insertion Activity Diagram

6.3.5 The Activity Diagram for Deletion

After a deletion, the page could become sparse. In this case the index will need to borrow from or merge with a neighbor page. Figure 6.8 shows the activity diagram for deletion.

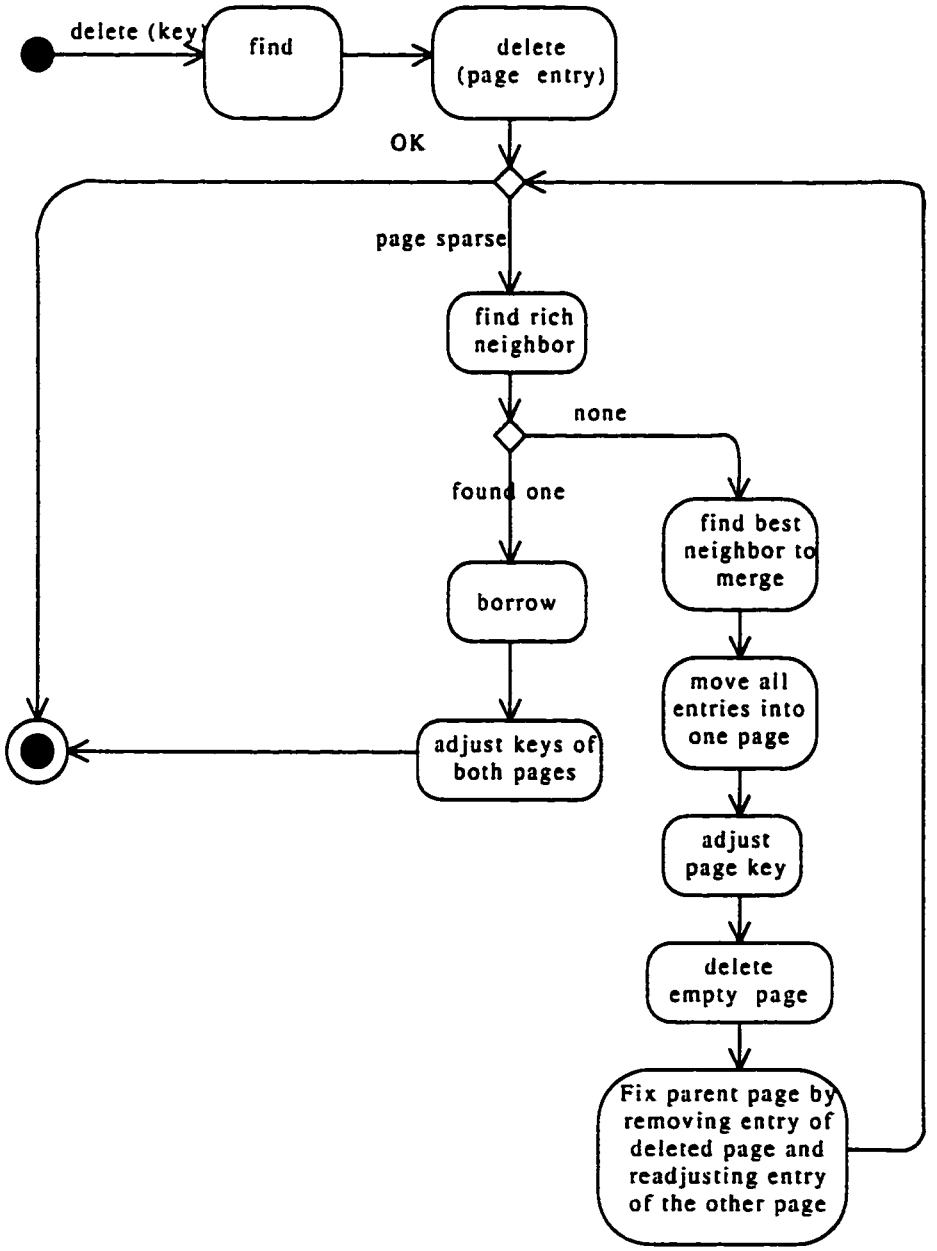


Figure 6.8: Deletion Activity Diagram

Chapter 7 Design for General Domain

The linearly ordered domain is a common application in database, but it does not work for all applications as it poses some constraints and assumptions on the properties of data to be used as a database. We need to isolate the characteristics that are specific to the linearly ordered domain, but not necessarily applicable to the general domain, then we generalize them to define the general characteristics that apply to the general domain. This would mean easing up some of the constraints required on data, to make the system workable with general data formats.

7.1 The Design

The main difference we need to address in general domain index is that each page searched may have more that one result. For example, in the R-tree index, as shown earlier, searching a key might give more than one matching entry. Each of these entries will lead the index algorithm to descend to a different child. So one page leads to descending to multiple children, each of which may lead to descending to yet more children. In the end the index search algorithm will arrive at more than one leaf page, probably tens or hundreds of them as shown in figure 7.1. All of these leaves may have some matching data for the application. The problem is that they are not stored in a sequential order anymore, so they cannot be accessed by just diving to the first matching data entry and then sequentially reading the rest of the entries. We cannot even define first match or sequential in an unambiguous way anymore. The design will include two main modifications to be able to accommodate a general domain database:

1-When searching a page, the searching algorithm should not terminate as soon as the first match was found. Since there might well be other

matches in the page, the algorithm should check all the entries in a page and return all the matching ones.

2-When traversing the index tree, the traversal process should have the ability to go through different paths downward. This cannot be done simultaneously, so the index will use some temporary container to store all the candidate paths and then access them one at a time. A stack will be used to push all the children references returned by a page, and they will be popped one at a time for access.

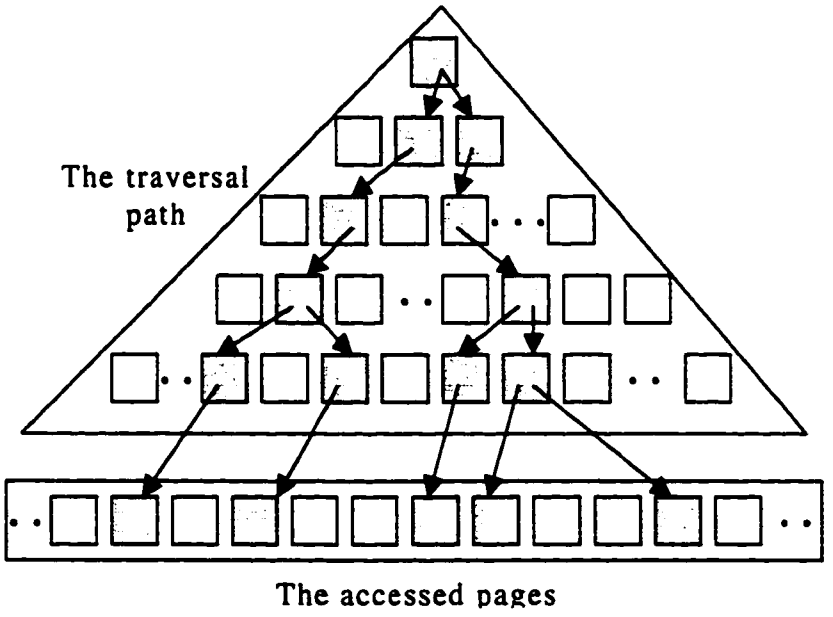


Figure 7.1: The Index Traversal for General Domain

7.2 The System Layout

Figure 7.2 shows a general view of the system components connected together. The index container will include an internal helper container, typically a stack. Note that the stack is not exactly a container, but just an adaptor that connects to an implicit container, masking its unneeded operations and offering only stack operations like push ( ) and pop ( ). The index, in this layout, cannot iterate through the stack by directly

using its implicit container iterator. It can only pop the contents sequentially by sending a pop ( ) request to the stack adaptor.

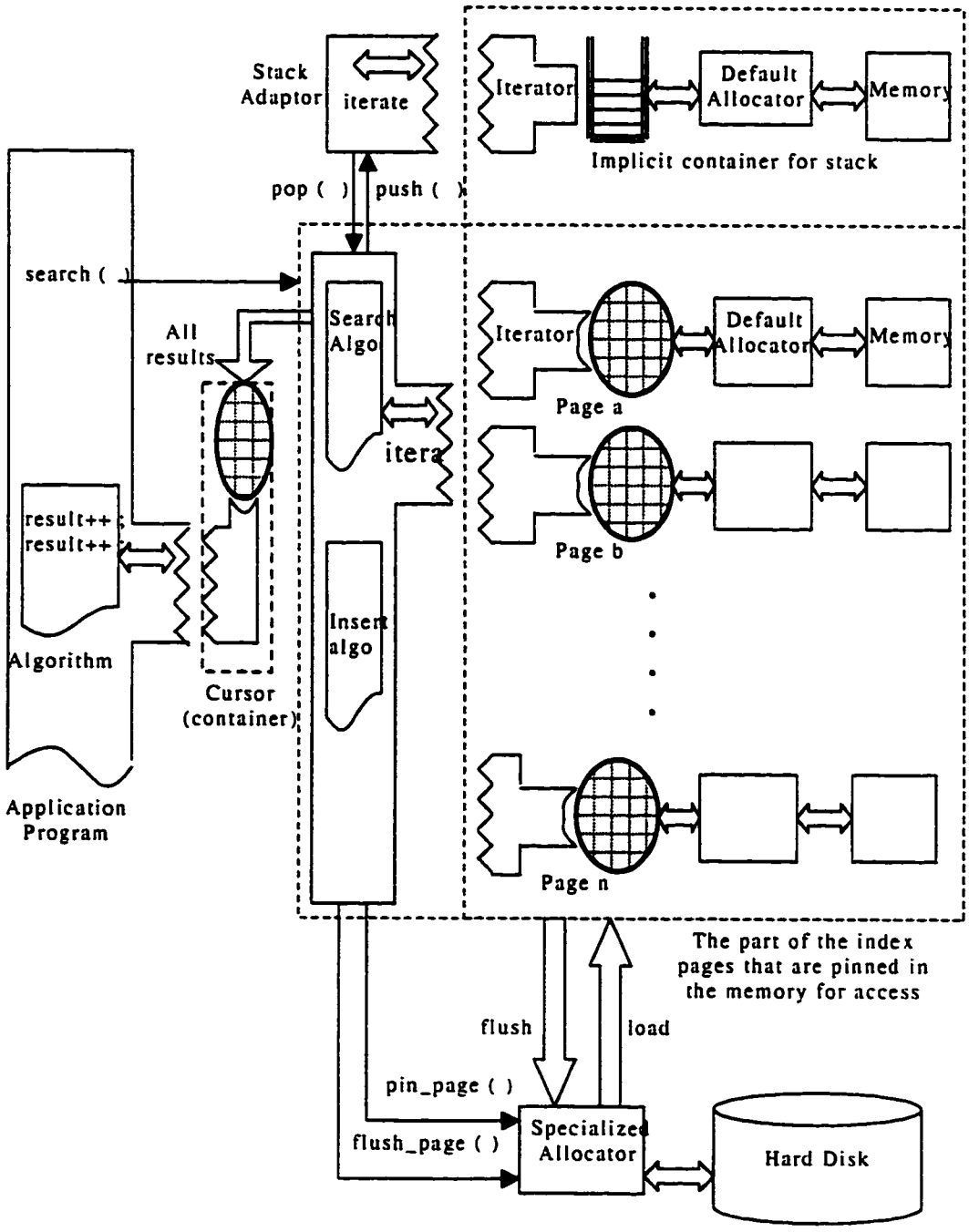


Figure 7.2: System Layout for General Domain Index

7.3 System Flexibility; Using Stack or Queue

Note that the leaf pages are retrieved in a reversed order due to the stack returning the last element first (Last In First Out, LIFO). This makes no impact on this batch search because in the end all matching entries will be in the cursor. If an application requires accessing the leaf pages in a forward order, the system allows for an easy adjustment by replacing the helper container stack with another helper container, Queue (STL container deque) without affecting any of the other system components. Since queue has a first-in-first-out policy (FIFO), this will give the needed result of accessing the leaf pages in a forward order. All the analysis, and requirements for using the stack use are applicable when using the queue.

7.3.1 The Effect of Stack and Queue on Index Behavior

Comparing the behavior of stack and queue and following the pattern of visiting the index pages shows some index behavior differences. The queue allows the index to access the leaf pages in a forward order, while the stack allows accessing them in a reversed order.

Besides, after completing a search in any page, the stack favors the depth direction while the queue favors the breadth direction (if possible). This makes the stack-based search capable of reaching the first leaf page faster (before exhaustively searching each level), while the queue-based search visits all pages in each level before moving to the next level, so after 50% of the search time has elapsed, a stack-based search would have accessed more leaf pages than a queue-based search. In the end, however, they will both have accessed the same pages.

Both these search policies have one thing in common; they both completed the search in every page before going to the next page. They then acted differently when deciding on the next page, going depth or breadth. A third search policy, depth first will follow a different behavior. The search in each page is not even completed, but rather

interrupted after finding the first matching child. This behavior allows reaching the first leaf page even faster than the other two. Table 7.1 shows a brief comparison between the three policies. The Depth first policy is discussed in more details at the end of this chapter.

	Page search	Next page level	Index Traversal
Queue-based	Complete searching every page, visit each page once.	Complete searching all pages in same level.	Pure breadth-first policy.
Stack-based	Complete search in every page, visit each page once.	Quit other pages in same level, go to next level, and come back later to other pages of same level.	Mix of breadth- and depth-first policies.
Stack-based depth-first	Quit page after finding first child, go to child, and come back later to other children of same page.	Quit other pages in same level, go to next level, and come back later to same page or other pages of same level.	Pure depth-first policy.

Table 7.1: Comparison between Index Traversal Policies

7.4 The Class Diagrams

The nature of our framework allows for strong cohesion of classes and weak coupling. This is advantageous here as it reduces the modifications necessary to change from one design to another to as few classes as possible. The first modification, showing the advantage of strong cohesion, will be restricted to the page algorithm or the tree algorithm without affecting other system components. The second modification, showing the advantage of weak coupling, will be done by adding a new

component to the design, a stack, and slightly modifying the index tree algorithm. We will study both modifications in more details. An object diagram of the new design is shown in the figure 7.3. It has a stack as a new component owned and used by the index algorithm to remember the multiple paths in traversing the tree.

The application always owns the result cursor. It uses the index algorithm to ask for the available services like search, insert, delete. The index algorithm will deposit all the results (in case of search for example) into the cursor. The cursor here does not have to access the page iterator or the index iterator to get more results, as they are all deposited into it during the search.

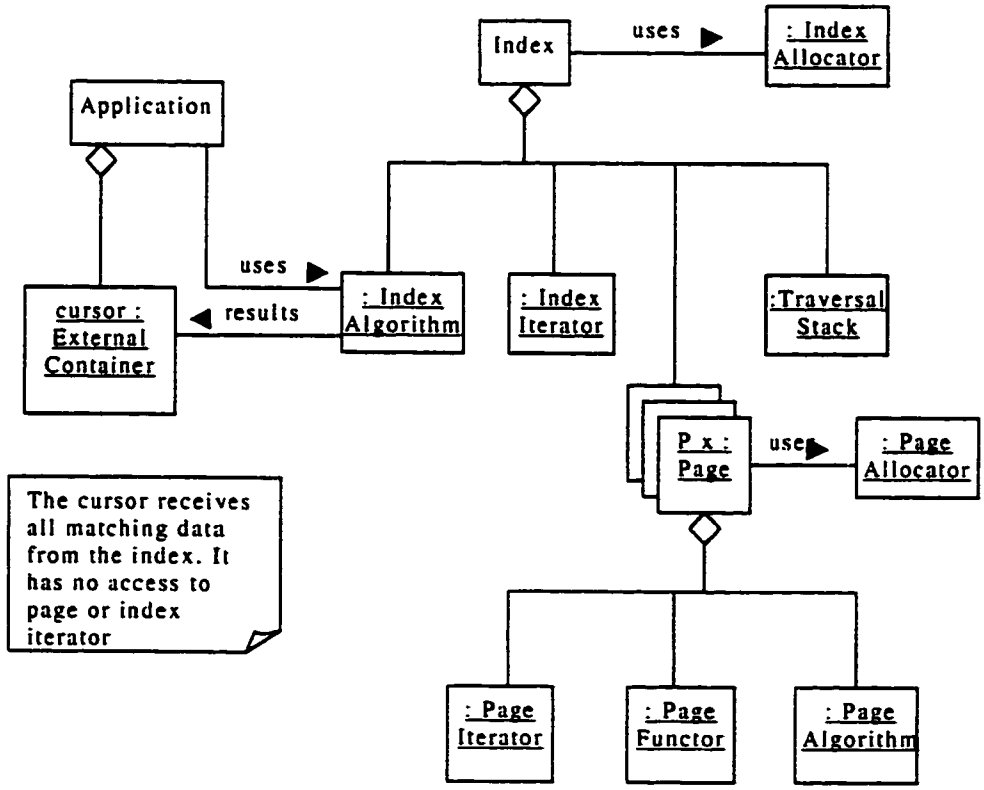


Figure 7.3: Object Diagram for Using Internal Query

7.5 The Behavior of the System

Figure 7.4 shows the sequence of interacting with the traversal stack using an internal query. The query is built-in inside the page container as an internal functor. The index will be the main object controlling the interaction here. It will create a page algorithm and connect it to the root page through an iterator. It will also create a traversal stack object. Next it will start iterating through the page by sending a search command. The page algorithm will search for the first match, send it to the index and stop at the next position after the first match. Assuming that the page accessed is an internal node page having other page references as its data entries, the index, receiving the matching data in the form of page references, will send it to the stack for later use, (see also section 3). Next the index will send another search command to the page iterator, which will start the search from its current position, right after the first matching data, and search for the next match, send it to the index, and so on. As the page algorithm reaches the end of page, it will send an end-of-page, EOP, signal to the index. The index will proceed by popping the next page reference from the stack and access that page and iterate through it.

Modifying the system to use an external query requires only slight changes. The modification will be limited to making the page algorithm ask the external query if entries were matching, instead of asking the internal functor. The index will not always send the received page entries matching the searched key to the stack. It will need to decide on whether to send them to the cursor or to the stack depending on their type. Index page references will go to the stack, while data page references will go to the cursor. The source of entries will decide their type. Entries from a leaf page will certainly be data references and must be sent to the cursor to be retrieved by the application, so index will test if the statement *page is leaf* was true then send entries to the cursor.

Entries from a node page will not, however, always be page references to be sent to the traversal stack. For example, a single-node index will have only one node page (the root), which is also a leaf page. This page will therefore have data references as its entries (there is no other pages to send to the traversal stack). Testing if this page was a node page will give true even that, being a leaf as well, it has entries that are actually data references and should be sent to the cursor, not to the stack. Therefore, the index must check if *page is a leaf* was false and not *page was a node* was true to avoid such a problem. This should be reflected in the flag of the page itself, by having the flag to test leaf_page or not. There will be no need for another flag like node-page.

The iteration can be left to the page to do without index interaction. The page algorithm will send data directly to its proper destination.

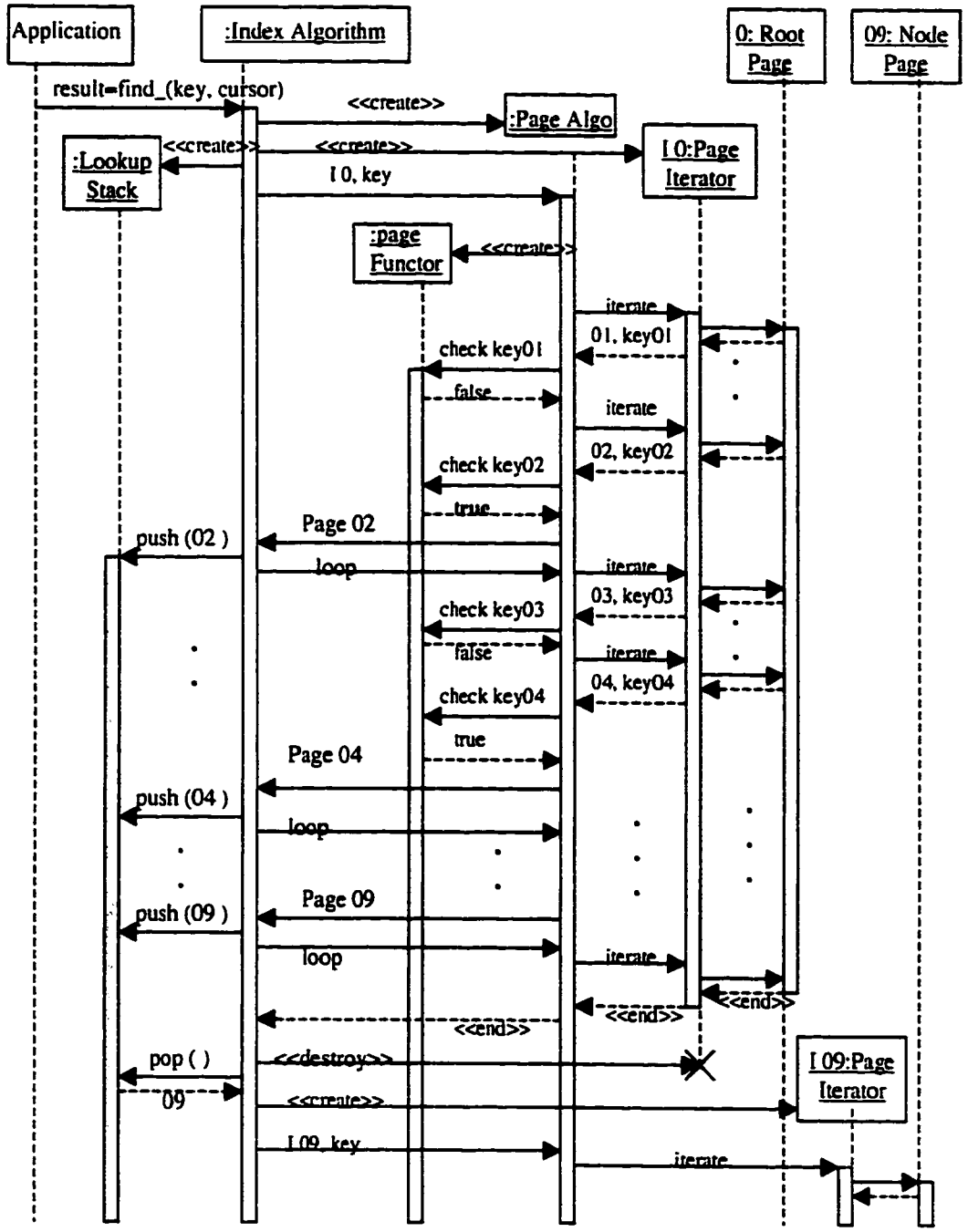


Figure 7.4: Using Internal Query

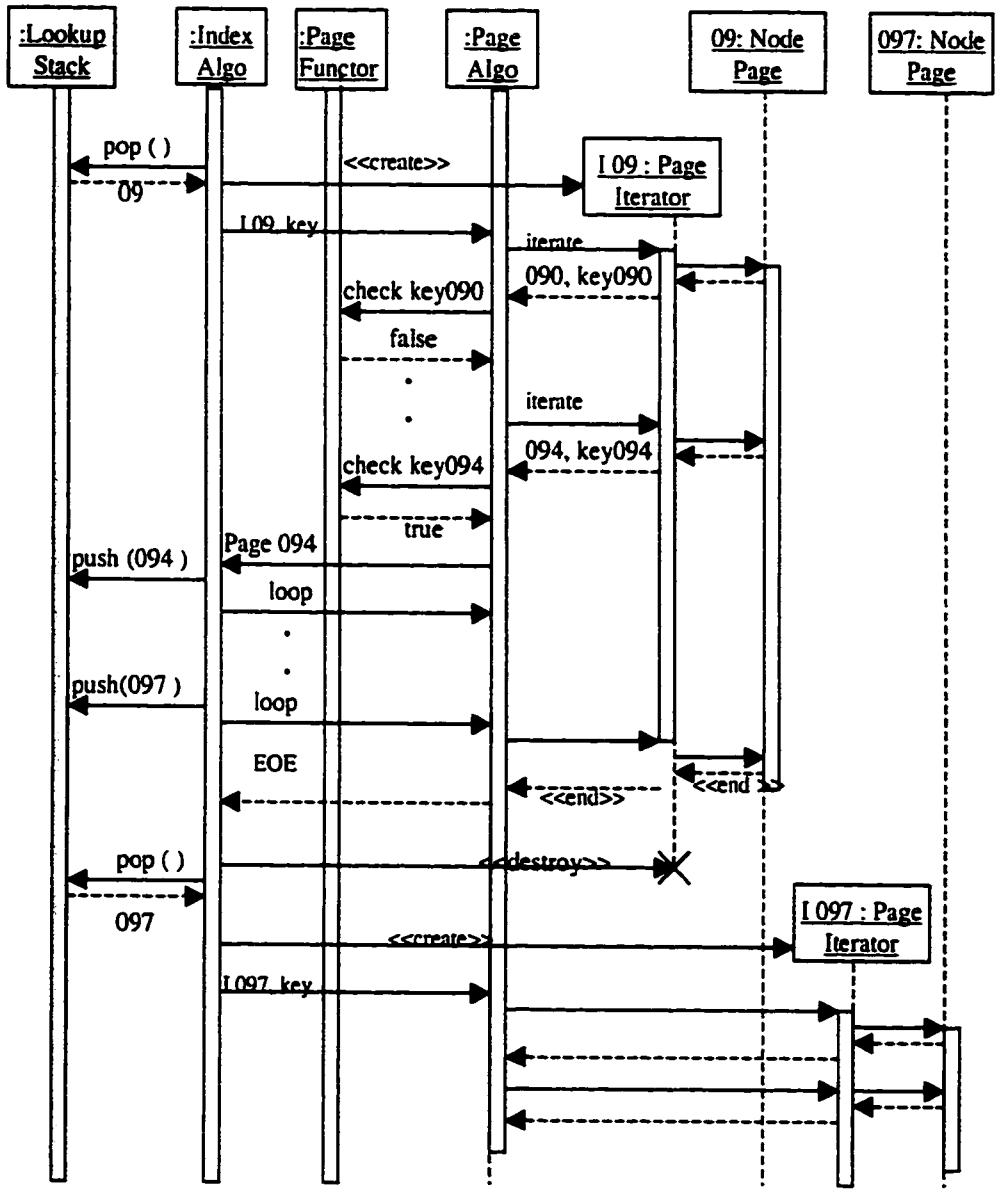


Figure 7.4 (cont): Using Internal Query

Chapter 8 Similarity Search Applications

So far we have seen the analysis and design for two different domains: the linearly ordered domain, and the general domain (with both breadth first and depth first access methods). These domains have one common search concept; the equality search. As we searched for a key in different index pages, we compared the keys in a page with the key we are searching. Each comparison was done using an internal- or an external functor (query), and always gave one of two outcomes, matching or not matching. We then did our next move either downward (in depth-first general domain, and in linearly ordered domain) or laterally (in breadth-first general domain).

All these methods depend on choosing one of the two outcomes of the comparison object. No other outcomes of the comparison were allowed.

In some database applications, this is not always the case. Some times the comparison of two objects does not just give a yes/no result, but rather how close the two objects are; or in exact terminology how similar the two objects are. Even if the two objects do not exactly match, we are interested in them if they were similar, and not interested if they were not similar.

8.1 The Analysis

The similarity comparison would search for common features between two different objects, depending on the application. As explained in 3.6, the search result could produce a number to represent the distance between two objects (with 0 being exactly the same) or a percentage value showing how similar the two objects are (with 100% being exactly the same). A higher distance value or a lower percentage value would mean less similarities between the two objects.

Translating the percentage value into more similar or less similar is also application dependent. A 90 % similarity could be considered very similar

in one application, while considered not similar enough in another. This introduces the concept of threshold value for similarity, wherein we may define a threshold value for similarity outcomes to be considered in the result set of similar objects or not.

Changing this rather vague concept of the percentage values outcome into a more solid one like similar or not will make data easier to handle and study. The application may set the threshold for the percentage outcome depending on the domain. All values below this threshold value will be considered as not matching and ignored, while values above it will be considered as matching and taken into account (another approach is to consider the n-most similar objects without using a threshold value. This, again, depends on the application, but does not affect our analysis here).

This helps in eliminating a certain amount of data that is not similar enough (to the data we are searching) to be considered.

As we search we will have on hand more than one object to consider as similar to the one we are looking for. We may want to visit some or all of them, so it would make sense to visit the more similar ones first; the ones with higher percentage similarity. In other words out interest in visiting these objects will be proportional to their value of similarity. The best way to do that is by ordering them in a queue according to their similarity value, and start visiting them from higher to lower.

With the index pages arranged into a hierarchy over different levels, it will be slightly more complex. Starting from the top, we go downward visiting all similar pages (with similarity percentage value above threshold value) from higher to lower percentage. As we visit a page, we exhaustively search all its entries and the search will return certain page entries (of key and page reference) accompanied with how similar it is to our key. All these entries are then added to a priority queue with their percentage similarity value as its priority. The priority queue will take the responsibility of inserting entries partially or totally ordered such that we can always access the entry with the highest priority next.

As we finish searching a page, we look for the next page to visit by reading the priority queue. We will get the page with the highest similarity value, and search it, inserting further entries with their similarity values into the queue, read the next page, search it and so on.
This search is neither a depth first nor a breadth first. As explained in chapter 3, the direction of our next move is not simply one step lateral or one-step vertical. It solely depends on the next value popped from the queue. This could result in any kind of movements. The next page could be anywhere in the index, two steps up, five pages to the right, ... etc. On the index it will look like jumping in all directions, possibly multiple levels at a time, just favoring the highest similarity value as the sole motive to our next move (a demonstration with a numerical example of this movement is given in section 8.2). It is clear that, unlike depth-first general domain, and linearly ordered domain, and similar to the breadth-first general domain, we are doing an exhaustive search on each page. The reason is that we are not in a hurry to descend to the first match found, but rather want to find the best match in each page before leaving it. This implies that we delete the page from the priority queue after we visited it.
On the index level, we pick up the best page to visit, and search for the best matching entries in it (more pages). We are doing what can be called the best of the best search. To guarantee this criterion, we do not just take the best match in a page and visit it directly; we rather put all matching pages from every search, regardless of their level, in the queue with the matching pages from previous search. We then select the best global match over the whole index again and again, and search for the best matches in it, add them to the queue, and so on.
As we hit a data item, we send it to the cursor. This item should be the most similar item to the one the application is looking for. We then wait for the application to access it.

8.2 A Numerical Example

Figure 8.1 shows with a numerical example how the similarity search proceeds to find the matching data according to how close they are to data we are searching. For simplicity, we will assume that each leaf page contains directly the data we are looking for (an Image, for example). If the leaf pages contained keys with similarity percentage values, we can just add one more level to the example, and use the same technique. It would only be unwieldy large. This example here will show the necessary details only. The pages of interest are shown with their numbers, using the same numbering convention as before. The pages, which are of no interest to us, are shown only as smaller rectangles to work as a placeholder for those pages. They are considered anonymous pages and their numbers are not shown, for obvious space reasons

We note that as we descend on the branches of a subtree in the figure, we might get lower priority values than the higher level nodes of the same subtree. For example, node 07 had priority 89%, but as we descend on it, its children 075, and 077 have lower priorities of 75 % and 85%, then further child 0774 has only 74%. This is not an unusual feature of similarity search. The keys at higher levels of the index, covering larger subtrees, will be required to summarize the features of all the pages in their subtrees. This might lead to ignoring some details, or generalizing them, making them coarse-grained. Testing one of those coarse-grained keys might give a less accurate high similarity value. As we take on this key and descend in its subtree, we get more fine-grained keys as more details are revealed about the now smaller subtree. Those new details might result in a lower, but more accurate similarity values than first appeared on the parent page key. This could lead to the unwanted side effect of accessing less similar node pages before accessing more similar ones, being misled by this feature. Using the priority queue helps recover and avoid this side effects in the leaf level as we put all pages in one global priority queue, with their similarity to the searched key as the

priority metric. As the pages in a branch get less similar than their parent page once did, they are automatically pushed behind older pages already in the queue with now more similarity to our searched key, resulting in abandoning the less similar branch, and moving to the more similar one. In figure 8.1, instead of going to 0774, we abandoned the whole subtree of node 07, and moved to 03. This will always guarantee that we will visit the page with highest priority, and reduce the possibility of going on a less lucrative path. In this example, the arrows show that we have information about that page with the percentage of their similarity to the key we search. The dotted ones mean that the page is not yet visited. The solid arrows show that we used this path, and already visited the page at its end. The visited pages are shown in a light gray tone.

8.3 The Order of Reading Leaf Pages

In both linearly ordered and general domain, we access the leaf pages in some forward or reverse physical order depending on the access method and the type of helper container used. The breadth first policy using a stack (we called it a mix of breadth- and depth first policies) always read the matching leaves from the last to the first. The breadth first policy using a queue (we called it pure breadth-first policy) always read the matching leaves from the first to the last. The depth first policy starts by reading the first and precedes to the last match. All these policies are position-oriented in that the order of accessing the pages depends on their position, regardless of their contents.

In similarity search, there is no predetermined order in accessing the matching pages. This policy is contents-oriented in determining the sequence of pages to access, regardless of their position. We access the most similar page, then the second most similar and so on. Changing the similarity value of the pages to the one we search will therefore be the factor leading to changing the order we access them.

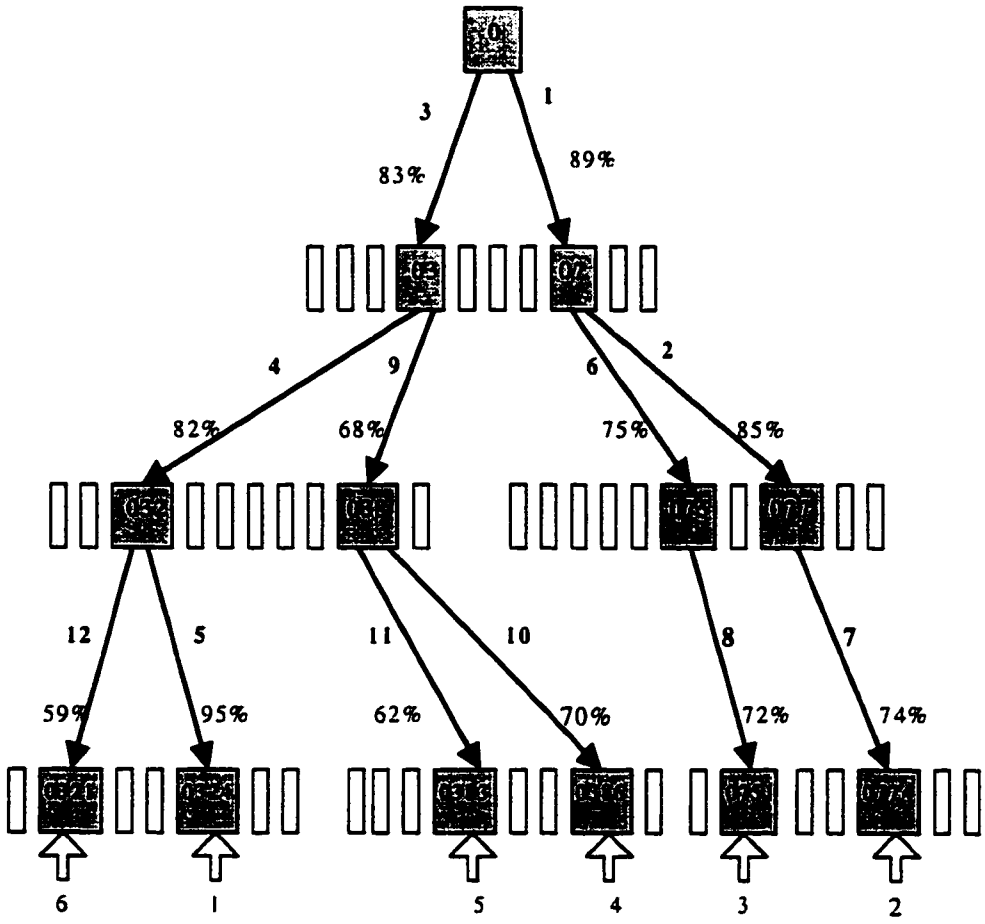


Figure 8.1: Search Sequence in Similarity Search

Chapter 9 Conclusion

In this thesis we describe how to build a generalized tree index from independent building blocks that are coherent and decoupled.

A database index is a data structure with a set of functionality to search and update a database. The index structure follows certain algorithms for searching, insertion, and deletion of data as well as internal index maintenance algorithms that guarantee data integrity.

The C++ STL library, a part of the standard library, introduces a concept for breaking a container data structure into a set of algorithm – iterator – container modules. We expand this concept to build a complete modular index system. In the design, we break the index data structure with functionality into container of pages with each page built as container of entries where each entry is a pair of <key, reference> .

The index module, being a container of pages, is further decomposed into algorithm – iterator – container modules. Similarly the page module, being a container of entries is further decomposed into algorithm – iterator – container modules. The entries are then stored in each page as pairs of key, reference. Other modules; allocators, further separate the process of primary and secondary storage management from the other modules. We also use some helper modules; like result cursor container, search container (stack / queue), and functors to complement the necessary system parts. We use adaptor modules like functor adaptors, container adaptors, and iterator adaptors to adjust the slight irregularities between some incompatible modules. In the end we add the data and the data reference modules to complete the system. This allows us to construct a complete index system from modules.

In order to adapt the system to different keys / data types, different queries, different access methods, and different storage media, we need to locate and modify (or simply replace) some modules in the system. We provide some examples to show how to adopt the system to linearly

ordered domain applications by providing an analysis of this domain. This analysis identifies the modules in the system that needed modifications. We reflect these modifications on the system design and eventually on the interface. Similarly we adapt the system to general domain applications by first providing an analysis of this domain and exposing the main changes found in this domain. Again, this leads us to identifying the modules to modify in the design along with a new module to add (either a stack, or a queue) to achieve a new system in general domain (with both depth-first and breadth first access methods). Again these modifications are then reflected to the interface. Finally, we provide an analysis of the similarity-search domain and follow the same steps to identify and apply the changes needed in the affected modules.

Using a modular design for the index system has the advantage of making it easier to adapt the system to work in different database domains. The analysis of the domain determines the modules that need changes (or replacement), and the kind of changes (or modules) required. This means that the other modules need not be changed. This greatly reduces the efforts needed to modify the system. The complexity of modification is also reduced since the developer does not need to know about the details of all modules, but only of those modules to be changed along with an overview of the system. The adoption of STL approach adds great advantage of having a wealth of off-the-shelf standard modules that can be simply used to replace system modules in the process of modifying the system. This promotes code reuse and thus increase readability, user-friendliness and reduces time and money overheads incurred during the application development process.

9.1 Future Work

Our ultimate goal is to build an index system to handle any type of data, key, query or access method in a real database application. One important issue in database is concurrency and locking mechanisms to allow for

multiple access to the same piece of information while guaranteeing data integrity. This issue needs to be addressed according to the classes suggested in the design. Another important work that still needs to be done is the construction of concrete classes as shown in the design to obtain a working index.

The design started with tree indexes and we see that it has a potential to handle other index types such as inverted lists, hash tables, and parallel search. These types of indexes deserve further investigations, and we hope to see the design extended to cover them as well.

Bibliography

[AKKS99] M. Ankerst, G. Kastenmüller, H.-P. Kriegel, T. Seidl. **3D Shape Histograms for Similarity Search and Classification in Spatial Databases**. Proceedings of the 6th International Symposium on Large Spatial Databases (SSD'99), Hong Kong, China, in: Lecture Notes in Computer Science, Vol. 1651, Springer, 1999, pp. 207-226.

[AKKS99-2] M. Ankerst, G. Kastenmüller, H.-P. Kriegel, T. Seidl. **Nearest Neighbor Classification in 3D Protein Databases**. Proceedings of the 7th International Conference on Intelligent Systems for Molecular Biology (ISMB'99), Heidelberg, Germany, AAAI Press, 1999, pp. 34-43.

[Aok98] P. M. Aoki. **Generalizing "Search" in Generalized Search Trees**. Proceedings of the 14th IEEE International Conference on Data Engineering, Orlando, FL, Feb. 1998, pp. 380-389.

[Aus99] Matthew H. Austern. **Generic Programming and the STL**. Addison-Wesley, 1999.

[BBB+97] S. Berchtold, C. Böhm, B. Braunmüller, D.A. Keim, H.-P. Kriegel. **Fast Parallel Similarity Search in Multimedia Databases**. Proceedings of the ACM SIGMOD International Conference on Management of Data AZ, 1997, Best Paper Award, pp. 1-12.

[BBBK00] C. Böhm, B. Braunmüller, M. Breunig, H.-P. Kriegel. **High Performance Clustering Based on the Similarity Join**. Proceedings of the 9th International Conference on Information and Knowledge Management (CIKM 2000), Washington DC, 2000, pp. 298-313.

[BBKM00] C. Böhm, S. Berchtold, H.-P.Kriegel, U. Michel. **Multidimensional Index Structures in Relational Databases**. in: Journal for Intelligent Information Systems, Vol. 15, 2000, pp. 51-70.

[BCC+02] Greg Butler, Ling Chen, Xuede Chen, Ashraf Gaffar, Jinmiao Li, Lugang Xu. **The Know-It-All Project: A Case Study in Framework Development and Evolution**, to appear in Domain Oriented Systems Development: Perspectives and Practices, Kiyoshi Itoh, Satoshi Kumagai (eds), Gordon Breach Science Publishers, UK, 2002.

[Bch94] Grady Booch. **Object-Oriented Analysis and Design**. The Benjamin/Cummings Publishing Company 1994.

[BD99] G. Butler and P. Dénommée. **Documenting Frameworks**, in Building Application Frameworks: Object-Oriented Foundations of Framework Design. M. Fayad, D. Schmidt, R. Johnson (eds). John Wiley and Sons, New York, September 1999, pp. 495-504.

[BEK+98] S. Berchtold, B. Ertl, D. A. Keim, H.-P.Kriegel, T. Seidl. **Fast Nearest Neighbor Search in High-dimensional Space**. Proceedings of the 14th International Conference on Data Engineering (ICDE'98), Orlando, FL, 1998, pp. 209-218.

[BKM00] G. Butler, R.K. Keller, H. Mili. **A Framework for Framework Documentation**. ACM Computing Surveys 32,1 (March 2000) electronic symposium.

[BKS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, Bernhard Seeger. **The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles**. Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990. ACM Press 1990, pp. 322-331.

[BRI99] Grady Booch, James Rumbaugh, Ivar Jacobson. **The Unified Modeling Language User Manual**. Addison-Wesley 1999.

[Bry00] Ulrich Breymann. **Designing Components with the C++ STL: A New Approach to Programming**. Addison-Wesley 2000.

[But99] Greg Butler. **Developing Frameworks by Aligning Requirements, Design, and Code**. Proceedings of the 9th Workshop on Software Reuse (WISR-9), Austin, Texas, January 1999, 5 pages.

[BX01] Greg Butler, Lugang Xu. **Cascaded Refactoring for Framework Evolution**. Proceedings of the 2001 Symposium on Software Reusability. ACM Press, 2001, pp. 51-57.

[Dsi90] Bipin C. Desai. **An Introduction to Database Systems**. St. Paul West Publishing 1990.

[EN94] Ramez Elmasri, Shamkant B. Nanathe. **Fundamentals of Database Systems**, second edition. Benjamin/Cummings Publishing Company 1994.

[FBF+94] C. Faloutsos, R. Barber, M. Flickner, J. Hanfner, W. Niblack, D. Petkovic, W. Equitz. **Efficient and Effective Querying by Image Content**. Journal of Intelligent Information Systems, Vol 3 (1994) pp.231-262.

[FZ92] Michael J. Folk, Bill Zeollick. **File Structures**, Second Edition. Addison-Wesley 1992.

[Gdm93] John Goodman. **Memory Management for All of Us**. Sams Publishing 1993.

[GG98] V. Gaede, O. Guenter. **Multidimensional Access Methods**. ACM Computing Surveys, Vol. 30, No. 2, 1998. pp. 170-231.

[GHJV95] Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides. **Design Patterns**. Addison-Wesley 1995.

[Gut84] Antonin Guttman: R-Trees. **A Dynamic Index Structure for Spatial Searching**. Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, June 18-21, 1984. ACM Press 1984, pp. 47-57

[HCB+99] Joseph M. Hellerstein, Chad Carson, Serge J. Belongie, Megan C. Thomas and Jitendra Malik. **Blobworld: A System for Region-Based Image Indexing and Retrieval**. Proceedings of the 3rd International Conference on Visual Information Systems, VISUAL 99. Amesterdam, Netherlands, June 1999, published as lecture notes in computer science, Springer-Verlag, vol. 1614, pp. 509-516.

[HNP95] Joseph M. Hellerstein, Jeffrey F. Naughton and Avi Pfeffer. **Generalized Search Trees for Database Systems**. Proceedings of the 21st International Conference on Very Large Data Bases, Zurich, September, 1995.

[HSE+95] J. Hafner, H. S. Sawhney, W. Equitz, M. Flickner, W. Niblack. **Efficient Color Histogram Indexing for Quadratic Form Distance Functions**. IEEE Trans. On Pattern Analysis and Machine Intelligence, Vol 17, No. 7. IEEE Press (1995) 729-736.

[JD88] A. K. Jain, R.C. Dubes. **Algorithms for Clustering Data**. Prentice-Hall 1988.

[KKS98] G. Kastenmüller, H.-P.Kriegel, T. Seidl. **Similarity Search in 3D Protein Databases**, Proceedings of the German Conference on Bioinformatics (GCB'98). Köln, 1998.

[Knt73] Donald E. Knuth. **The Art Of Computer Programming**, Volume2 / Seminumerical Algorithms. Addison-Wesley 1973.

[Knt73-2] Donald E. Knuth. **The Art Of Computer Programming**, Volume3 / Sorting and Searching. Addison-Wesley 1973.

[KS97] Norio Katayama, Shin'ichi Satoh. **The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries.** Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data (May 1997) pp. 369-380.

[LH96] Jesse Liberty, J. Mark Hord. **ANSI C++.** Sams Publishing 1996.

[Lip96] Stanley B. Lippman. **Inside The C++ Object Models.** Addison-Wesley Longman, 1996.

[LL98] Stanley B. Lippman, Josee Lajoie. **C++ Primer**, Third Edition. Addison-wesley 1998.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Loensen. **Object-Oriented Modeling and Design.** Prentice Hall 1991.

[Res99] Steven P. Reiss. **A Practical Introduction to Software Design with C++.** John Weily and Sons. 1999.

[Rbn81] John T. Robinson. **The K-D-B-Tree: A Search Structure For Large Multidimensional Dynamic Indexes.** Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data, Ann Arbor, Michigan, April 29 - May 1, 1981. ACM Press 1981 , pp. 10-18

[Rob00] Robert Robson. **Using the STL: the C++ Standard Template Library**, second edition. Springer-Verlag 2000.

[SK01] T. Seidl, H.-P. Kriegel. **Adaptable Similarity Search in Large Image Databases**, in: R. Veltkamp, H. Burkhardt, H.-P.Kriegel (eds.): State-of-the Art in Content-Based Image and Video Retrieval, Kluwer Publishers, 2001, pp. 297-317.

[SK95] T. Seidl, H.-P. Kriegel. **Solvent Accessible Surface Representation in a Database System for Protein Docking.** Proceedings of the 3rd International Conference On Intelligent Systems for Molecular Biology (ISMB), 1995, pp. 350-358.

[SK97] T. Seidl, H.-P. Kriegel. **Efficient User-Adaptable Similarity Search in Large Multimedia Databases**, Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB'97), Athens, Greece, 1997, pp. 506-515.

[SKH99] Mehul Shah, Marcel Kornacker, Joseph M. Hellerstein. **amdb: A Visual Access Method Development Tool**. User Interfaces for Data Intensive Systems (UIDIS). Edinburgh, 1999.
[Slz88] Betty J. Salzberg. **File Structures**. Printice-Hall 1988.

[Som95] Ian Sommerville. **Software Engineering**, Fifth Edition. Addison-Wesley 1995.

[SRF87] Timos K. Sellis, Nick Roussopoulos, Christos Faloutsos. **The R+-Tree: A Dynamic Index for Multi-Dimensional Objects.** Proceedings of the 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England, pp. 507-518.

[Str00] Bjarne Stroustrup. **The C++ Programming Language**, special edition. Addison-Wesley 2000.

[Str94] Bjarne Stroustrup. **The Design and Evolution of C++**. Addison-Wesley 1994.

[WJ96] D. A. White and R. Jain. **Similarity Indexing with the SS-tree**. Proceedings of the 12th International Conference on Data Engineering, New Orleans, USA (Feb. 1996), pp. 516-523.

Appendix A Flow of Events

Expert developer "designs a database system"
Pre-condition: None
Main flow: The developer studies the different functional and non-functional requirements of the system and puts a design to satisfy them.
Post-condition: A design for a database and index system, including the primary and secondary indexes.
Expert developer "Implements a database system"
Pre-condition: An integrated design of the system.
Main flow: The developer implements the different components of the system and test it on a platform.
Post-condition: A core code for a database system including the indexes to help access them.
Database Developer "provide a query "
Pre-conditions: An index core code must be present.
Main flow: Database Developer writes the code for a new index query class to be used by an existing system.
Post-condition: A customized index code that supports a new query.
Database Developer "provides a data type "
Pre-conditions: An index core code must be present.
Main flow: Database Developer writes the code for a new index data type class to be used by a new system. This data type will give a new index capable of dealing with database that uses the new data type.
Post-condition: A customized index code that supports an application with a new data type.
Database Developer "defines new access method"
Pre-conditions: An index core code must be present.
Main flow: Database Developer writes the code for building a new index
required by a new system. The new index structure will be capable of using a new access method concept to traverse the tree.
Post-condition: A customized new index code that supports a new access method concept.
Database Developer "sets index parameters "
Pre-conditions: An index core code must be present, and any customizable parts (extensions) must be implemented.
Main flow: The extension programmer sets the index parameters (like the page fill factor), the tree order to adjust the index implementation to the particular application.
Post-condition: A customized index code that has specific values for its parameters suitable for the application and the system variables.
Database Administrator "defines scheme"
Pre-conditions: A suitable design for data tables and a Data Definition Language (DDL).
Main flow: The DBA uses the DDL tools to define the format of the tables used to store the data. The definition will include table name, the name and type of each field and the constraints applied to them.
Post-condition: A definition of the tables used in the data and the relations between them.
Database Administrator "builds data references"
Pre-conditions: Physical data files carrying the bulk of data to be used in the database must be available in the data tables.

Main flow: Build data groups / partitions (if necessary), and define references to their physical locations, as well as suitable keys describing those partitions.
Post-condition: Creation of data references that will be used by the index.

Database Administrator "build index"

Pre-condition: A customized index for a particular application must be available. Data scheme must be defined and Physical data must be loaded. Reference file comprising data keys and references to their physical locations must be available.
Main flow; using a bottom up technique: Create an empty index file, sort the reference file, then build the index using the bottom up technique.
Alternative flow; using the top down technique: Create an empty index file then build the index using top down technique.
Post-condition: Creation of an index to the data.

Database Administrator "dynamically fine-tune index"

Pre-conditions: An index file built on the physical data, and an index object that can successfully open an index file (same type)
Main flow: The DBA runs the system by using an index object to open an existing index file of the same type, then dynamically fine-tune the system by monitoring the performance while the index is being used and adjusting the system parameters (bucket/page size, fill factor etc.) to reach the optimal performance.
Post-condition: An index that is optimized for the applications using it.

Database application "uses index"

Precondition: A customized index suitable for this application and a cursor must be available. A previously created index object must exist, that can be opened by the application, connected to an index file built with the same extension type.
Main flow: Search data
The application will search for some data using a query. First it initializes an iterator (cursor) object to accept pairs of key, data reference as an output from the search operation. The search algorithm traverses the index using the query to select the candidate data references at leaf level that satisfy the given key. These will be returned to the cursor object, where the application can access them sequentially.
Alternative flow: Insert data
The application will insert some data in the form of key, data reference. The application will provide the key, data reference pair to be inserted. The index insertion algorithm will perform it in two steps: First it will use the key in the search algorithm to find a suitable location on the index leaf level for insertion. Second the insertion algorithm is used to insert the key, data reference pair in the resulting location. The necessary adjustments to the index are then performed.
Alternative flow: Delete data
The application will attempt to delete some data from the database using a key. The application provides a key for data to be deleted. This is done in two steps: First the delete algorithm will use the key to try and locate the data to be deleted using the search algorithm. Second if the index find the location of candidate data for deletion, the delete will be applied on data occupying that location. The necessary adjustments to the index are then performed.
Post-condition: The index is adjusted to reflect the current status of the data.

Appendix B Interfaces

Page public type members and data members.

page <Key, T, Compare, Allocator> :: **iterator** ; //internal iterator class

page <Key, T, Compare, Allocator> :: **const_iterator** ;
//An iterator class that allow read-only access to the container elements

page <Key, T, Compare, Allocator> :: **reference** ;
//type of the reference to the **contents** of the container

page <Key, T, Compare, Allocator> :: **const_reference** ;

page <Key, T, Compare, Allocator> :: **key_type**;
//typedef, allow for user defined key types

page <Key, T, Compare, Allocator> :: **mapped_type**; //typedef of the class T

page <Key, T, Compare, Allocator> :: **value_type**; //typedef, the pair <const Key, T>

page <Key, T, Compare, Allocator> :: **size_type**;

page <Key, T, Compare, Allocator> :: **difference_type**;

page <Key, T, Compare, Allocator> :: **allocator_type**;

page <Key, T, Compare, Allocator> :: **key_compare**;

page <Key, T, Compare, Allocator> :: **page_key**;

Page public member functions.

```
page <Key, T, Compare, Allocator> :: page (const Compare& comp = Compare ( ),
                                           const Allocator& = Allocator ( ) );
//constructs an empty page

template <class InputIterator>
page <Key, T, Compare, Allocator> :: page ( InputIterator first,
                                            InputIterator last,
                                            const Compare& comp = Compare ( ),
                                            const Allocator& = Allocator ( ) );
//constructs a page and inserts the values in the range first to last.
```

page <Key, T, Compare> :: **page** (const page <Key, T, Compare, Allocator> & x);
//copy constructor

size_type page <Key, T, Compare, Allocator> :: **size** ( ) const; //returns current size

bool page <Key, T, Compare, Allocator> :: **is_leaf** ( ) const; //true if page is a leaf

bool page <Key, T, Compare, Allocator> :: **is_full** ( ) const;//true if page is full

bool page <Key, T, Compare, Allocator> :: **is_sparse** ( ) const; //true if page is sparse

bool page <Key, T, Compare, Allocator> :: **is_dirty** ( ) const;
//true if page contents have been modified.

size_type page <Key, T, Compare, Allocator> :: **max_size** ( ) const;
//returns maximum size

iterator page <Key, T, Compare, Allocator> :: **parent** ( );

const_iterator page <Key, T, Compare, Allocator> :: **parent** ( ) const ;

iterator page <Key, T, Compare, Allocator> :: **begin** ( );

const_iterator page <Key, T, Compare, Allocator> :: **begin** ( ) const ;

iterator page <Key, T, Compare, Allocator> :: **end** ( );

const_iterator page <Key, T, Compare, Allocator> :: **end** ( ) const;

iterator page <Key, T, Compare, Allocator> :: **find** (const key_type& x);

const_iterator page <Key, T, Compare, Allocator> :: **find** (const key_type& x) const;
//find first object equal to x

iterator page <Key, T, Compare, Allocator> :: **find_if** (query& Query);

const iterator page <Key, T, Compare, Allocator> :: **find_if** (query& Query);
//this algorithm is using an external comparison query.

pair <iterator, bool> page <Key, T, Compare, Allocator> :: **insert**
(const value_type& x);
//inserts a value (the pair <const Key, T) in its proper place.

iterator page<Key, T, Compare, Allocator>:: **insert**
(iterator position, const value_type& x);
//inserts a value in its ordered place but start searching for the place from position.

void page <Key, T, Compare, Allocator> :: **erase** (iterator position);
//delete object at position

size_type page <Key, T, Compare, Allocator>::**erase** (const key_type& x);
//delete all occurrences of object x and returnthe number of objects deleted (zero or one
//for unique key

void page <Key, T, Compare, Allocator> :: **erase** (iterator first, iterator last);
// delete all objects in the range from first to last.

T& page <Key, T, Compare, Allocator> :: **operator** [ ] (const key_type & x);
//allow for directly indexing the objects by their keys

key_compare page <Key, T, Compare, Allocator> :: **key_comp** ( ) const;
// return the functor used in the page (for comparing keys)

iterator page <Key, T, Compare, Allocator> :: **find_split_point** ( );
//return an iterator to the best position to split a page if it was full.

key_type page <Key, T, Compare, Allocator> :: **find_page_key** ( );
//each page is able to extract the key of its contents

Index public type members and data members.

typedef page <Key, T, Compare, Allocator> :: key_type **key_type**;
//the key type used in the index container is obtained from the page container

index < page, Allocator, Container> :: **iterator**; //An iterator to the container

index < page, Allocator, Container> :: **const_iterator**;

index < page, Allocator, Container> :: **reference**;
//type of the reference to the contents of the container: a page reference

index < page, Allocator, Container> :: **const_reference**;

index < page, Allocator, Container> :: **data_type**; // the class page

index < page, Allocator, Container> :: **allocator_type**;

const index < page, Allocator, Container> :: **page_fill_factor**;

Index public member functions.

reference index < page, Allocator, Container> :: **root_page**( );

reference index < page, Allocator, Container> :: **find** (key_type& x, cursor &results);
//key_type is from page container.

const_ reference page < page, Allocator, Container> :: **find**
(const key_type& x, cursor &results) const;

reference index < page, Allocator, Container> :: **find_if** (query &Query , cursor results);
// searching the index using an external query

const_reference index < page, Allocator, Container> :: **find_if**
(query &Query , cursor results);

reference index < page, Allocator, Container> :: **insert**
(const key_type& x, const page :: mapped_type);

reference index < page, Allocator, Container> :: **delete** (const key_type &x);

reference index < page, Allocator, Container> :: **operator** [ ] (const page &p);

Some index private member functions.

page& index < page, Allocator, Container> :: **create_page** ( );

page& index < page, Allocator, Container> :: **check_to_borrow**
(page& left, page& right);

void index < page, Allocator, Container> :: **borrow** (page &from, page &to);

page& index < page, Allocator, Container> :: **check_to_merge**
(page &left, page &right);

void index < page, Allocator, Container> :: **merge** (page &from, page &to);

void index < page, Allocator, Container> :: **fix_page** (page &p);

iterator index < page, Allocator, Container> :: **split_page** (page &p);

iterator index < page, Allocator, Container> :: **fix_sparse_page** (page &p);

iterator index < page, Allocator, Container> :: **pin_page** (page &p);

void index < page, Allocator, Container> :: **flush_page** (page &p);

//For a B+ tree, we need the following additional methods :

iterator page <Key, T, Compare, Allocator> :: **left_sibling** ( );

const_iterator page <Key, T, Compare, Allocator> :: **left_sibling** ( ) const ;

iterator page <Key, T, Compare, Allocator> :: **right_sibling** ( );

const_iterator page <Key, T, Compare, Allocator> :: **right_sibling** ( ) const ;

iterator page <Key, T, Compare, Allocator> :: **rbegin** ( );

const_iterator page <Key, T, Compare, Allocator> :: **rbegin** ( ) const ;

iterator page <Key, T, Compare, Allocator> :: **rend** ( );

const_iterator page <Key, T, Compare, Allocator> :: **rend** ( ) const;

Interface example: B+ tree with integer.

We can implement the complete class from scratch, or include an internal STL container as an implicit container and just use it.

Interface for complete implementation method

```
#ifndef   page_H
#define   page_H
#include <....>

template <class Key, class T, > //using default comparison and default allocator
class page

        {
        public :
                // Constructors
                page ( )          { }     //constructs an empty page

                page ( InputIterator first,  InputIterator last ) { . . . }
                //constructs a page and inserts values in the range first to last.

                page (const page <Key, T>& x ) { }
                //copy constructor

        class iterator
        {
          public:
                iterator (pair<Key, T>  * Initial = 0) :current (Initial ) { } //constructor
```

```
        T& operator* ( )
        {
         return current -> second ;
        }

        const T& operator* ( ) const
        {
          return current -> second ;
        }

        bool operator = = (const iterator& x) const
        {
         return current = = x . current;
        }


iterator& operator ++ ( ) { }

        // . . . with other operators and functions

private:
        pair <Key, T> * current;
        // . . . possibly with other data members

 } // end iterator

class const_iterator { . . . }
//An iterator class that allow read-only access to the container

typedef pair<const Key, T>& reference;
//type of the reference to the contents of the container

typedef const pair<const Key, T>& const_reference;

typedef Key key_type;
//typedef, allow for user defined key types

typedef T mapped_type;
//typedef of the class T

typedef pair<const Key, T> value_type;
//typedef, the pair <const Key, T>

typedef ptrdiff_t size_type;

typedef ptrdiff_t difference_type;
```

```
typedef Allocator allocator_type;

typedef less<T> Compare key_compare;

public:

page ( )        {
   //construct an empty page
                }

page ( InputIterator first, InputIterator last )
        {
        //constructs a page and inserts the values in the range first to
        //last.
        }

page (const page <Key, T, Compare, Allocator> & x )
{ //copy constructor }

size_type  size ( ) const
{ return size ;}

bool is_full ( ) const
{ return is_full ;}

bool is_sparse ( ) const
{return is_sparse ;}

bool is_dirty ( ) const
{return is_dirty ;}

size_type max_size ( ) const
{ return maximum_size ;}

iterator parent ( )
{return parent ; }

const_iterator parent ( ) const
{return parent ;}

iterator begin ( )
{// return iterator to the first element }
const_iterator begin ( ) const
{ //return iterator to the first element }
```

```
iterator end ( )
{ //return iterator to location after last element}
const_iterator end ( ) const{ }

iterator find (const key_type& x) { }
const_iterator find (const key_type& x) const { }

iterator  find_if (query& Query);
const_iterator  find_if (query& Query) const;

pair <iterator, bool> insert (const value_type& x){ }

iterator insert (iterator position, const value_type& x) { }

void erase (iterator position) { }       // delete object at position

size_type erase ( const key_type& x) { }
//delete all occurrences of object x and return the number of objects deleted

void erase (iterator first, iterator last) { }
// delete all objects in the range from first to last.

T& operator [ ] (const key_type & x) { }
//allow for directly indexing the objects by their keys

key_compare key_comp ( ) const{ }
// return the functor used in the page (for comparing keys)

iterator find_split_point ( ) { }
// return an iterator to the best position to split a page.

key_type find_page_key ( ) { }
//each page is able to extract the key of its contents

private:

size_type  size = 0;
bool is_full = false;
bool is_sparse = false;
bool is_dirty = false;
size_type max_size ;
iterator parent ;
```

Interface for implicit container method

We can design the page to be a class that use an implicit container as one of its private data members, namely the map, and thus use the existing map iterator and methods in the implementation of the page (easier).

```
template<class Key, class T>
class page
{
 public:

        typedef map <Key, T> Container

        //import these definitions from map to page. Now they'll apply to the page.
        typedef Container::iterator iterator;
        typedef Container::const_iterator const_iterator;
        typedef  Container::key_type  key_type;
        typedef  Container::mapped_type mapped_type;
        typrdef  Container::value_type value_type;
        typedef Container::size_type  size_type;

 private:
        bool is_full;
        bool is_sparse;
        bool is_dirty;
        size_type max_size;
        size_type current_size;

        iterator parent ;

        Container C;            //implicit map container

 public:
        page ( ) : is_full (0), is_sparse(0), is_dirty(0), current_size (0) { }
        //constructs an empty page

        page (  InputIterator first,  InputIterator last ) : is_full (0), is_sparse(0),
        is_dirty(0) , current_size (0) { }
        //constructs a page and inserts the values in the range first to last.

        page (const page&  x ) ;

        size_type  size ( ) const { return current_size ;}

        bool is_full ( ) const { return is_full ;}

        bool is_sparse ( ) const {return is_sparse ;}
```

```
        bool is_dirty ( ) const {return is_dirty ;}
        size_type max_size ( ) const { return max_size ;}
        iterator parent ( ) { return parent ; }
        const_iterator parent ( ) const { return parent ;}
        iterator begin ( ) {return C.begin ( ) ;}
        const_iterator begin ( ) const {return C.begin ( ) ;}
        iterator end ( ) {return C.end ( ) ; }
        const_iterator end ( ) const{ }{return C.end ( ) ;}
        iterator find (const key_type& x) ;
        const_iterator find (const key_type& x) const ;        //find first object equal to x
        pair <iterator, bool> insert (const value_type& x);
        iterator insert (iterator position, const value_type& x) ;
        void erase (iterator position) ;
        size_type erase ( const key_type& x) ;
        void erase (iterator first, iterator last) ;
        T& operator [ ] (const key_type & x) ;
        key_compare key_comp ( ) const ;
        iterator find_split_point ( ) ;
        key_type find_page_key ( ) ;
} //end class page
```

Usage examples.

```
//to create an index of pages:
index < page <int, smart_pointer>, Physical_Alloc>  An_index ;

//Internally, the index creates pages and bilk-load data into them:
page <int, smart_pointer>  page_1 ;

        //bulk-load the page
        for ( int i = 1 , i = 200 , i ++)
        {
                . . . "read data (a_key, a_smart_pointer) "
              pair <int, smart_pointer >  p = make_pair  (a_key, a_smart_pointer) ;
              page_1 . insrt (p) ;
        }
        . . .

//To insert a pair into the index:
        int  key_x = 515;
        smart_pointer  sp_x ( physical_data_reference ) ;
        . . . = An_index . insert ( key_x, sp_x ) ;

//To search for a key:
        int  key_y = 515 ;
        cursor  result ;
        An_index . find ( key_y , result) ;
        int  First_result =  * result ++ ;
        int  Second_result = * result ++ ;
```

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

C++ SOFTWARE FOR COMPUTING AND VISUALIZING 2-D MANIFOLDS USING HENDERSON'S ALGORITHM

YOUSSEF OMRAN GDURA

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

AUGUST 2001

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: **YOUSSEF OMRAN GDURA**

Entitled: **C++ Software for Computing and Visualizing 2-D Manifolds Using Henderson's Algorithm**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

_____ Chair

_____ Examiner

_____ Examiner

_____ Examiner

_____ Supervisor

Approved _____
Chair of Department or Graduate Program Director

AUG 16 2001

for Dr. Nabil Esmail, Dean
Faculty of Engineering and Computer Science

Abstract

C++ Software for Computing and Visualizing
2-D Manifolds Using Henderson's Algorithm

YOUSSEF OMRAN GDURA

Scientific Computing is an exciting and growing area that provides an important link between Computer Science and the Engineering and Physical Sciences. Today, computer graphics and geometric modeling are used routinely in science, engineering, business, and entertainment. In this thesis we develop object-oriented techniques and software for computing and visualizing implicitly defined manifolds ("surfaces") that arise a wide range of applications. The software differs from existing software for computing such manifolds in its software architecture. Furthermore, its algorithms are based on numerical continuation methods, rather than on subdivision techniques, which allows its practical application to the computation of two-dimensional manifolds in high-dimensional Euclidean spaces. The overall software provides a graphical user interface, algorithms for computing two-dimensional manifolds in higher-dimensional spaces, and graphics routines to visualize the manifolds.

Dedication

In the memory of my parents; Omran Gdura and Halima Fadel, who I wish could take part in celebrating my achievements. I ask God that they forgive me for my choice to pursue my studies away from home when, unknowingly, they needed me the most.

To my wife and my children Raian, Raihan, and Omran. Thank you for your patience, understanding and love. Your support has been invaluable.

Acknowledgments

I would like to express my sincere gratitude to my supervisor professor Eusebius J. Doedel for his support, guidance, patience, and valuable insight, which have made the completion of my thesis possible.

Furthermore, I would like to express my gratitude to my country Libya and to the Ministry of the Education in Libya, for offering me the financial support throughout my studies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Implicit representations of manifolds appear often in science and engineering. They are generally more powerful than parametric representation in describing complicated two-dimensional manifolds. A k-dimensional manifold, which is defined by equations of the form

$$F(x) = 0, \qquad F : R^{n-k} \to R^n,$$

is a topological space with the same properties as Euclidean k-space. Algorithms and software for computing one-dimensional manifolds ($k = 1$) or, equivalently, implicitly defined curves, have been developed years ago (see, for example, [16], [1], [11], [26]). Also, algorithms and software for computing manifolds ($k = 2$) in three-dimensional spaces ($n = 1$) have been well-developed in Computer Graphics and Computer Aided Design. In contrast, there is not much software for computing two-dimensional manifolds embedded in higher-dimensional spaces. Such software, based on simplicial continuation algorithms, works well for manifolds embedded in relatively low-dimensional spaces, but they are inefficient in high-dimensional spaces.

In this thesis we present object-oriented software for the computation of implicitly defined manifolds embedded in (possibly high-dimensional) Euclidean spaces, using an algorithm of Michael Henderson [14]. We also present software for visualizing the resulting manifolds. The computational part of our software is based on a predictor-corrector algorithm. The algorithm was chosen because it has desirable features that cannot be found in other algorithms. It differs from simplicial algorithms (see [5]) in that it is generally more efficient for manifolds embedded in a high-dimensional spaces. Another feature of Henderson's algorithm is its ability to compute manifolds with

global tolerance by using Newton's method, instead of using power series expansions, where the accuracy of the computation is limited due to growth of small errors in the continuation process, (see [8]). Moreover, Henderson's algorithm uses adaptive step sizes, a feature not shared by other algorithms that deal with manifolds embedded in high-dimensional spaces. Another motivation for adopting the algorithm is the fact that it can handle compact manifolds such as spheres and tori.

The development of the software differs from most, if not all, existing two dimensional continuation software in its architecture; it is developed using object-oriented techniques. Object-oriented technology allows us to design our system as a collection of objects, which represent abstract models of geometric objects, such as Disk, Arc, Tangent Vector, etc. These objects can be reused in other applications. The software is divided into three parts. The first part is the Graphical User Interface (GUI) developed using Visual C++; it handles the interaction between the user and the application to provide the necessary information for the computational part. The second part is coded in C++; its task is to find an approximation of a manifold and provide the data needed for graphical rendering. The third part is for visualizing the resulting manifold; it uses the OpenGL graphics library.

This thesis is organized as a self-contained presentation. It contains concepts, definitions, and theorems that are used in our discussion of the algorithm and the development process. No proofs are given; one may refer to the cited references for details. In Chapter 2 we summarize some useful basic geometric concepts and introduce the principles of object-oriented technology. Chapter 3 contains a survey of related literature, while Chapter 4 is devoted to an explanation of the algorithm used in our implementation. The development process of our software is given in Chapter 5. We conclude with some examples of manifolds that are computed and visualized with our software.

Chapter 2

Prerequisites

This chapter introduces elementary concepts that are important for the understanding of the algorithms and the development of our software. Section 1 reviews some elements of vector algebra and vector calculus. Section 2 introduces the concept of continuity and regularity. Section 3 and Section 4 are devoted to curve and manifold representations, with a discussion of some of their geometric properties. Methods for solving linear and nonlinear equations are briefly reviewed in Section 5.

2.1 Vectors

A line segment between two points is called a *vector*; a vector $v \in R^n$ can be represented by an ordered set of n real numbers. For example, a vector $v \in R^3$ can be represented by a triple of real numbers:

$$v = (a_1, a_2, a_3).$$

A vector is a quantity that has both length and direction. The length of a vector v, denoted by $|v|$, is defined as the length of the segment which describes the vector v; thus, nonzero vectors always have a positive length, while the length of zero vector is 0. More precisely, the length of a vector $v \in R^n$ is

$$|v| = \sqrt{{a_1}^2 + {a_2}^2 + \cdots + {a_n}^2}.$$

A vector v is called a *unit vector* if $|v| = 1$. Figure 1 shows some examples of planar vectors.

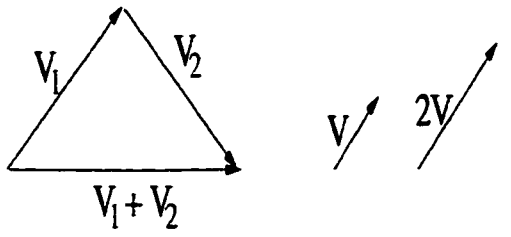


Figure 1: Vectors in Two-Dimensional Space

2.1.1 Operations on Vectors

1. **Addition:** The sum of two vectors is a vector: if $v_1 = (a_1, a_2, \cdots, a_n)$ and $v_2 = (b_1, b_2, \cdots, b_n)$, then

$$v_1 + v_2 = (a_1 + b_1, a_2 + b_2, \cdots, a_n + b_n).$$

2. **Multiplication:** The product of a vector $v = (a_1, a_2, \cdots, a_n)$ by a scalar λ is a vector λv

$$\lambda v = (\lambda a_1, \lambda a_2, \cdots, \lambda a_n).$$

3. **Dot Product:** The dot product of two nonzero vectors v_1 and v_2, denoted by $v_1 \cdot v_2$, is the real number obtained by multiplying corresponding components and adding the resulting products. That is,

$$v_1 \cdot v_2 = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n.$$

The real number can be positive, negative, or zero, depending on the angle between the two vectors. The dot product of two vectors can be also written as

$$v_1 \cdot v_2 = |v_1||v_2| \cdot \cos\theta.$$

If $v_1 \neq 0$ and $v_2 \neq 0$, then the angle between v_1 and v_2 is

$$\theta = \cos^{-1} \frac{v_1 \cdot v_2}{|v_1||v_2|}.$$

Hence, we can say that two nonzero vectors v_1 and v_2 are *orthogonal* if and only if $v_1 \cdot v_2 = 0$. Orthogonal vectors of unit length are called *orthonormal* vectors.

4. **Cross Product:** The cross product of two vectors, denoted by $v_1 \times v_2$, forms a new vector. For example, the cross product of $v_1 = (a_1, a_2, a_3)$ and $v_2 = (b_1, b_2, b_3)$, in R^3 is

$$v_1 \times v_2 = (\ a_2b_3 - a_3b_2 \ , \ a_3b_1 - a_1b_3 \ , \ a_1b_2 - a_2b_1 \ ).$$

The result of the cross product of two vectors, v_1 and v_2, is a vector perpendicular to both vectors, i.e., a normal to the plane spanned by v_1 and v_2.

2.1.2 Linear Dependent and Independent Vectors

Given n vectors $v_1, v_2, \cdots, v_n$ and n scalars $\lambda_1, \lambda_2, \cdots, \lambda_n$, we call the expression

$$\sum_{i=1}^{n} \lambda_i v_i = \lambda_1 v_1 + \lambda_2 v_2 + \cdots + \lambda_n v_n,$$

a *linear combination* of the vectors v_i. The scalars λ_i are called the *coefficients*. The vectors v_i are called *linearly dependent* if there is a linear combination such that $\sum_{i=1}^{i=n} \lambda_i v_i = 0$, where at least one of coefficients does not equal zero. Note that if one of the vectors v_i is a zero vector then obviously the vectors are linearly dependent. The vectors v_i are called *linear independent* if the linear combination $\sum_{i=1}^{i=n} \lambda_i v_i$ equals zero only when $\lambda_1 = \lambda_2 = \cdots = \lambda_n = 0$. We say that $e_1, e_2, \cdots, e_n$ span the space R^n if every vector in R^n can be expressed as a linear combination of these vectors. In R^3, for example, there exist three independent vectors e_1, e_2, and e_3 which span R^3, and any vector $v \in R^3$ can be represented by

$$v = \lambda_1 e_1 + \lambda_2 e_2 + \lambda_3 e_3.$$

In particular, the vectors $e_1 = (1, 0, 0), e_2 = (0, 1, 0), e_2 = (0, 0, 1)$, are the *standard basis* for R^3.

2.1.3 Vector-Valued Functions

A function of the form

$$F(X) = (f_1(X), f_2(X), \cdots, f_n(X)),$$

where $X \in R$, is called a *vector-valued* function of *one variable*. If $X \in R^m$ and $m > 1$, then the function F is called a *vector-valued* function of *several variables*. The scalar-valued functions $f_1(X), \cdots, f_n(X)$ are called the component or coordinate functions of F. The derivative of a function F of one variable is given by

$$F'(X) = \left(\frac{\partial f_1(X)}{\partial X}, \cdots, \frac{\partial f_n(X)}{\partial X} \right).$$

A vector-valued function is said to be differentiable at a point X_0 if and only if its components are differentiable at that point. To differentiate a vector-valued function $F : R^n \to R^m$, we view F as a system of m scalar-valued functions. If we fix the values of all variables except one, then the function becomes a function of one variable; the derivative of such function is called the *partial derivative* of F with respect to this variable. The total derivative $F'(X_0)$ of $F : R^n \to R^m$ at X_0 is a matrix of partial derivatives as shown below:

$$F'(X_0) = \begin{bmatrix} \frac{\partial f_1(X_0)}{\partial X_1} & \cdots & \frac{\partial f_1(X_0)}{\partial X_n} \\ \vdots & \vdots & \vdots \\ \frac{\partial f_m(X_0)}{\partial X_1} & \cdots & \frac{\partial f_m(X_0)}{\partial X_n} \end{bmatrix}.$$

This matrix is called the *Jacobian* matrix.

2.2 Continuity and Regularity

The concept of continuity is based on the behaviour of a given function near a given point (in a "neighborhood" of a point).

Definition A neighbourhood of a point p_0 is the set of points in an open ball centered at p_0. In the case of the real line, a point is ε-close to a given point $p_0 \in R$ if it is in the open interval with center p_0 and length 2ε. A point in R^2 is a neighbourhood of p_0 or ε-close to p_0 if it is interior of a disk with center p_0 and radius ε. In R^3, neighbourhoods are the interior of a sphere with center p_0 and radius ε, see Figure 2.

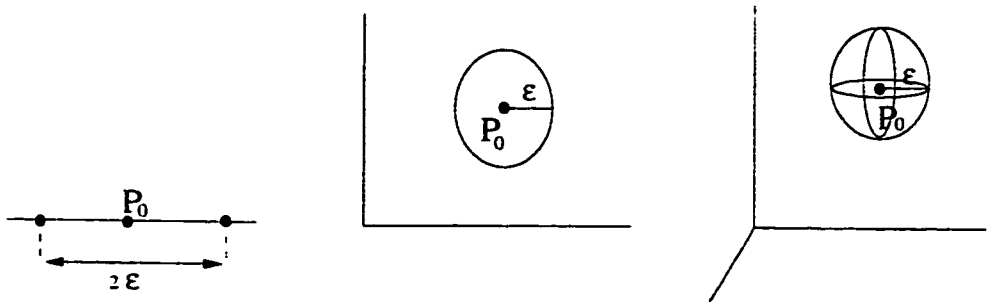


Figure 2: Graphical Representation of Neighborhoods

The next definition gives the notion of a continuous function $F : R^n \to R^m$.

Definition A mapping F is called continuous in the set $U \in R^n$ if F is continuous for all $p \in U$.

Definition If $F : R^n \to R^m$ has a Jacobian matrix at $X_0 \in R^n$, and all partial derivatives $\frac{\partial f_i}{\partial X_j}$ are defined and continuous near X_0, then F is said to be differentiable at X_0.

Definition If $F : R^n \to R^m$ then a point $X_0 \in R^n$ is called a *regular point* if the Jacobian matrix has full rank, i.e., if $Rank(F_x(X_0)) = m$. Correspondingly, a point $X_0 \in R^n$ is called a *singular point* if the rank of the Jacobian matrix $F_x(X_0)$ is less than m.

The first derivative of a function F provides information about the function near a given point X, specifically, whether X is a regular point or a singular point. For example, if F is a real-valued function ($m = 1$), then singular points occur when all the derivatives

$$\frac{\partial F}{\partial X_i}, \qquad i = 1, 2, \cdots, n,$$

vanish.

From the second derivatives we can extract further local information about F; specifically the curvature of the solution manifold of F. The second-order partial derivative, F'', of real-valued functions

$$F : U \in R^n \to R,$$

can be regarded as a symmetric bilinear map

$$F_{xx} : R^n \times R^n \to R,$$

and represented in matrix-vector notation by the *Hessian Matrix*

$$H = \begin{bmatrix} \frac{\partial F}{\partial x_1 x_1} & \frac{\partial F}{\partial x_1 x_2} & \cdots & \frac{\partial F}{\partial x_1 x_n} \\ \frac{\partial F}{\partial x_2 x_1} & \frac{\partial F}{\partial x_2 x_2} & \cdots & \frac{\partial F}{\partial x_2 x_n} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial F}{\partial x_n x_1} & \frac{\partial F}{\partial x_n x_2} & \cdots & \frac{\partial F}{\partial x_n x_n} \end{bmatrix}.$$

Similarly, the second-order partial derivatives (Hessian matrix) of a vector-valued function

$$F : R^n \to R^m \qquad n \neq m,$$

can be viewed as a three-dimensional matrix.

2.3 Curves

A curve can be considered as a path of a moving particle. It can be visualized as a collection of points that are properly spaced and connected by short line segments.

2.3.1 Representation of Curves

Representing a point on a curve as a function of another variable (a "parameter") is called a *parametric representation*. For example, let t denote time, and assume that at any particular time t the point p has a definite location. Also, let α be the vector from the origin to the point p (*position vector*). It follows that the vector function of a plane curve

$$\alpha(t) = (\alpha_1(t), \alpha_2(t)),$$

specifies the vector α; the functions $\alpha_1(t)$ and $\alpha_2(t)$ give the x-component and y-component, respectively. In fact, the components x and y can be written as

$$x = \alpha_1(t),$$

$$y = \alpha_2(t),$$

which are called the parametric equations of a curve in R^2; t being the parameter of the equations. Similarly, parametric curves in n-space require n parametric equations. An example of a curve is the circular helix; its parametric equations are

$$\alpha_1(t) = r\cos t,$$

$$\alpha_2(t) = r\sin t,$$

$$\alpha_3(t) = kt.$$

where $k \neq 0$.

Curves can be also described using *nonparametric representation* (*explicit and implicit* forms). The form

$$y = f(x).$$

is called an *explicit form*. The above form implies that y is explicitly determined as a function of x; that is, for each point of x, only one value is obtained to y. However, this is not always the case. Some curves can only be described using an implicit equation

$$F(x) = c.$$

where c is a constant and $x \in R^n$.

2.3.2 Tangent Vectors to Curves

Let $\alpha : R \to R^n$ be a parametric curve. The first derivative of the function α, denoted by α', describes the tangent vector to α at t, (see Figure 3)

$$\alpha'(t) = (\alpha_1'(t), \cdots, \alpha_n'(t)) .$$

The magnitude of a tangent vector is the norm of the tangent vector α'

$$|\alpha'(t)| = \sqrt{(\alpha_1'(t))^2 + ... + (\alpha_n'(t))^2} .$$

and if $|\alpha'(t)| \neq 0$ then the unit tangent vector is given by

$$T(t) = \frac{\alpha'(t)}{|\alpha'(t)|} .$$

A curve $\alpha = \alpha(t)$ is called a *smooth curve* if the first derivative $\alpha'(t)$ exists and is continuous for all t in the interval under consideration.

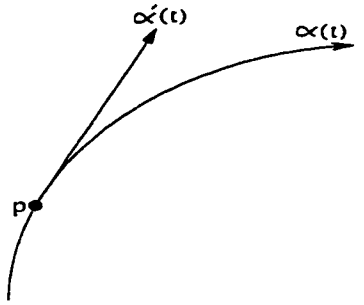


Figure 3: Tangent Vector to a Curve

2.3.3 Curvature of Curves

Curvature measures the bending of a curve. Let C, for example, be a curve in R^2 represented parametrically by $\alpha(s)$, where s is the arc length as illustrated in Figure 4. We can see from Figure 4 that as s varies, the tangent vectors to the curve change

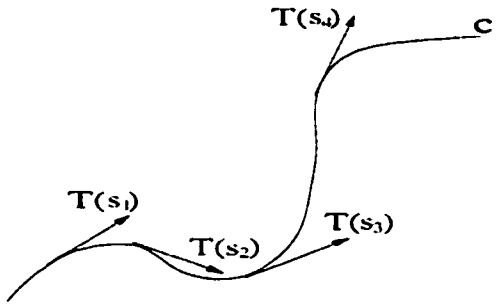


Figure 4: Tangent Vectors to a Curve in R^2

in direction; the more rapidly $T(s)$ changes, the more rapidly the curve bends. The curvature κ of the curve C, with a position vector $\alpha(s)$, can be defined as the absolute value of the first derivative of the tangent vector $T(s)$ or the second derivative of the position vector $\alpha(s)$

$$\kappa(s) = |T'(s)| = |\alpha''(s)| .$$

The above formula implies that if a curve is nearly straight then κ is near zero; for example, the tangent vector T of a straight line is a constant, so $\kappa = T' = 0$. On

other hand, if κ becomes large then we may conclude that the curve bends rapidly.

2.4 Manifolds

Ordinary manifolds, which we see in everyday life, can be thought of as two-dimensional objects, even though these manifolds reside in R^3. For example, if a cylinder is cut lengthwise and rolled out then the cylinder becomes a flat surface. Thus, we can say that these types of manifolds are inherently two-dimensional, and therefore only two coordinates are needed to represent them locally.

2.4.1 Representation of Manifolds

Higher-dimensional manifolds can be described with the same type of equation used for curves, but with a different number of variables or parameters. A manifold $S \in R^3$, for example, can be defined as the set of points whose coordinates x, y, z are the values of the functions

$$\begin{aligned} x &= f(u, v), \\ y &= g(u, v), \\ z &= h(u, v). \end{aligned} \tag{1}$$

Equations 1 are called *parametric equations* with two variables, u and v. *Non-parametric representation* (*explicit* and *implicit* forms) can also be used as alternate definitions of a manifold. An equation of the form

$$z = F(x, y), \tag{2}$$

defines a manifold explicitly, while implicitly defined manifolds can be described using the form

$$F(x, y, z) = 0. \tag{3}$$

Equation 3 can also be written in vector notation as

$$F(X) = 0, \tag{4}$$

where $X \in R^3$ and $0 \in R^1$. Implicitly defined manifolds, such as the one defined above, can also be viewed as a mapping from R^3 into R^1; in formal notation

$$F : R^3 \rightarrow R^1 .$$

In general, a mapping $F : R^n \rightarrow R^m$ is called *smooth mapping* if all partial derivatives of each component function exist and are continuous.

2.4.2 Tangent Space and Normals to a Manifold

Just as a tangent vector was used to approximate a curve near a point, a tangent plane can be used to approximate manifolds near a given point. In order for a manifold S to have a tangent plane T at a point p_0, the function F must be reasonably well-behaved near p_0. The set of all the tangent vectors to S at p_0 forms the tangent space of the manifold S at the point p_0. The tangent vectors at any point on a manifold can be computed using partial derivatives. For example, the vector function

$$F = F(u, v) = [f_1(u, v), f_2(u, v), f_3(u, v)].$$

yields the following partial derivative vector with respect to u

$$v_1 = F_u = \frac{\partial F(u, v)}{\partial u} = \left(\frac{\partial f_1}{\partial u}, \frac{\partial f_2}{\partial u}, \frac{\partial f_3}{\partial u} \right),$$

and the partial derivative vector with respect to v

$$v_2 = F_v = \frac{\partial F(u, v)}{\partial v} = \left(\frac{\partial f_1}{\partial v}, \frac{\partial f_2}{\partial v}, \frac{\partial f_3}{\partial v} \right).$$

Similarly, we can compute the tangent vectors of explicitly defined manifolds at a given point by computing the partial derivatives. For instance, the partial derivatives of Equation (2) with respect to the independent variables, are

$$v_1 = \frac{\partial z}{\partial x} = \frac{\partial F(x,y)}{\partial x},$$

$$v_2 = \frac{\partial z}{\partial y} = \frac{\partial F(x,y)}{\partial y}.$$

For implicitly defined manifolds, the tangent vectors v_1 and v_2 at a point X_0 can be computed by providing two approximate vectors v_1' and v_2', and solving the linear systems

$$\begin{pmatrix} F_x(X_0) \\ (v_1')^T \\ (v_2')^T \end{pmatrix} \begin{pmatrix} v_1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \qquad \begin{pmatrix} F_x(X_0) \\ (v_2')^T \\ (v_1')^T \end{pmatrix} \begin{pmatrix} v_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}.$$

In fact, if the vectors, v_1 and v_2, at the point X_0 are not collinear then they determine a tangent plane to the manifold at X_0. Moreover, their cross product defines a vector orthogonal to the tangent plane; a *normal* vector, (see Figure 5). The unit normal vector is

$$n = \frac{v_1 \times v_2}{|v_1 \times v_2|}.$$

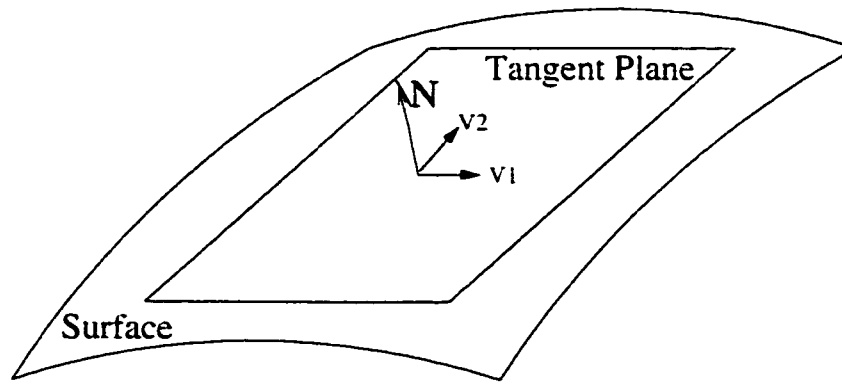


Figure 5: Normal to a Manifold

2.5 Solution of Linear and Nonlinear Equations

A system of equations in the form

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \ , \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \ , \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \ . \end{aligned} \tag{5}$$

is called a system of n linear equations in n unknowns; it can be written in matrix form as

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ & \vdots & & \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} .$$

and in vector notation as

$$Ax = b.$$

2.5.1 The Gauss Elimination Method

The most often used method to solve a linear system $Ax = b$ of n equations in n unknowns is the Gauss Elimination Method. The solution of the system in (5) is an ordered n-tuple $(x_1, \cdots, x_n)$ which satisfies (5); there can be no solution, a unique solution, or infinitely many solutions. The Gaussian Elimination Method reduces the system $Ax = b$ to an equivalent upper-triangular system of the form

$$\begin{bmatrix} a'_{11} & \cdot & \cdot & \cdots & \cdot \\ 0 & a'_{22} & \cdot & \cdots & \cdot \\ 0 & 0 & & \vdots & \vdots \\ 0 & 0 & 0 & 0 & a'_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_n \end{bmatrix} .$$

Reducing the system to upper-triangular form $A'x = b'$ can be achieved using elementary operations (addition, multiplication, and interchanging equations) (see [9]). The final matrix A' is called an upper triangular matrix which can be easily solved by back substitution; starting with setting

$$x_n = \frac{b'_n}{a'_{nn}} ,$$

followed by

$$x_{n-1} = \frac{b'_{n-1} - a'_{n-1,n} x_n}{a'_{n-1,n-1}} ,$$

and so on. The total number of multiplications and divisions needed to solve the above system is

$$\approx \frac{n^3}{3} + n^2.$$

The matrix A in the linear system given in Equation 5 is often fixed, while the right hand side vector; b, changes. In such cases, we can use LU-decomposition to solve for multiple right hand sides b^k. The LU-decomposition of a given matrix can be obtained by saving the multipliers that arise in the Gauss Elimination procedure. The multipliers can be saved in a separate matrix (originally the identity matrix), or in the given matrix if one wants to minimize memory requirements. For now, we use the identity matrix I since this is easier to present. Once the elimination process is complete, and all the multipliers are saved, then we have

$$LUx = Lg .$$

and $A = LU$, $b = Lg$, and $Ux = g$. Given the LU decomposition of A, we can now solve

$$Lg^k = b^k , \qquad \text{for} \quad g^k ,$$

and

$$Ux^k = g^k , \qquad \text{for} \quad x^k .$$

Notice that the decomposition is computed only once and, therefore, the total number of multiplications and divisions for solving a system with m right hand sides is approximately equal to

$$\frac{n^3}{3} + mn^2.$$

A system of nonlinear equations can be written as

$$f_1(x_1, x_2, \cdots, x_n) = 0 ,$$

$$f_2(x_1, x_2, \cdots, x_n) = 0 , \tag{6}$$

$$\vdots$$

$$f_n(x_1, x_2, \cdots, x_n) = 0 .$$

The above system can also be represented in vector function notation as

$$F(X) = 0,$$

where $X \in R^n$. $0 \in R^n$ and F is a vector-valued function. The functions f_i; $i = 1, 2, \ldots n$, can be viewed as mapping a vector $X \in R^n$ to R.
Solutions ("roots") of nonlinear equations are generally computed using iterative techniques because it is normally impossible to get an exact solution in a finite number of steps.

2.5.2 Newton's Method

This method, which is also known as the Newton-Raphson method, uses the tangent to the graph of a function. It is perhaps the most widely used method for approximating the solution of a system of nonlinear equations. Figure 6 illustrates Newton's method for finding a root of a real-valued function of a single variable. The general Newton algorithm is given by

$$x^{(i+1)} = x^{(i)} - \frac{f(x^{(i)})}{f'(x^{(i)})}, \qquad i = 0, 1, 2, \cdots.$$

This iteration converges to a root x^* of $f(x) = 0$ if

- the initial guess $x^{(0)}$ is reasonably close to the root x^*.
- the slope of the graph of f at x^* does not equal zero, that is, $f'(x^*) \neq 0$.
- f is continuous of degree 2 near the root.

Assume that $x^{(0)}$ is an initial guess, and that x^* is the root of $f(x) = 0$, as shown in Figure 6. The algorithm is then as follows:

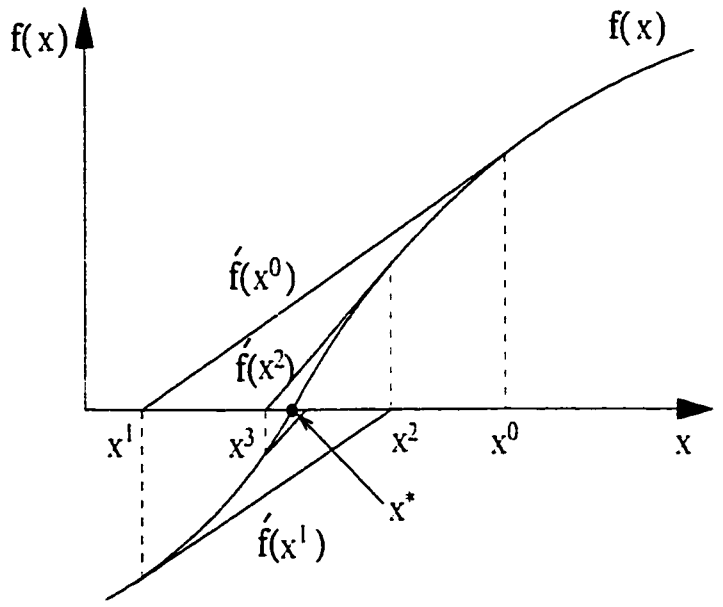


Figure 6: Newton's Method for a Scalar Equation

1 - Set $i = 0$. maxIter $= n$.

2 - Compute $f(x^{(i)})$ and $f'(x^{(i)})$.

3 - $x^{(i+1)} = x^{(i)} - \dfrac{f(x^{(i)})}{f'(x^{(i)})}$.

4 - If ($x^{(i+1)}$ and $x^{(i)}$ are very close) AND ($f(x^{(i+1)})$ is near zero)
then $x^{(i+1)}$ is an approximate root, and the procedure stops.
else
{
$i = i + 1$
If (i <=maxIter) then go back to step 2
else
no root was found. stop.
}

The same technique is applicable to systems of nonlinear equations (n equations in n unknowns). Newton's method solves a system of nonlinear equations by reducing the problem to a set of linear equations. Let the nonlinear system be given by

$$F(X) = 0, \tag{7}$$

where the vector function F has n variables $x_1, x_2, \cdots, x_n$ and n component functions $f_1, f_2, \cdots, f_n$. A solution of this system is then obtained by iteratively solving

$$X^{i+1} = X^i - \frac{F(X^i)}{F'(X^i)}, \qquad i = 0, 1, 2, \cdots.$$

The reason why Newton's method is so widely used is that it converges very rapidly, given that the initial guess is sufficiently close to the root. In fact, Newton's method is quadratically convergent. Roughly speaking, this means that the number of accurate decimal digits doubles at each iteration. When using Newton's method, the following facts must be taken into account:

1. The initial guess must be close enough to the root, otherwise the iteration may not converge or it may converge to a root different from the intended one.

2. The Jacobian matrix F' must be nonsingular at the root x^*: i.e.

$$\det\,(F'(x^*)) \neq 0.$$

3. F must have two continuous derivatives near the root x^*.

2.6 Principles of Object-Oriented Software Development

This section introduces some of the fundamental characteristics of Object-Oriented technology such as the software development process, object-oriented programming, and modeling languages.

2.6.1 Object-Oriented Approach

Early software development was based upon structured and functional approaches. In these approaches, a software system is usually decomposed into functions, then recursively decomposed into subfunctions. The software architecture, which is implemented in this fashion, separates the data from the code. By contrast, object-oriented technology treats a software system as a system made of components or objects which represent real-word entities. Objects are formed from both *state* and *behaviour*. The values of all the attributes (data) of an object at any given time determine the state of the object, and the object's operations (code), which describe the actions and the reactions of that object, determine the behaviour of the object. For example, an object of type Circle can be defined using two attributes (center and radius), and several operations such as `getCenter()`, `setCenter()`, `getRadius()`, `setRadius()`, `draw()`, `move()`, and `resize()`. The values of the two *data members* (the center and the radius) at any given time define the state of the object while the *member functions* determine the behaviour of that object.

2.6.2 Object-Oriented Development Process

The process of developing object-oriented software goes through three phases: *Analysis, Design, and Implementation.* In the first phase, the developer's task is to study and understand the problem to be solved and to determine the objects and the relationship between them. Once the initial analysis is complete, and the software requirements and specifications are established, then the programmers can start the design phase. In this phase, the programmers identify the data members and the member functions of each object. A modeling tool such as the Unified Modeling Language (UML) can be used to assist in class design as well as modeling the software

system. The third phase involves coding and testing the application to make sure that the software fulfils all the requirements and satisfies its intended purpose.

2.6.3 Object-Oriented Programming Concepts

The activity of translating the design of a problem into actual code is called *Object-Oriented Programming* (OOP). This section introduces the principal techniques of OOP in C++.

Classes and Objects

Classes are used to describe groups of objects that have similar attributes and common behaviours. In C++, an instance of built-in data types such as int, float, or char is usually called *variable*, while an instance of a class (user-defined type) is often called an *object* which behaves exactly as specified in the class definition.

- **Class Structure**

  The structure of a class can be divided into two parts: Class *declaration* and Class *implementation*. The declaration and implementation of a class can be written in one file. However, it is a good practice to separate them by saving the class declaration in a header file with **.h** extension, and saving the class implementation in a source file with a **.cpp** extension. Note that if a class was coded in two files (header file and source file), then the header file has to be included in the source file using the keyword `#include`.

- **Class Declaration**

  The declaration of a class must be started with the keyword *class*, followed by the class name, two braces { }, and ended with a semi colon ";". The declaration of the data members of the class, the prototypes of the member functions (methods), and the implementation of the inline functions must be coded inside the class body, which is designated between the two braces {}. If one file is used to declare and implement a class, then the implementation of its member functions, that are not inline functions, must to be placed after the terminator ";". The syntax form for declaring a class is as follows:

```
class cname
{
private:
    // Data members
    ⋮
public:
    // Member functions prototypes and inline functions
    ⋮
};
```

- **Accessing Class Members**

  Accessing a data member or a member function of a class depends on how they are labeled inside the class. C++ provides three different access specifiers: *private*, *public*, and *protected*. Class members, which are labeled as private, can only be accessed from inside the class. For example, private date members can only be accessed via the member functions of that class, and private member functions can only be called by other member functions in the same class. Data members or member functions, that are declared as public, can be accessed from outside the class without any restrictions. A protected access specifier, which is associated with inheritance, is used to allow whatever labeled as protected in a class, whether a member function or a data member, to be accessed by derived classes implementation.

  For encapsulation properties, data members are recommended to be private, as well as critical member functions; whereas typical access functions such as constructors and overloaded operators should be public.

- **Class Constructor and Destructor**

  Every class has two special member functions called *constructor* and *destructor*. Constructors' names and a destructor's name must be the same as the class name, except that a destructor's name is proceed by a "~" symbol. Moreover, constructors and destructors do not have a return value specified, not even void. The task of the constructors is to initialize class data members which can not be initialized inside the class body; thus, they are invoked whenever an object

of the class type is declared. A class destructor is called if an object of that type has to be released from the static memory or deallocated from the heap. Note that if one or more data members of a class are allocated dynamically, then the class destructor has to be implemented in order to release any allocated space on the heap.

Pointers

A *pointer* is a variable that holds the starting address of an object in memory. The object could be built-in data type, user-defined type, or a function. Pointers can be declared explicitly using an indirection operator "*" before the pointers name; for example, the following statement:

```
int *A;
```

states that `A` is a pointer to an integer. In contrast, arrays' names can be considered as an implicit form for declaring pointers. An array's name, in C++, is a constant pointer which points to the first element of the array. For example, the array name `arr` declared in the following statement

```
int arr[3];
```

is a constant pointer, i.e., it always points to the same allocated space in memory. Pointers must be initialized to point to an object of the same type; otherwise it must be assigned one of the two initial values: `NULL` or 0 (zero). For example, the above two pointers can be initialized as follows:

```
int *A = NULL;
int arr[] = {1,5,6,2,9}
```

Such initialization implies that the pointer `A` points to nowhere, while the pointer `arr` holds the starting address of the array `arr`. It is very important to note that if a pointer is an array name, the pointer arithmetic is automatically scaled depending on the size of the objects in the array. The expression (`arr+n`), for instance, represents the address of the $(n+1)^{th}$ element of the array `arr`. The compiler identifies the address by computing an offset in terms of the size(in bytes) of the data type. Thus, the address, which is corresponding to (`arr+n`), is computed by the compiler as

$$\text{the starting address of the array (arr)} + n \times \text{size of(int)}.$$

Dynamic Memory

The phrase "supply on demand" gives good insight into what dynamic memory is about. In many programs, especially when arrays and linked lists are used, the demand for a space in memory is known only at run time. Therefore, instead of preallocating a portrait memory space to store data, dynamic memory gives the programmers the ability to allocate the appropriate space in memory at the run time, and to release it when it is no longer needed. The block of the memory, which is reserved to allocate any space dynamically, is called the *heap*.

Dynamic memory management is the responsibility of the programs. If a class with dynamically allocated data members is used, then member functions such as a copy constructor, an assignment operator, and the class destructor must be implemented properly to handle the dynamic components of the class and to avoid any disastrous results. The copy constructor needs to be implemented to insure that copying one object to another as a result of initialization or passing objects by value will not lead to fatal program errors. Also, the assignment operator needs to be overloaded to avoid similar problems. The destructor must be implemented also to deallocate the space which has been already allocated; otherwise, the space which has been reserved on the heap will not be released, and memory on the heap is wasted. To allocate and deallocate memory dynamically, C++ provides two operators: the *new* operator and the *delete* operator.

The *new* Operator

The functionality of the *new* operator is to make a request for space from the heap. The size of the space needed depends on the object type; if there is enough space for that object, the system reserves the space and returns the starting address of that object; otherwise, the system returns NULL which implies that there is not enough space available for that object on the heap.

The *delete* Operator

This operator releases the memory which has already been reserved by the *new* operator. The syntax is simple, and it comes in two forms:

```
delete A;          // deallocate dynamic object.
delete [] arr;     // deallocate dynamic array
```

Function Overloading

This feature allows two or more functions in one program to have the same name as long as their argument lists are different. The task of determining which function to invoke is left to the compiler. The compiler checks the arguments in the caller function and according to that, the function, whose argument list best matches the argument list of the caller function, will be called. If the compiler did not find an exact match to the argument list, then it uses type conversion, but if no match was found, then the compiler will issue an error message. This feature may ease the programmers task, but they still have to write several functions and do some changes inside the overloaded functions. To avoid such overloaded work, programmers can use another feature in C++ which is called *templates*.

Templates

The use of templates applies to both functions and classes. A template function is a single function with generic type of arguments. The types of the arguments are determined at run time, and an appropriate version of the function is created by the compiler. Template functions are defined by using the keyword *template* followed by an argument list enclosed in angle brackets "$< \cdots >$". The argument list must include at least one general type that is proceeded by the keyword class, and at least one of the general types has to be used in the function argument. The syntax:

```
template < class T >
Return-Value function-Name (T obj1, ···)
{
    //The implementation of the function.
}
```

C++ also provides another feature which is a special type of function overloading; it is called *operator overloading*.

Operator Overloading

In C++, operators such as `+`, `-`, `[]`, `<<` $\cdots$ etc. are not only defined, but most of them can also be overloaded. The ability to redefine operator symbols gives the

programmers the flexibility to handle objects of user-defined types as any other primitive objects. Operators can be redefined as long as one of the operands is of a class type. The general operator function formats are:

ReturnType `operator` operator symbol (*classname* L, *classname* R) // Binary operator

ReturnType `operator` operator symbol (*classname* obj) // Unary operator

Operators can be overloaded as class member functions or as friend free functions.

Free Functions

Ordinary free functions are independent functions. They are not associated with any object of any class type. Thus they can not be used to access private data members of any class. A free function can be called by referencing its name along with its arguments. However, C++ allows free functions to be declared as friend free functions by placing the keyword *friend* before its prototype as shown in the following syntax.

`friend void` *function-Name*(... , ... , ...):

The difference between a free function and a friend free function is that the latter has access to the private data members of the class in which it is declared while the former one does not.

Linked Lists

C++ provides two different list structures, arrays and vectors, to store list elements. However, both structures are inefficient because operations, such as insertion, deletion, resizing, and list reorganization, require significant overhead. In C++, these limitations can be overcome by using a linked list. The data elements of a linked list are stored as independent nodes and linked together without being stored in consecutive memory locations. Thus, each node in a linked list must have two data members: a data value and a pointer to another node. The pointer is the connection which link together the individual nodes in the list; therefore, each node in the list except the last must have a unique successor.

The first node in a linked list is usually identified by a pointer called *front*, and the last is often identified by a pointer called a *rear*. With such a structure, linked lists can be increased, as far as there is a free memory on the heap, by just adding new

elements. Also the operations can be performed in very efficient way. For example, a function with only one statement such as

`Node* getNext() { return next; }`,

can be used to move from one node to the next in a linked list. In another instance, if a node needed to be removed from a list, the process is very simple and can be done in two steps: first, get from the current node (CN) the pointer of the next node to be removed (RM). Then connect the current node (CN) to the successor of the node (RM). The following two statements, for example, will do the job.

```
RM=CN.getNext();
CN.setNext(RM.getNext());
```

Tree Data Structure

Tree data structures are very useful. For example, they can significantly speed up algorithms that repeatedly search for certain features in data. An n-tree is organized so that the root node represents a complete n-dimensional region, and each node in the n-tree corresponds to a sub-region of n-dimensional space. In other words, each node in an n-tree must have at most 2^n leaves or data elements. Quadtrees, for example, are generated by recursively dividing a two- dimensional region into four quadrants as illustrated in Figure 7; thus each node in the quadtree has four leaves that represent the quadrants in the region. Figure 8 shows the quadtree structure of the example given in Figure 7.

Octree data structures are similar to quadtrees. Each node in an octree represents a sub-region of a three-dimensional region (or "box"), and so each node has at most eight nodes. The nodes of an octree should maintain the center of the box, its size, and obviously references to its eight leaves.

2.6.4 Unified Modeling Language

UML is an object-oriented modeling languages which can be used in software development to model a problem in terms of objects, to define the relationship between these objects, and to provide a graphical representation of the system model. Figure 9 shows some of the graphical representations which are used to visualize different

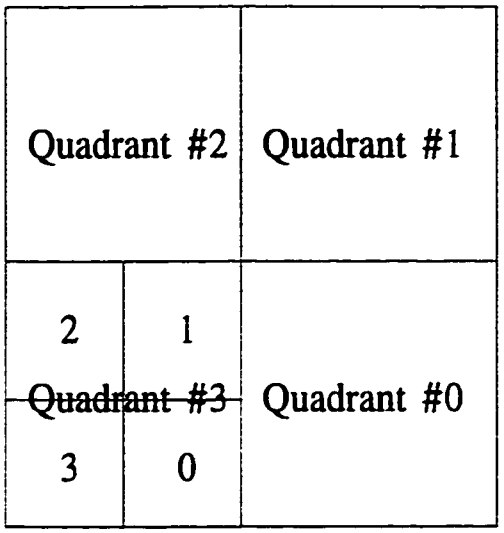


Figure 7: Dividing a Two-Dimensional Region into Four Quadrants

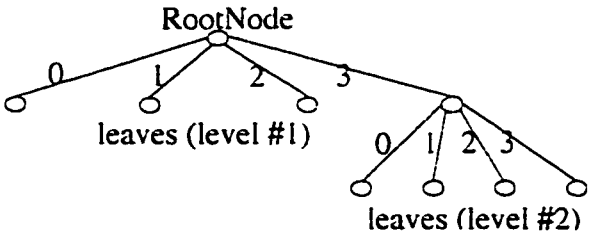


Figure 8: Quadtree Data Structure

relationships among the system components.

The UML provides several types of diagrams to represent the various modeling elements. The details that a diagram can provide depends on the type of diagram; some diagrams give more details than others as we will see in the following two sections.

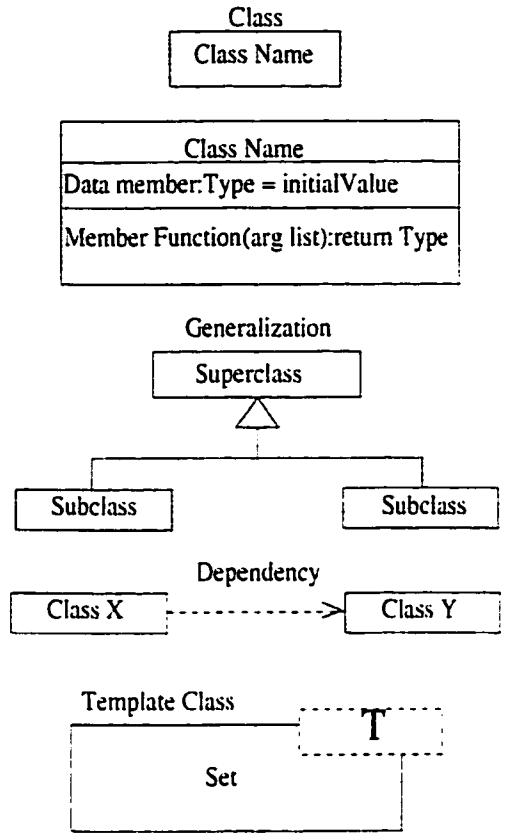


Figure: 9 UML Graphical Representations

Packages

Packages are used to organize modeling elements into groups so that we can easily understand them. They are not like classes; packages have no identity and, hence, we cannot have instances of packages, but rather instances of classes. The UML defines several standard stereotypes that apply to packages (see, for example, [15] for more details). However, we will use only the system standard which specifies a package to represent the entire system being modeled. Figure 10 shows how a package can be represented graphically.

Class Diagrams

Class Diagrams represent the structure of the system or the application in an abstract way. In other words, a class diagram does not provide many details; it shows only

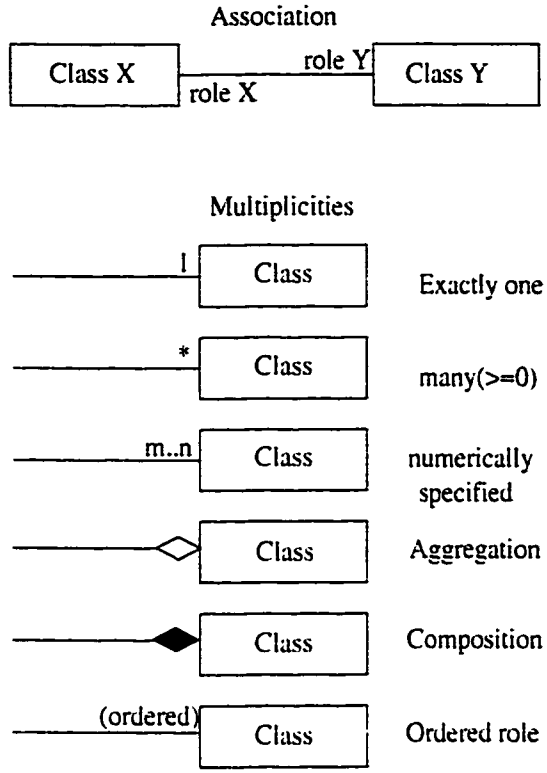


Figure 9: UML Graphical Representations

the classes. of which the system consists, and the relationship between these classes.

UML Graphical Representation of Classes

A class is represented graphically as a rectangle subdivided into three parts as shown in Figure 11. The class name must be identified at the top part, the class data members are placed in the middle, and the bottom portion is designated to define the member functions of the class.

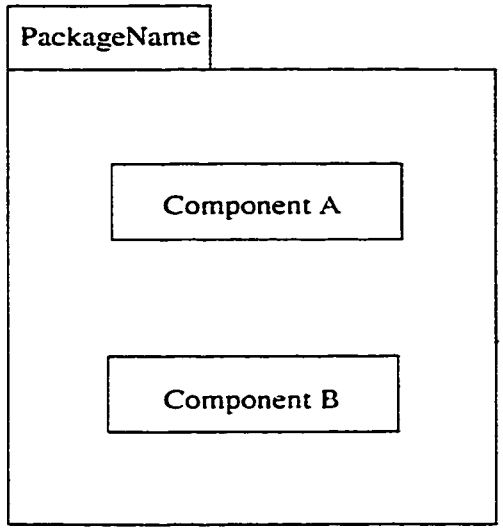


Figure 10: Graphical Representation of a Package

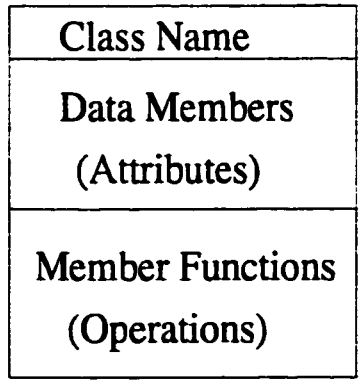


Figure 11: UML Graphical Representation of a Class

Chapter 3

Numerical Continuation Methods for Computing Manifolds

In this chapter we give a brief review of algorithms for computing implicitly defined one-dimensional and two-dimensional manifolds. We do not attempt to give a comprehensive survey of such algorithms; more extensive discussions may be found in [1], [3], [4], [10], [12], [13], and [17], and further references cited therein. Before we introduce these algorithms, we introduce the background that is essential to our later discussion.

In general, k-dimensional manifolds are defined implicitly by a system of equations

$$F(X) = 0, \qquad \text{where} \quad F : R^{n+k} \to R^n. \tag{8}$$

In the mapping $F : R^{n+k} \to R^n$, if $k = 1$, then, generically, the solutions of these equations are curves; if $k = 2$, the solutions of these equations are surfaces.

Implicit Function Theorem (Parameter-Dependent Form)
Let $F : R^n \times R^k \to R^n$. Suppose that $F(X_0, \lambda_0) = 0$. $F_x(X_0, \lambda_0)$ is nonsingular, and F is smooth near (X_0, λ_0). Then, near (X_0, λ_0), the solution of $F(X, \lambda) = 0$ consists of a smooth k-dimensional manifold $X(\lambda)$, i.e.,

$$F(X(\lambda), \lambda) = 0, \qquad X(\lambda_0) = X_0 .$$

Implicit Function Theorem (Parameter-Independent Form)
Let $F : R^{n+k} \to R^n$. Suppose that $F(X_0) = 0$, X_0 is a regular point, and F is

smooth near X_0. Then, near X_0, the solution of $F(X) = 0$ consists of a smooth k-dimensional manifold containing X_0.

3.1 Continuation Methods

Numerical continuation methods have proved valuable in approximating the solution manifold of a system of nonlinear equations $F(X) = 0$, where $F : R^n \rightarrow R^m$. The two broad classes of continuation methods are: Simplicial methods and Predictor-Corrector continuation methods. A brief review of these techniques will be given in the following two sections. We will, unless explicitly stated otherwise, assume hereafter that $F : R^n \rightarrow R^m$, where $n > m$, is a smooth mapping and that the manifolds to be computed consist of regular points.

3.1.1 Simplicial Continuation Methods

These methods rely on a grid of simplices for computing mappings on lower dimensional spaces, while triangulation and labeling techniques are employed for mappings on higher dimensional spaces. Simplicial methods have traditionally been used to approximate one-dimensional manifolds where $F : R^n \rightarrow R^m$ and $m = n - 1$. Also, there are some algorithms for computing a simplicial approximation of two-dimensional manifolds. For further reading, refer to [3], [7], [26], [22], and [25].

3.1.2 Predictor-Corrector Continuation Methods

These methods are based on iteration methods (usually Newton's method) and hence, an initial guess *(predictor)* is needed as a starting point for the *corrector* iteration, which maps the predicted point onto the solution manifold. Under mild regularity conditions, the corrector iteration is guaranteed to converge to a point on the manifold if the starting point is sufficiently close.

1. **Predictor Step**

   In this step, an initial point, a tangent space, and step length must be provided to start the computation. To trace out the solution manifold, the process first computes the tangent spaces to the manifold at new points, and then uses these

tangent spaces to find new starting points, as illustrated in Figure 12 for two dimensions.

2. **Corrector Step**

   Once a starting point is computed, the next step is to map the starting point onto the manifold using an iterative method.

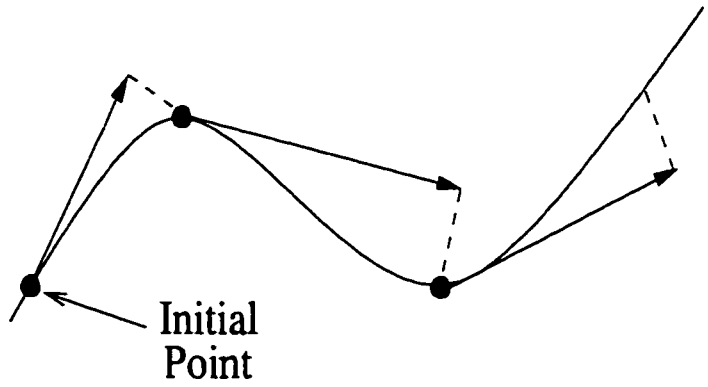


Figure 12: Predictor-Corrector Method Applied to 1-Manifold

3.2 One Dimensional Manifolds

Algorithms and software for computing implicitly defined curves (one-dimensional manifolds) have existed for many years (see, for example, [1], [11], [22], [23], and [26]). A one-dimensional manifold of solutions (a solution "branch") can be parametrized by a single parameter, e.g., arc length. According to the Implicit Function Theorem, if a point $X_0 \in R^{n+1}$ of $F(X) = 0$ is a regular point, i.e., $Rank(F_x(X_0)) = n$, then there exists locally a unique solution branch.

Some parameter continuation methods assume that the solution path can be parameterized by an explicit parameter, such as the last variable of F, and as the chosen parameter varies, one expects branches of solutions. These branches cannot be continued at turning points with respect to that parameter; thus, such methods fail at folds along the curve. For an illustration see Figure 13.

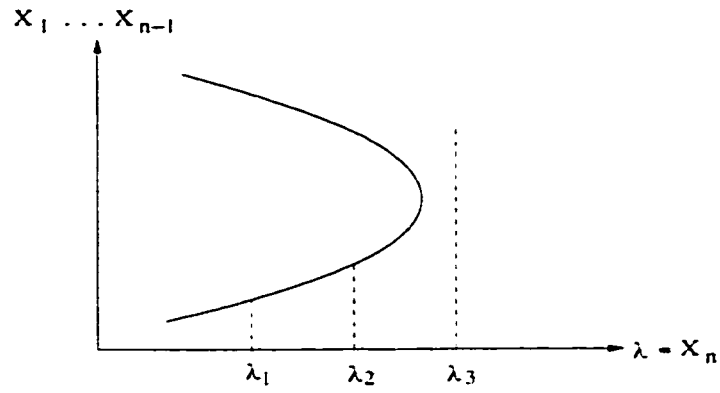


Figure 13: Continuation Parameterized by an Explicit Parameter

3.2.1 Keller's Pseudo-Arclength Continuation

Keller's method allows continuation of a branch past folds. It uses the approximate arclength of the solution branch as the continuation parameter. In this method, the solution curve, as illustrated in Figure 14, is locally parameterized (near X_0) by a *pseudo arclength* parameter Δs. The pseudo arclength between two consecutive solutions on the branch, say X_0 and X_1, denotes the projection of the difference of the two solution vectors onto the tangent vector at X_0. Given a solution X_0 of $F(X) = 0$, a nearby solution X_1 in the direction $\dot{X}_0$ can be computed by solving

$$F(X_1) = 0, \qquad (X_1 - X_0)^T \dot{X}_0 - \Delta s = 0.$$

Using Newton's method to solve these equations leads to the iterative scheme

$$\begin{pmatrix} F_x(X_1^{(v)}) \\ \dot{X}_0^T \end{pmatrix} \Delta X_1^{(v)} = -\begin{pmatrix} F(X_1^{(v)}) \\ (X_1^{(v)} - X_0)^T \dot{X}_0 - \Delta s \end{pmatrix},$$

$$X_1^{(v+1)} = X_1^{(v)} + \Delta X_1^{(v)}.$$

The new direction vector can be obtained by solving

$$\begin{pmatrix} F_x(X_1) \\ \dot{X}_0^T \end{pmatrix} \dot{X}_1 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

One may note that

- The orientation of the branch can be preserved if Δs, which represents the pseudo arclength between two consecutive solutions, is sufficiently small.
- The next direction vector $\dot{X}_1$ can be computed with just one extra solve, using the last LU-decomposition from the Newton iteration.
- The direction vector must be rescaled, so that $| \dot{X}_1 |^2 = 1$.

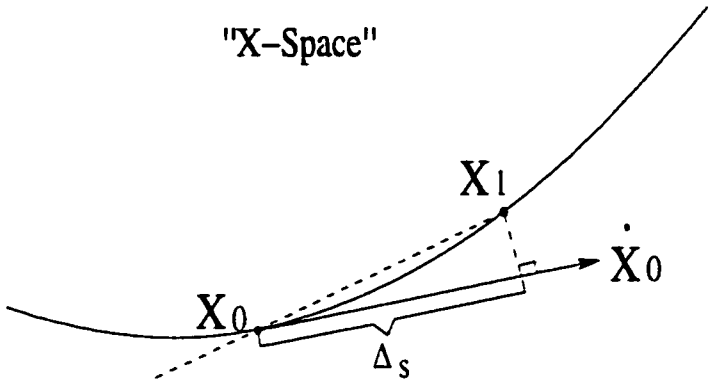


Figure 14: Graphical Interpretation of Pseudo-Arclength Continuation

The AUTO [11] software for continuation and bifurcation problems in ordinary differential equations uses this technique. Applications of AUTO [11] can be found in many papers and books in science and engineering. For an introduction to numerical continuation methods see [10].

3.3 Two-Dimensional Manifolds

Implicitly defined two-dimensional manifolds (*surfaces*) are, in general, defined by equations of the form

$$F(X) = 0, \qquad \text{where } F : R^{n+2} \to R^n \ .$$

Two-dimensional manifolds or 2-manifolds can be classified into two categories: 2-manifolds embedded in low-dimensional space such as in CAD, and 2-manifolds embedded in high-dimensional space which are common in physics and engineering problems. Computing 2-manifolds embedded in low-dimensional space ($n = 1$) is called

iso-surface extraction; they are much more easily computed than manifolds embedded in a space of large dimension. Various algorithms have been proposed and software has been developed for computing 2-manifolds embedded in low dimensional space (see, for example, [19], [20], and [6]). However, the second category of manifolds has not received as much attention, and there are only few existing algorithms and corresponding software. The following are two different algorithms for approximating 2-manifolds.

3.3.1 Simplicial Approximation

Early algorithms for computing a simplicial approximation of a 2-manifolds are presented in [24] and [25]. More recently, Brodzik and Rheinboldt [7] proposed an algorithm for computing simplicial approximations of implicitly defined two-dimensional manifolds. We introduce here the algorithm given in [7] which is an extension of the algorithm developed by Rheinboldt [25]. The original algorithm can be only applied locally and, hence, it can not be used to approximate manifolds globally.

The extended algorithm given in [7] can be described informally in two phases. In the initial phase, a tangent space to the manifold at a starting point X_0 is computed and the first hexagonal neighbourhood is constructed around X_0. The vertices of the hexagonal are then mapped onto the manifold and inserted into a queue. In the second phase, the process continues as follows: Pick a point X_0 from the queue and project its existing neighbourhoods on the tangent space T, as illustrated in Figure 15.

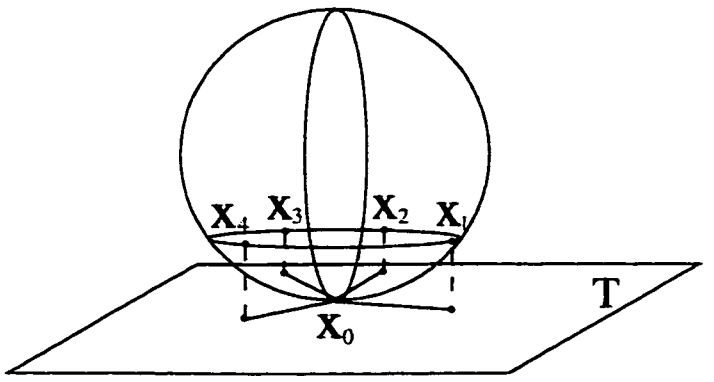


Figure 15: Mapping Neighborhoods of a Point

The projection of the existing neighbourhoods results in a portion of a simplicial neighborhood of X_0 as shown in Figure 16 with a gap G that still needs to be closed. The gap G is the space between the open edges of two simplices. If the angle between the open edges is, for example, less than or equal to $\pi/3$, then G must be closed by identifying its two open edges as neighbours; otherwise, divide G into small triangles, and map the vertices of these new triangles onto the manifold. Then add the mapped vertices to the queue and remove the point X_0 (which became an interior point) from the queue. Accordingly, the queue maintains only the points that *do not* have complete neighbourhoods. The process is terminated once every point on the manifold has a complete neighborhood, that is, when the queue is empty.

The difference between the original algorithm and this algorithm is that in the original algorithm [25], each point has hexagonal neighbourhood of six triangles, while in this version a point may have fewer than six neighbours.

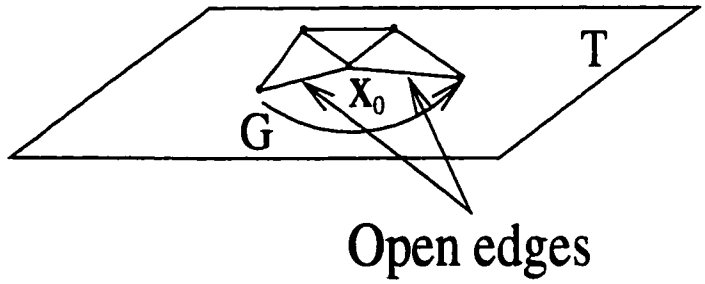


Figure 16: Existing Simplicial Neighborhood

Pseudo-Code for the Algorithm

- Obtain an initial point X_0, and a stepsize.
- Compute a tangent plane T at X_0.
- Form the first hexagonal neighborhood around the starting point X_0.
- Map the points in the tangent plane onto the manifold, and then store them in the queue Q.
- while (Q not Empty)

{

- Pick a new point X_0 from the queue, X_0 = Q[0].
- Compute a new tangent plane T at X_0.
- Project the existing neighbours of X_0 on the tangent plane T.

- Compute the gap G between the open edges of the existing neighbours.
- if $(G \leq \pi/3)$
  No more divisions are needed.
  Link the open edges so that they become neighbours
else
  Divide G into small triangles (new vertices)
  Map the new vertices onto the manifold, and add them to Q.
- Remove the front element in the queue.

}

This algorithm has been implemented in Fortran77 (PITMAN): see [25] for a comprehensive explanation of the algorithm and results.

3.3.2 Predictor-Corrector Approximation

The following algorithm is based on predictor-corrector techniques: it is an algorithm for two-dimensional numerical continuation presented by Meville and Mackay [21]. Just like the simplicial algorithm introduced above, this algorithm is an extension of an existing one-dimensional continuation algorithm given in [2]. It was designed in such a way that it uses an existing software for computing one-dimensional manifolds. In order for this algorithm to use the existing software, the system of equations

$$\begin{aligned} f_1(x_1, x_2, \cdots, x_{n+2}) &= 0, \\ f_2(x_1, x_2, \cdots, x_{n+2}) &= 0, \\ &\vdots \\ f_n(x_1, x_2, \cdots, x_{n+2}) &= 0, \end{aligned} \tag{9}$$

which defines the solution of two-dimensional manifolds, $F : R^{n+2} \rightarrow R^n$, has to be modified by appending an equation $g(x_1, x_2, \cdots, x_{n+2}) = 0$, where $g : R^{n+2} \rightarrow R$, to the system given in Equation 9. The new modified system will then be a system of $n+1$ equations and $n+2$ unknowns, of the form

$$F(X) = 0, \qquad \text{where } F : R^{n+2} \rightarrow R^{n+1} \ . \tag{10}$$

Providing the new system given in Equation 10, the solution set can now be traced with a one-dimensional solver.

Informal Description

Let M be the unexplored region of a manifold with a boundary curve B. The boundary curve is represented as a collection of vertices computed by the one-dimensional solver and stored in a circular linked-list; $(v_0, v_1, \cdots, v_{n-1}, v_0, v_1, \cdots)$; see Figure 17. The algorithm starts by computing an initial boundary, and then the computation is

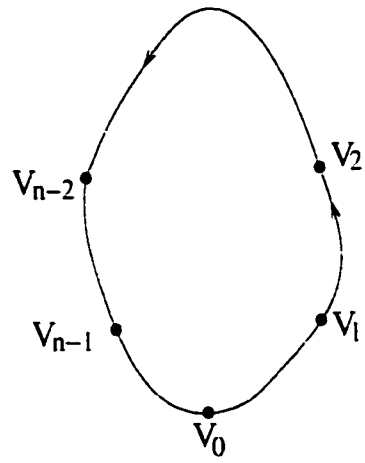


Figure 17: Circular Sequence of Vertices

continued on the unexplored part of the manifold M as follows:

1. Select a point p_0 on the boundary curve B.

2. Construct a sphere-like hypersurface S centered at p_0, which will (generically) intersect M in a one-dimensional curve.

3. Following the boundary curve B starting from p_0 in the direction of its link, let B exit S at a point X and enter S at a point E before returning to p_0 as illustrated in Figure 18. The point X is computed as follows:

   - Find a pair of consecutive points p_1 and p_2 of B that are on opposite sides of S. That is, the function g defining S has different sign when evaluated at p_1 and p_2 (see Figure 19) .
   - Construct a straight line segment L from p_1 to p_2.
   - Determine the zero of the equation for S along the segment L.

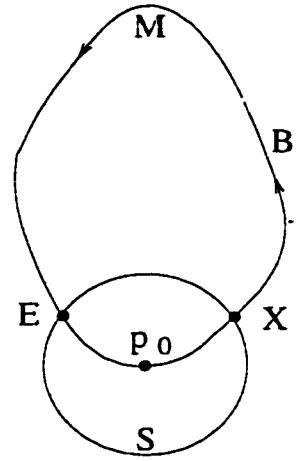


Figure 18: Boundary Curve Intersects a HyperSurface

- Form a new system of $n + 2$ equations and $n + 2$ unknowns, consisting of the system given in Equation 9, augmenting equation aug(p_1), and the equation for S.
- Let the segment intersection computed above be an initial guess, and solve the system formed in the previous step using Newton's methods.
- Add the computed point X to the current boundary as a new vertex in between p_1 and p_2.

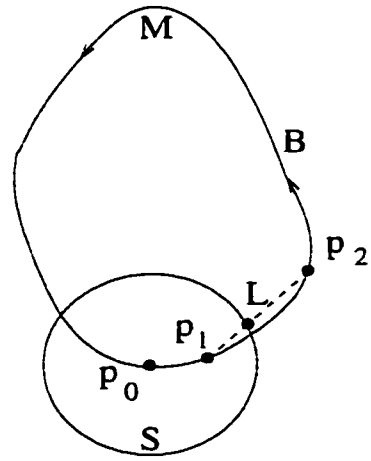


Figure 19: Consecutive Vertices on Opposite Sides of S

4. Compute the point E in the same fashion.

5. Once the two points X and E are computed, the one-dimensional solver can

be applied to determine the path $S \cap M$ defined by appending the equation of S to the system given in Equation 9. Start the process at X, proceed in the direction towards the unexplored region of the M, and terminate the process when it reaches E.

6. Now the portion C of M inside S can be formed by the combination of $B[E, X)$ and $S \cap M$ from X to E (see Figure 20).

7. Update the boundary B by replacing the portion of B from E to X with the portion of $S \cap M$ from E to X.

Repeat this process until the boundary is completely explored. Generally, the above steps are repeated until the boundary B fits entirely inside S, that is, the entry and exit points do not exist. The basic algorithm described above can be improved by

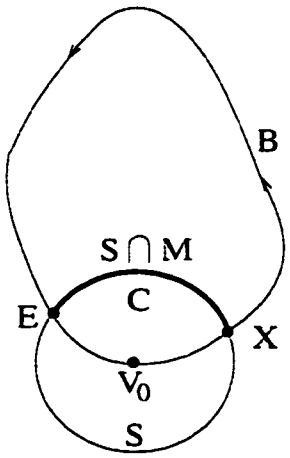


Figure 20: Portion of a Manifold Inside a Hypersurface

appending tangent basis into the augmented equations instead of adding one augmented equation. This leads to a system that has the same number of equations as unknowns.

Chapter 4

Henderson's Algorithm

4.1 Introduction

Let $M \in R^{n-2}$ be an implicitly defined manifold embedded in R^n. That is, any point $X \in M$ satisfies an equation

$$F(X) = 0 \,, \qquad \text{where} \quad F : R^{n+2} \rightarrow R^n \,.$$

In this mapping F, a point X_0 on the manifold M is a regular point if

$$dim(\mathcal{N}(F_x(X_0))) = 2.$$

or equivalently, if

$$dim(\mathcal{R}(F_x(X_0))) = n,$$

where $\mathcal{N}$ denotes the null space, and $\mathcal{R}$ represents the range space.

Henderson's algorithm [14] is designed to compute two-dimensional manifolds (regular implicitly defined manifolds) embedded in higher dimensional spaces. It is based on the predictor-corrector technique and, therefore, it can be thought of as consisting of two main steps that are executed repeatedly. First, at a given point, a local region *(an ellipse)*, that is centered at that point and that lies in the tangent plane, is computed in order to select new points; this is the "predictor" step. Then, an approximation of the mapping of the new points onto the manifold is computed; this is the "corrector" step.

The algorithm (as mentioned before in Chapter 1) overcomes some of the drawbacks

of previously discussed algorithms. For example, it provides an adaptive step size, and it can handle compact topologies such as spheres and tori.

The two fundamental data structures on which the algorithm relies on to construct a manifold are disks and arcs. A manifold data structure is shown in Figure 21. There are other data structures needed in the computation, such as tangent vectors, domain points, and tangent plane points.

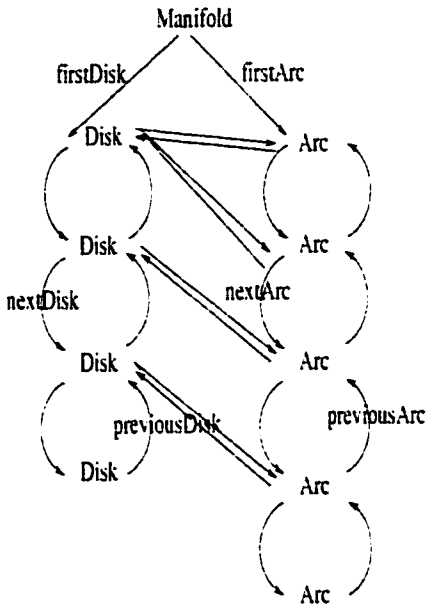


Figure 21: Manifold Data Structure

Let v_1 and v_2 be two vectors, M be a manifold, and X_0 be a domain point. Then the algorithm, in general, works as follows:

1. Find a new point X_0 on the manifold M.

2. Compute the basis, v_1 and v_2, of the tangent plane at the new point X_0.

3. Create a local region, an ellipse, centered at X_0 in the tangent plane. The elliptical region, E, is the set

$$E = \{\ X \mid X = X_0 + sv_1 + tv_2\ ,\ s^2/{R_s}^2 + t^2/{R_t}^2 \leq 1\ \},$$

The projection of the elliptical neighborhood of the tangent plane onto the manifold is called a *disk*, see Figure 22. The disk D is the set

$$D = \{\ X \in M \mid X = X_0 + sv_1 + tv_2 + a(s,t)\ ,\ s^2/{R_s}^2 + t^2/{R_t}^2 \leq 1\ \},$$

where $a(s,t)$ is a mapping from $R^2 \rightarrow R^{n+2}$.

4. Create a new disk centered at X_0.

5. Add the new disk to the manifold data structure after removing the interior parts of the union of the new disk with the old disks. Note that the boundary of the computed piece of M at any given time is the union of the remaining arcs of all disks. An arc, A, is the set

$$A = \{\ X \in M \mid X = X_0 + R_s cos(\theta) v_1 + R_s sin(\theta) v_2 + a(R_s cos\theta, R_t sin\theta)\ \},$$

where X_0 , R_s , and R_t are the values from the associated disk. θ_0 and θ_1 represents the end points of an arc and $\theta \in [\theta_0, \theta_1]$.

Repeat the same steps until no new points are found.

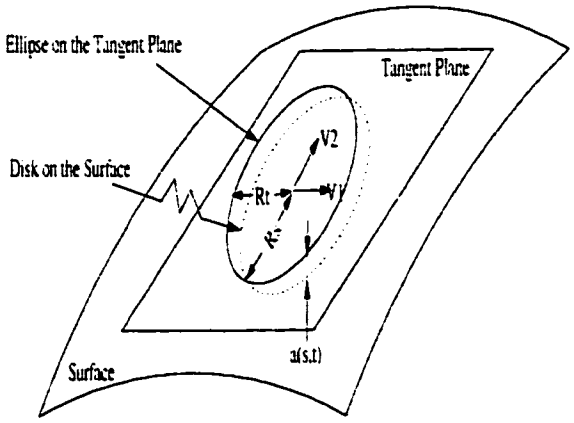


Figure 22: Disk Data Structure

The next section explores some properties of implicitly defined manifolds that are essential for understanding the algorithm. An informal overview of the algorithms is given in a subsequent section.

4.2 Properties of Implicitly Defined Manifolds

4.2.1 Computing a Basis of a Tangent Plane

Let v_1 and v_2 be an orthonormal basis for $\mathcal{N}(F_x(X_0))$ at a regular point X_0 on a manifold $M \in R^{n+2}$. That is

$$\begin{aligned} F(X_0) &= 0, \\ F_x(X_0)\, v_1 &= 0, \\ F_x(X_0)\, v_2 &= 0. \end{aligned}$$

and

$$\begin{aligned} {v_1}^T v_1 &= 1, \\ {v_2}^T v_2 &= 1. \\ {v_1}^T v_2 &= 0. \end{aligned}$$

It is easily shown that the $n+2 \times n+2$ matrix

$$\begin{bmatrix} F_x(X_0) \\ {v_1}^T \\ {v_2}^T \end{bmatrix},$$

is non-singular. Moreover, the matrix remains non-singular for sufficiently close approximations ${v_1}'$ and ${v_2}'$ to v_1 and v_2, respectively.

In our implementation, if X_0 is an initial point, then provided approximate vectors, v_1' and v_2', are used to compute the basis of the tangent plane at X_0. However, if X_0 is a mapped point, as illustrated in Figure 23, then the basis vectors of the tangent plane in which the starting point X^0 was found, are used as approximate vectors. We refer to these approximate vectors as the "basis of a nearby point". Given the approximate vectors, v_1' and v_2', the basis for the tangent plane at a given point X_0 on the manifold can then be computed by solving the linear systems

$$\begin{pmatrix} F_x(X_0) \\ (v_1')^T \\ (v_2')^T \end{pmatrix} \begin{pmatrix} v_1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \qquad \begin{pmatrix} F_x(X_0) \\ (v_2')^T \\ (v_1')^T \end{pmatrix} \begin{pmatrix} v_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}. \qquad (11)$$

The basis, v_1 and v_2, from Equation 11 need not be orthogonal. Thus, the orthogonal basis can be constructed using the Gram-Schmidt technique; that is,

$$v_2 = v_2 - \frac{v_2 \cdot v_1}{|v_1|^2} v_1 .$$

After normalizing v_1 and v_2, we get the orthonormal basis

$$v_1 = \frac{v_1}{|v_1|}, \qquad v_2 = \frac{v_2}{|v_2|}.$$

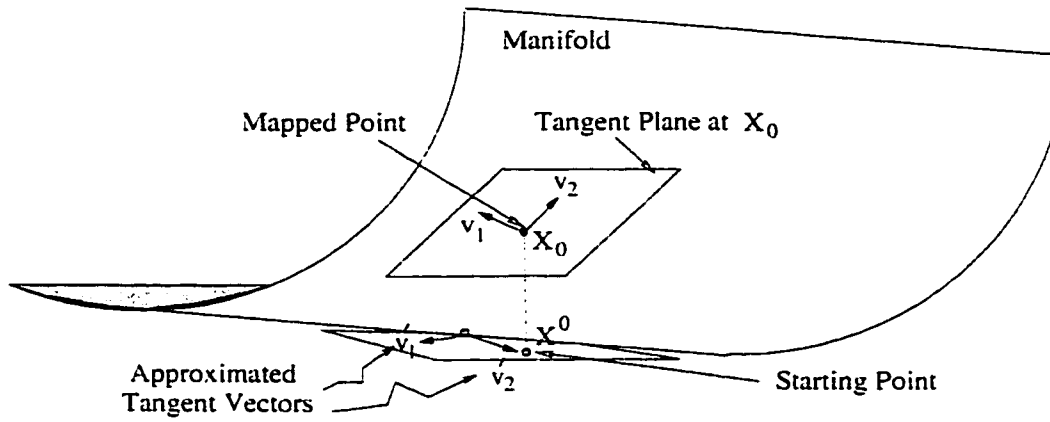


Figure 23: Nearby Tangent Vectors

4.2.2 Mapping a Point onto a Manifold

A mapping from a tangent plane onto a manifold can be constructed once the orthonormal basis, v_1 and v_2, of the tangent plane are known. Figure 24, for example, shows a point (s,t) in the tangent plane near X_0 mapped onto a manifold. The mapped point $p(s,t)$ on the manifold can be easily computed by adding the two vectors u_1 and u_2 as shown in Figure 25, where

$$u_1 = X_0 + sv_1 + tv_2, \qquad \text{and} \qquad u_2 = a(s,t).$$

This is stated in Henderson [14] as follows:

Lemma For each regular point on a manifold M, there is a neighborhood of (0,0) $\in R^2$ in the tangent plane, and a unique mapping $a(s,t)$, from $R^2 \rightarrow R^{n+2}$, such that

$$p(s,t) = X_0 + sv_1 + tv_2 + a(s,t) \in M.$$

The mapping $a(s,t)$ can be approximated using the lower order terms of the Taylor series:

$$a(s,t) = \frac{1}{2}(a_{ss}s^2 + 2a_{st}st + a_{tt}t^2),$$

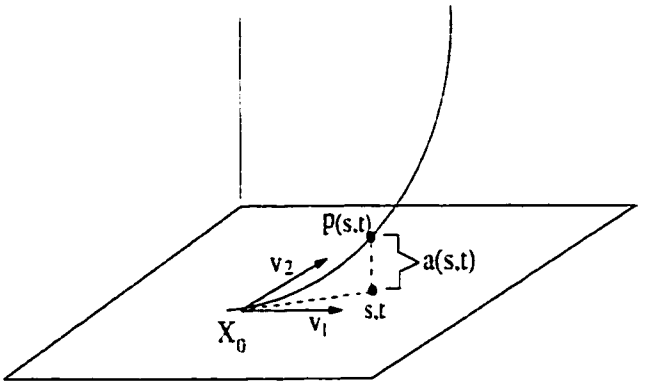


Figure 24: Mapping a Point in the Tangent Plan onto a Manifold

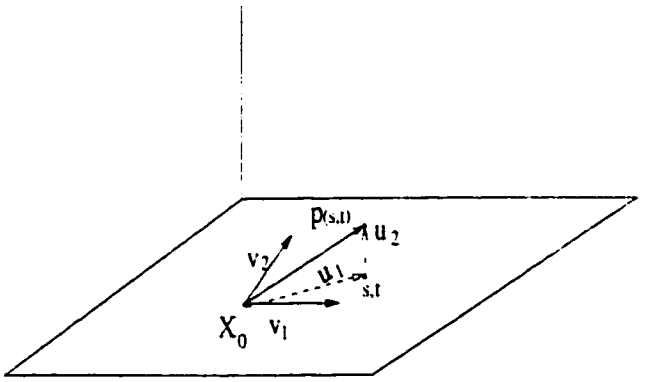


Figure 25: Computing a Point onto a Manifold

where

$$\begin{pmatrix} F_x(X_0) \\ v_1^T \\ v_2^T \end{pmatrix} \begin{pmatrix} \\ a_{ss} \\ \\ \end{pmatrix} = \begin{pmatrix} -F_{xx}(X_0)v_1v_1 \\ 0 \\ 0 \end{pmatrix}, \tag{12}$$

$$\begin{pmatrix} F_x(X_0) \\ v_1^T \\ v_2^T \end{pmatrix} \begin{pmatrix} \\ a_{st} \\ \\ \end{pmatrix} = \begin{pmatrix} -F_{xx}(X_0)v_1v_2 \\ 0 \\ 0 \end{pmatrix}, \tag{13}$$

$$\begin{pmatrix} F_x(X_0) \\ v_1^T \\ v_2^T \end{pmatrix} \begin{pmatrix} \\ a_{tt} \\ \\ \end{pmatrix} = \begin{pmatrix} -F_{xx}(X_0)v_2v_2 \\ 0 \\ 0 \end{pmatrix}. \tag{14}$$

4.2.3 Estimating a Region in which a Mapping is Valid

To ensure that the manifold is computed to a certain accuracy, it is required that the distance between the tangent plane and the manifold be small. That is, $|a(s,t)| < \epsilon$ inside an ellipse centered at X_0, for some $\epsilon > 0$. This constraint can be approximately satisfied by relying on the local vectors a_{ss} and a_{tt}, which estimate the curvature of a manifold at a given point. Thus the axes R_s and R_t of an ellipse can be chosen so that

$$|\tfrac{1}{2}a_{ss}{R_s}^2| = \epsilon \ , \quad \text{and} \quad |\tfrac{1}{2}a_{tt}{R_t}^2| = \epsilon \ .$$

or,

$$R_s \leq \sqrt{\frac{2\epsilon}{a_{ss}}}, \quad \text{and} \quad R_t \leq \sqrt{\frac{2\epsilon}{a_{tt}}}. \tag{15}$$

Note that the curvature of a manifold and the value ofϵ the control the size of a local region, that is, the length of an ellipse's axes. As a result, a greater curvature results in a smaller axes, but if the manifold becomes nearly flat (very small curvature) then the axes can become very large and, hence, in practice a maximum size is imposed.

4.2.4 Projecting a Point onto the Tangent Plane

Let u and v be two vectors. The coordinate of the vector u in the direction of vector v is

$$u_v = u \cdot \frac{v}{|v|} \ ,$$

or,

$$u_v = u \cdot w \ ,$$

where $w = \dfrac{v}{|v|}$ is the unit vector in the direction of v.
Let v_1 and v_2 be the basis of a plane centered at X_0 as illustrated in Figure 26. The coordinates (s,t) of the projection of a point X onto the plane are the coordinates of the vector $(X - X_0)$ in the direction of v_1 and v_2. That is,

$$s = v_1 \cdot (X - X_0) \ , \qquad t = v_2 \cdot (X - X_0) \ .$$

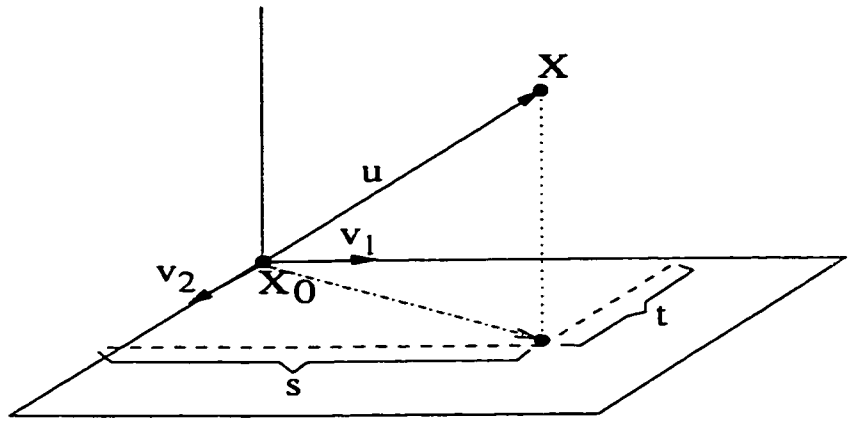


Figure 26: Projecting a Vector onto a Plane

Accordingly, given a point X on a manifold M and the associated disk, the projection of X onto the tangent plane, in matrix-vector notation, is represented by

$$\begin{pmatrix} s \\ t \end{pmatrix} = \begin{pmatrix} 1 & v_1{}^T v_2 \\ v_2{}^T v_1 & 1 \end{pmatrix}^{-1} \begin{pmatrix} v_1{}^T (X - X_0) \\ v_2{}^T (X - X_0) \end{pmatrix},$$

where v_1 and v_2 are the basis of the tangent plane to M at X_0, and X_0 is the disk's center.

4.2.5 Computing the Intersection of Two Disks

One of the important issues in the computation of a manifold is updating the manifold's boundary. The boundary needs to be updated whenever a new disk is created. This requires the computation of the intersection of the new disk with each of the old disks in order to remove the interior parts of the union of any two disks (see Figure 27). The process of updating the boundary can be done in two steps. In the first step the intersection points of the disks are computed and, based on these intersection points, the arcs of the two disks are split. The second step determines which of the split arcs are interior to the union of the two disks, and once done the interior parts are then removed. To show how the intersection points are computed, let the quantities associated with any two disks, $D1$ and $D2$, be given by

$$X_{10} = D1..X_0 , \qquad X_{20} = D2..X_0 .$$

$$
\begin{aligned}
R_1 &= D1.R_s\,, & S_1 &= D2.R_s\,,\\
R_2 &= D1.R_t\,, & S_2 &= D2.R_t\,,\\
v_1 &= D1.v_1\,, & u_1 &= D2.v_1\,,\\
v_2 &= D1.v_2\,, & u_2 &= D2.v_2\,.
\end{aligned}
$$

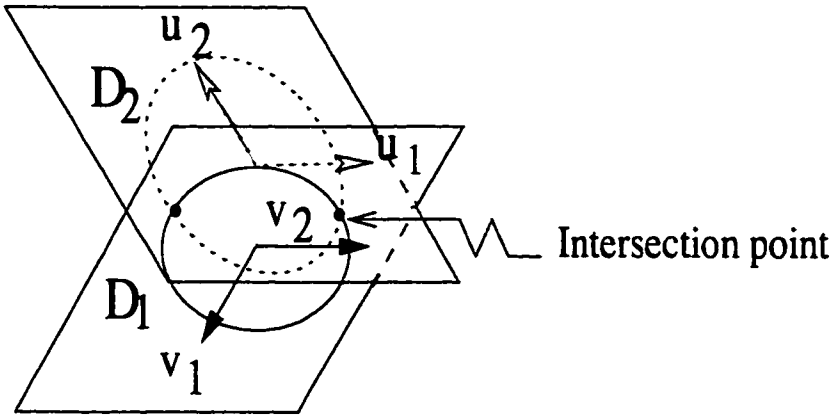


Figure 27: Intersection of Two Disks

Then, as stated in Henderson [14], the points X which are on the boundary of both disks satisfy the system

$$
\begin{aligned}
F(X) &= 0,\\
{v_1}^T\,(X - X_{10}) &= R_1 \cos\theta_1,\\
{v_2}^T\,(X - X_{10}) &= R_2 \sin\theta_1,\\
{u_1}^T\,(X - X_{20}) &= S_1 \cos\theta_2,\\
{u_2}^T\,(X - X_{20}) &= S_2 \sin\theta_2\,.
\end{aligned}
$$

Using the mapping $a_1(s_1, t_1)$ and $a_2(s_2, t_2)$, this is equivalent to solving the system of the following equations

$$
{v_1}^T\,(X_{10} + s_1 v_1 + t_1 v_2 + a_1(s_1, t_1)) = {u_1}^T\,(X_{20} + s_2 u_1 + t_2 u_2 + a_2(s_2, t_2))\,, \quad (16)
$$

$$
{v_2}^T\,(X_{10} + s_1 v_1 + t_1 v_2 + a_1(s_1, t_1)) = {u_2}^T\,(X_{20} + s_2 u_1 + t_2 u_2 + a_2(s_2, t_2))\,, \quad (17)
$$

$$
(s_1/R_1)^2 + (t_1/R_2)^2 = 1\,, \quad (18)
$$

$$(s_2/S_1)^2 + (t_2/S_2)^2 = 1\,, \tag{19}$$

with

$$\begin{aligned} s_1 &= R_1 \cos\theta_0, \\ t_1 &= R_2 \sin\theta_0, \\ s_2 &= S_1 \cos\theta_1, \\ t_2 &= S_2 \sin\theta_1. \end{aligned}$$

The variables s_1 and t_1 can be eliminated:

$$\begin{pmatrix} s_1 \\ t_1 \end{pmatrix} = A \begin{pmatrix} s_2 \\ t_2 \end{pmatrix} + b\,. \tag{20}$$

where,

$$A = \begin{pmatrix} 1 & v_1{}^T v_2 \\ v_1{}^T v_2 & 1 \end{pmatrix}^{-1} \begin{pmatrix} v_1{}^T u_1 & v_1{}^T u_2 \\ v_2{}^T u_1 & v_2{}^T u_2 \end{pmatrix}.$$

$$b = \begin{pmatrix} 1 & v_1{}^T v_2 \\ v_1{}^T v_2 & 1 \end{pmatrix}^{-1} \begin{pmatrix} v_1{}^T (X_{20} - X_{10}) \\ v_2{}^T (X_{20} - X_{10}) \end{pmatrix}.$$

Let

$$R_q = \begin{pmatrix} 1/R_1{}^2 & 0 \\ 0 & 1/R_2{}^2 \end{pmatrix},$$

$$S_q = \begin{pmatrix} 1/S_1{}^2 & 0 \\ 0 & 1/S_2{}^2 \end{pmatrix}.$$

then Equation 18 can be written as

$$\begin{pmatrix} s_1 & t_1 \end{pmatrix} \begin{pmatrix} 1/R_1{}^2 & 0 \\ 0 & 1/R_2{}^2 \end{pmatrix} \begin{pmatrix} s_1 \\ t_1 \end{pmatrix} = 1,$$

or,

$$\begin{pmatrix} s_1 & t_1 \end{pmatrix} R_q \begin{pmatrix} s_1 \\ t_1 \end{pmatrix} = 1. \tag{21}$$

Equation 19 and 21 lead to a pair of two quadratic equations in two variables namely, s_2 and t_2

$$\begin{pmatrix} s_2 & t_2 \end{pmatrix} A^T R_q A \begin{pmatrix} s_2 \\ t_2 \end{pmatrix} + 2 b^T R_q A \begin{pmatrix} s_2 \\ t_2 \end{pmatrix} + b^T R_q b = 1, \tag{22}$$

$$\begin{pmatrix} s_2 & t_2 \end{pmatrix} S_q A \begin{pmatrix} s_2 \\ t_2 \end{pmatrix} = 1. \tag{23}$$

The Equations 22 and 23 can be reduced to a quartic equation in one variable. This allows us to compute s_2 and t_2. The solutions to the quartic equation must be real numbers and the number of solutions must be even, that is, 2 or 4 roots. One can then solve the system given in Equation 20 for s_1 and t_1. For each solution (s_1 , t_1) and (s_2 , t_2), we have the angles θ_0 and θ_1 represent the intersection points on the two disks $D1$ and $D2$, respectively.

$$\begin{aligned} \theta_0 &= \tan(s_1/t_1) , \\ \theta_1 &= \tan(s_2/t_2) . \end{aligned}$$

4.3 Informal Description of the Algorithm

The algorithm initially requires an initial point X_0 on a manifold, approximate tangent vectors v_1' and v_2', and a cube in which the manifold has to be computed. Once this data are provided, the algorithm starts by constructing the first local region as described in the following steps:

- Compute the basis for the tangent plane at X_0 by solving the system given in Equation 11 using the approximate vectors, v_1' and v_2'. Then orthonormalize the basis.
- Determine the size of the elliptical region by first solving the systems given in Equations 12 and 14 to determine the manifold curvature, and then computing the length of the ellipse's axes using the formulas given in Equation 15.
- Create the first disk with the point X_0 as a center.
- Initialize the manifold data structure M with the first disk. Note that the entire disk here represents the initial boundary of the manifold.

Once the initial boundary of the manifold is determined, the algorithm continues by selecting new points on the boundary and computing the unexplored part of the manifold. The computation is terminated if no new points are found. This process goes through several steps as follows:

1. Given that the list of arcs in each disk in M represent part of the boundary, a new point X_0 on the boundary of M can be easily obtained by traversing the list of disks and selecting an endpoint of the first arc found.

2. Let the basis of the tangent plane associated with the disk, on which the new point X_0 was found, be the approximate vectors; v'. Then the basis for the tangent plane at X_0 can be computed by solving the linear systems given in Equation 11.

3. Estimate the size of the local region; an ellipse centered at X_0, in which the mapping is valid. This is can be attained by first solving the systems given in Equations 12 and 14 to get an estimate of the manifold curvature, and then compute the axes of the ellipse using the formula given in Equation 15.

4. Create a new disk with the data obtained from the two previous steps.

5. Update the boundary of the computed manifold by merging the new disk into M as illustrated in the following steps:

   - Get a list of the old disks which represent the manifold M.
   - If the new disk intersects with any of the old disks, then split the arcs of both disks and remove the parts that are interior to their union as illustrated in Figure 28.

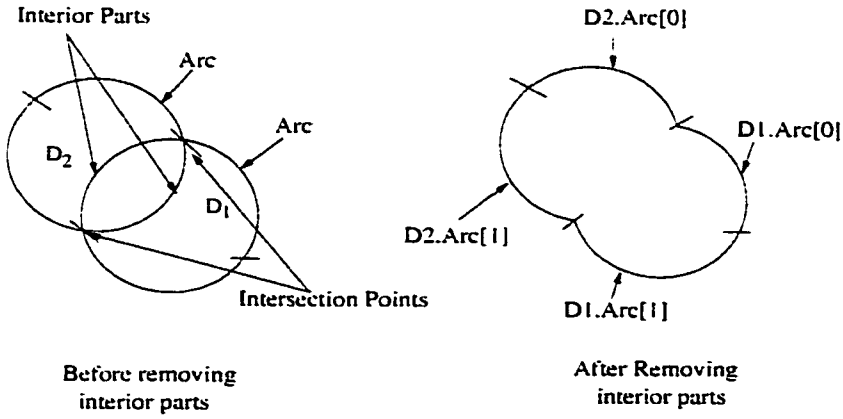


Figure 28: Splitting and Removing Arcs of Two Disks

- Once the process of checking the intersection of the new disk with all of the old disks is finished, then the new disk can be added to the manifold data structure. Note that any remaining arcs of the new disk now become part of the new boundary of the computed manifold.

Repeat steps 1 to 5 until no new points are found.

4.4 Enhancement of the Algorithm

Henderson proposed some refinements in his report in order to reduce the computation time and to have a triangulation of a manifold. The first refinement concerns the complexity of updating the boundary of the computed manifold. The algorithm, as stated, requires that every time a new disk is created, the intersection of the new disk with each of the old disks needs to be checked. Thus, if the total number of disk's computed is N, then the algorithm is $O(N^2)$: that is, for each of the N disks $N-1$ disks must be checked. The overhead work can be reduced by creating a quadtree data structure for the disks. The nodes of the quadtree store the projection of the centers of the disks and a bounding box that contains all of the disks in the leaf nodes. Such a data structure provides a fast way for obtaining a list of the disks which might intersect a new disk. It also eliminates a large collection of disks that are outside the region of intersect (i.e. , not near the new disk being added). The quadtree reduces the complexity of checking the old disks to $log\ N$, and the total complexity to $N\ log\ N$.

The second refinement concerns the triangulation of a manifold. To triangulate a manifold, the references of all disks which intersect each disk must be maintained. This requires that a list of type disk must be added to the disk data structure, and so the list must be created whenever a new disk is created. The process of updating these lists is as follows. If a new disk intersects with an old disk, then the reference of the new disk is added to the list of disks in the old disk, and the reference of the old disk is added to the list of disks in the new disk. Once the computation of a manifold is completed, then the disks which intersect any given disk must be sorted. The sorting is done as follows. Pick a disk from the manifold data structure. Project the vectors between the center of that disk and the centers of each disk in the list onto the tangent plane associated with that disk. Sort their projections in counterclockwise

order. For the case of three disks, $(D1, D2, D3)$ that intersect a disk D, Figure 29 shows how the vectors between the centers are projected.

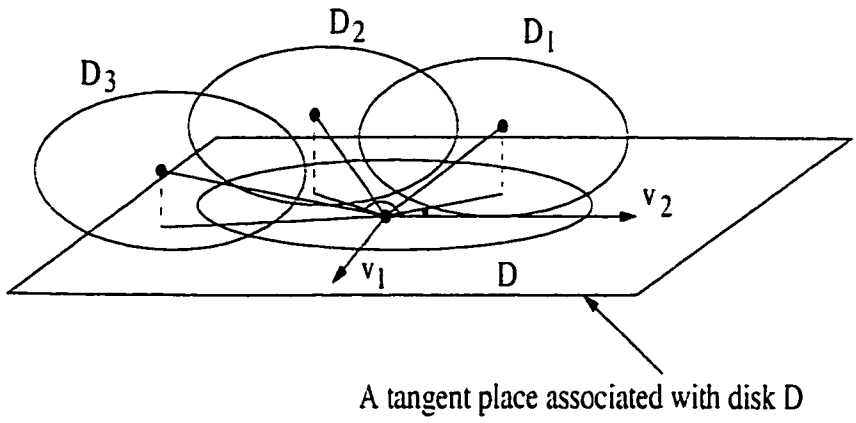


Figure 29: Projecting Vectors of Intersecting Disks

Chapter 5

The Software Development Process

5.1 Introduction

Our software has been given the name **CVIDM**, which stands for Computing and Visualizing Implicitly Defined Manifolds. It was designed using Object-Oriented techniques and was coded using the C++ programming language. The software package is divided into three components: Graphical User Interface, Computation, and Visualization. We start this chapter by introducing the first phase in the development cycle; the analysis phase. Then we move to the design phase to provide more details on the software components; the implementation is given in the last section.

5.2 The Analysis Phase

In this phase, we introduce the objective of the software, data structures, and system specifications. The software package given in Figure 30 shows a graphical representation of the whole system which also includes the OpenGL graphics library as an independent component.

The first requirement of the software is the capability to compute implicitly defined manifolds using Henderson's Algorithm [14]. Once the computation is successfully completed, then the program must be capable of visualizing the resulting manifold using the OpenGL graphics library. The program also provides a graphical user interface (GUI) to allow the user to interact with it. The GUI should state clearly what is required to be entered.

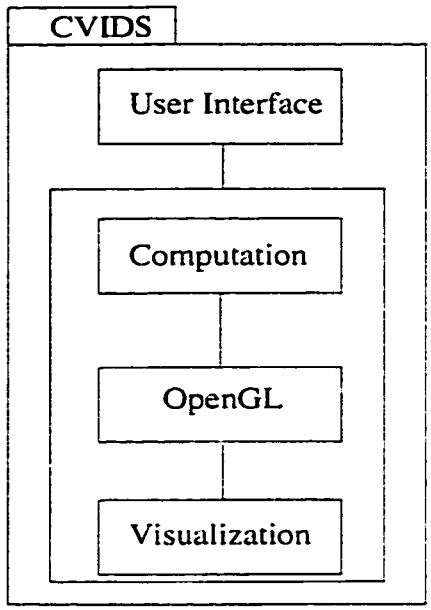


Figure 30: Graphical Representation of the Software Package

5.2.1 Data Structures

In this phase, we identify the classes and the date members of the objects that are needed in the development of our software. More details on these classes are provided in the design and implementation phases.

- **Manifold**

  Only one object of the class type is needed for the computation and the visualization. The object should be created after the first disk is constructed. The disks which form a manifold, as shown in Figure 21, are linked to each other using a doubly-linked list. Thus, if the first disk is known, then the other disks which represent the computed part of the manifold at any time instance can be easily obtained from the doubly-linked list. Also the list of arcs which represent the boundary of the computed manifold can be obtained the same way. As a result, only two data members are required for this class:

  - A pointer to the first disk.
  - A pointer to the first arc.

- **Disk**

  To describe a geometric object of this type we need the center of a disk, two tangent vectors, and the length of its axes. A doubly linked list is also needed to be able to trace the list of the disks which form the computed manifold. Moreover, each disk needs to maintain its remaining pieces (arcs), which represent parts of the boundary of the computed manifold. A linked-list of type disk should be also added to the disk data structure if a triangulation of a manifold is required. Thus, the data members of the Disk class are

  - The disk's center;
  - Two tangent vectors;
  - The length of the axes.
  - A list of the arcs of a disk.
  - A linked-list of the same type.
  - Two references of the same type (for a doubly-linked list).

- **Arc**

  One of the data members of this class must be a reference to a disk, to which an arc belongs. We also need two data members to store the end points of an arc. Moreover, this class must have two data members of type Arc to link arcs to each other.

  - The start and the end points of an arc (in terms of angles).
  - A pointer to the associated disk.
  - Two references of the same type (for a doubly-linked list).

- **Tangent Vector**

  Objects of this class type are used to represent tangent vectors in the domain space. Therefore, an array of size $n+2$ is needed to represent an object of this class type.

- **Tangent Plane Point**

  To be able to define a tangent plane point, we need the following:

  - The two coordinates of a point in a tangent plane.
  - A pointer to the disk to which the tangent plane belongs.

- **Domain Point**

  A domain point is a point in the space R^{n+2} such as a center of a disk. So objects of this class type can be defined using an array of size $n+2$.

- **Disk Intersection**

  An object of this class type is created if an intersection between two disks has occurred. Such an object should maintain the following:

  - The pointers of the two disks which intersect each other.
  - The number of intersection points on the two disks.
  - Two lists to maintain the angles where the intersections occur on each disk.

- **Node**

  This class should be declared as a template class in order to be used to store generic data. It is often defined by two data members: an object and a pointer to a node of the same type. The pointers are used to connect more than one node to form a linked list. Since the space for objects of this class type is allocated dynamically, this class can be made more efficient by reducing the allocated space for each object on the heap. This reduction in the memory space can be achieved by storing the reference of an object instead of the object itself. The class data members are

  - A pointer to an object of generic type.
  - A pointer to a node of the same type.

- **Linked-List**

  Since the linked-lists are built using nodes of generic type, this class must be also declared as a template class. Figure 31 shows a linked-list data structure. linked-lists are essential to our implementation because there are many instances in which the size of the lists needed is known only at run time. For example, the number of disks which intersect a disk is different from one disk to another. Therefore, linked-lists are more efficient than arrays of fixed size. The data members of this class are:

- Two pointers of a generic type.
- A variable to store the size of a list.

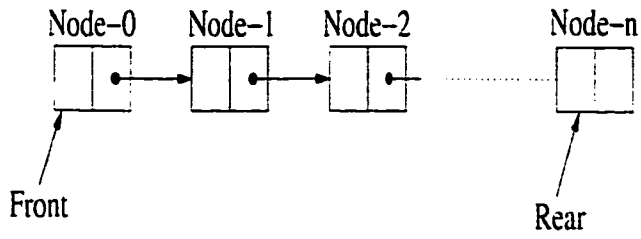


Figure 31: Linked-List Data Structure

- **Octree Data Structure**

  The purpose of using a tree data structure in our software is to reduce the complexity of searching for a list of the disks that may intersect a given region (a new disk). By using an octree data structure instead of the quadtree, which is proposed by Henderson in his paper [14], the total time is reduced from $N \log_2 N$ to $N \log_8 N$ where N is the number of disks. The octree stores the projection of the disks' centers and a bounding cube, which contains all the disks underneath a node. Therefore, the root node in the octree represents the entire computational region, while the leaves represents subcubes. The boundary of the subcubes in the tree is determined based on the maximum length of the two axes of an associated disk. The data members of this class are:

  - A pointer to a disk.
  - Variables of type double to store the boundary of the cube.
  - An array of size eight (8) to maintain the pointers of the leaves.

- **Matrix**

  Matrices are used in many parts of the application. A Matrix object has three data members:

  - Two values to store the size of a matrix (*Rows* and *Columns*).
  - An array of size *Rows* × *Columns*

5.2.2 Software Specifications

- Start the Program

  1. The user is required to provide the system of equations defining the manifold to be computed and its first and second derivatives before running the program.
  2. Once the equations of the desired manifold are entered, then the program can be compiled and executed.

- Obtain Input

  1. The program must first provide a GUI.
  2. The user can then enter the following information: the embedded space dimension (n), the size of a cube in which both the computation and the visualization are done, the type of visualization (solid or wireframe surface), the index of the manifold to be computed, Iso-value, and an initial point (optional).
  3. The program must notify the user if an invalid data is entered.

- Get the Initial Point

  1. If the user did not enter an initial point, the program must search for an initial point on the manifold in the computational space.
  2. If no points are found, the user must be informed that no initial point on the manifold has been found in the given space.
  3. Compute a basis for the tangent plane to the manifold at the initial point.
  4. Create the first disk.
  5. Create the manifold data structure using the first disk.

- Get New Points

  1. Select new points by walking down the list of disks and picking the first endpoint of an arc encountered.
  2. If no new points are found, then stop the computation and start the visualization phase.

3. Compute the basis for the tangent plane to the manifold at the new point.
4. Create a new disk with the new point and the basis obtained in the above two steps.

- Add a New Disk to a Manifold
  1. Get the list of the disks that may intersect the new disk.
  2. If the new disk intersects any of the disks in the above list, then split the arcs of both disks.
  3. Update the list of arcs in both disks after removing the pieces which are interior to the union of both disks.
  4. If a triangulation of the manifold is needed, then the intersecting disks must be added to the list of the disks in each disk.
  5. Repeat the three previous steps until the list of the disks is empty.
  6. Add the new disk to the manifold data structure and to the octree data structure.
- Repeat all steps in the preceding two stages until no new points are found.
- Visualize the Resulting Manifold
  If the wireframe option is selected, then the computed manifold should be visualized using wireframe disks. If a solid manifold is required, then do the following:
  1. Select one disk at the time from the manifold data structure and project the vectors between the center of that disk and the center of each disk intersects it.
  2. Sort the disks in a counterclockwise order based on the projected vectors.
  3. Repeat the steps 1 and 2 until all the disks are sorted.
  4. Draw the manifold using primitive OpenGL objects such as polygons.

5.2.3 Class Diagrams

We have not provided a class diagram for the user interface component as the framework and the classes for it have been provided by the Microsoft Foundation Class

(MFC) Library.

The classes for the computation component and the relationship between them is shown in Figure 32. The class diagram for visualization component is shown in Figure 33.

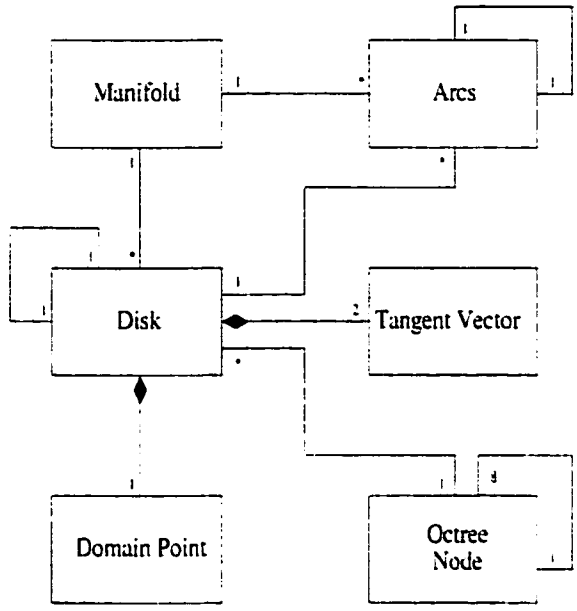


Figure 32: Computation Component Class Diagram

5.3 The Design Phase

In this section, we explore the three components of the software in more detail and introduce several free functions which are essential to the implementation.

5.3.1 Graphical User Interface

The design of our GUI is based on the Windows operating system and implemented using Visual C++ Version 6.0 and the MFC library. For more details on the classes that are used for the GUI see [18].

The GUI is divided into two parts. The top portion is designated to the user input,

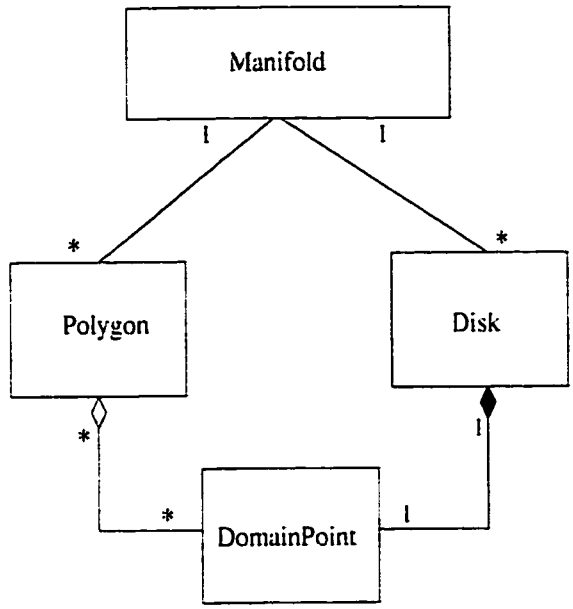


Figure 33: Class Diagram of the Visualization Component

while the bottom portion is reserved for the graphics window in which a resulting manifold is displayed.

5.3.2 Computation Component

Pseudo-Code for the Henderson's Algorithm

- Get an initial point X_0, two approximate tangent vectors: v_1' and v_2'. and the computation space: *cubeSize*.

  getInitialData(X_0 , v_1' , v_2' , cubeSize);

- Compute the basis: v_1 and v_2, for the tangent plane at the point X_0 using the approximate vectors.

  computeBasis(X_0, v_1' , v_2');

- Create the first disk.

  Disk **firstDiskPtr* = new Disk(X_0, v_1 , v_2);

- Instantiate an object of type manifold with the first disk.

  Manifold **manifoldPtr* = new Manifold (*firstDisk*);

Then the algorithm proceeds by searching for new points on the boundary of the computed manifold. The computation will be terminated if no new points are found.

while (getArcEndPoint (*manifoldPtr* , X_0 , v_1' , v_2'))

{

- Compute the tangent basis, v_1 and v_2, at X_0 using the basis of a nearby point as approximate vectors.

  computeBasis (X_0 , v_1' , v_2'):

- Create a new disk.

  Disk **newDisk* = new Disk (X_0 , v_1 , v_2).

- Add the new disk to the manifold.

  addNewDisk (*manifoldPtr*, *newDiskPtr*)

  {

  - Get a list of the disks which represent the computed manifold.

    getDisksList (*listOfDisks[]*):

  - Go over the list and check which disk intersects the *newDisk*. *i=0.*

    while (*listOfDisks[i]* $\neq$ NULL)

    {

    - Compute the intersection of the *newDisk* with i^{th} disk in the list.

      DiskIntersection **interID*=Intersection(*newDisk*, *listOfDisks[i]*):

      if (*interID* $\neq$ NULL)

      {

      - Split the arcs of both disks, j=0.

        while (*arc[j]* $\neq$ NULL)

        {

        splitArc (*arc[j]*, *InterID*, *newArcsList[]*);

        - Remove the interior parts.

        - Update the list of the arcs.

        $j = j + 1$;

        }

      }

      $i = i + 1$

}

- Add the new disk to the manifold.

manifoldPtr → addDisk (*newDisk*).

}

Data Structure of Classes

We provide here the class definitions, UML representations of the classes, and explore the member functions of each class of the computation component in details. However, many of these classes have similar member functions and to avoid repetition, we first introduce these functions and give a general description of each function.

1. `get()'s` and `set()'s` functions are used to access private data members from outside the classes. Most of these functions are implemented as inline functions.

2. The purpose of the constructors is similar across classes. Their task, as stated before, is to initialize class data members. Thus, only the argument list of the constructor(s) is given in the class definitions.

3. Dynamic classes have some member functions such as the destructors, copy constructors, and assignment operators that are similar across classes. Their tasks have already been explained in Section 2.6.3 and, hence, only their function prototypes are given in the class definitions.

4. The index operators; [ ] and (), are overloaded in many classes. They are used, for example, to return an element in a matrix or a coordinate of a point.

In the following class definitions, we define the types of the data members and then introduce the member functions of each class. For the member functions, we first give a description of the function followed by the function prototype.

1. **Range Point Class**

   Objects of this class type represent points in R^n, so each instance of this class type is represented by an array of type double and of size n. The array is allocated dynamically since the range dimension may vary from one system of equations to another. The RangePoint class diagram is given in Figure 34.

- **Detailed Description of the Member Functions**

  This class has three overloaded constructors, each of which allocates dynamically an array of size n which is assigned, by default, the range space dimension (R). The default constructor initializes all the elements of the array with zero. Another constructor receives an array as argument to initialize the data members, and the third constructor initializes all the data members with the same supplied value.

  ```
  RangePoint(int =R);
  RangePoint(double *, int =R);
  RangePoint(double, int =R);
  ```

  The index operator [ ] is overloaded, so that the user can access the coordinates of a point in a normal way.

  ```
  double & operator [ ](int)
  ```

  The prototypes of the destructor, the assignment *operator* =, and the copy constructor of this class are:

  ```
  ~RangePoint();
  RangePoint(RangePoint &);
  RangePoint & operator = (RangePoint &)
  ```

2. **Domain Point Class**

   Objects of this type are also allocated dynamically using an array of size $n + 2$. See Figure 35 for the UML representation of the class.

   - **Detailed Description of the Member Functions**

     The domain dimension; D = $n + 2$, is used as default value in the class constructors. The default constructor allocates an array of size $n + 2$ of type double, and it initializes its elements with zero. The other constructor initializes the allocated array with the values that are passed to the constructor as an array.

     ```
     DomainPoint(int =D);
     DomainPoint(double *,int =D);
     ```

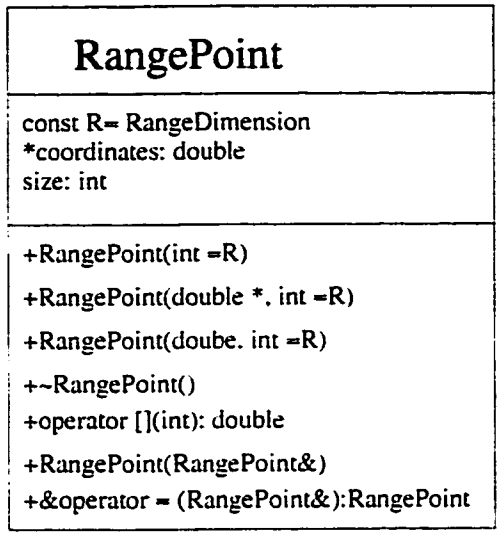


Figure 34: UML Representation of the RangePoint Class

The overloaded comparison operators "*operator ==*" and "*operator !=*", are used to compare two points. The implementation of one can be used to implement the other.

```
bool operator == (DomainPoint &);
bool operator != (DomainPoint &);
```

The following are the prototypes of the class destructor, the assignment *operator =*, and the copy constructor.

```
~DomainPoint();
DomainPoint(DomainPoint &);
DomainPoint & operator = (DomainPoint &);
```

The overloaded index operator [ ] prototype.

```
double & operator [](int);
```

The other member functions, which can be used to obtain or update data members of the class, are

```
setX(int,double);
```

```
getSize();
getX(int)
```

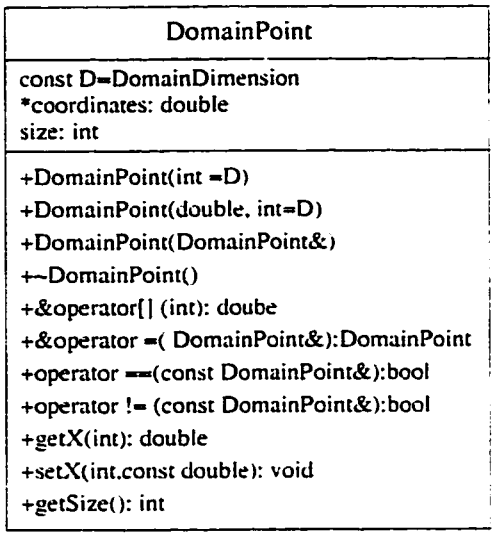


Figure 35: UML Representation of the DomainPoint Class

3. **Tangent Vector Class**

   To be able to store an object of this type, we need an array of type double and size $n+2$. The data members of this class are similar to the *DomainPoint* class, but the member functions are different. For example, for the class *TangentVector* we may need a member function to compute the cross product of two vectors, but there is no need for such a function in the class *DomainPoint*. See Figure 36 for the UML representation of the class.

   - **Detailed Description of the Member Functions**

     The following constructors have the domain dimension a default value.

     ```
     TangentVector(int =D);
     TangentVector(double*, int =D);
     ```

     The class includes also two overloaded functions to compute the cross product of two vectors. In the first function the produced vector is returned as a tangent vector, while in the second function the produced vector is

assigned to an array which is passed as an argument.

```
TangentVector crossPro(TangentVector &);
void crossPro(TangentVector & ,double *);
```

The following member function returns the dot product of two vectors.

```
double dotPro(TangentVector &);
```

This class also provides functions to compute and return a vector length and norm. The prototypes of these functions are as follows:

```
double vectorLength();
double vectorNorm();
```

The overloaded index operator's prototype.

```
double & Operator [](int);
```

4. **Tangent Plane Point Class**

   This class has two variables of type double and a pointer of type disk. The class diagram is shown in Figure 37.

   - **Detailed Description of the Member Functions**

     This class does not have a default constructor. Therefore, to create an object of this type, the coordinates of a point and a pointer to the associated disk must be provided.

     ```
     TangentPlanePoint(double, double, Disk*);
     ```

     The arguments of this function are passed by reference in order to return the two coordinates of a point as arguments.

     ```
     void getTangentPoint(double &, double & );
     ```

     The following function returns the disk to which a point belongs.

     ```
     Disk* getTangentDiskPtr();
     ```

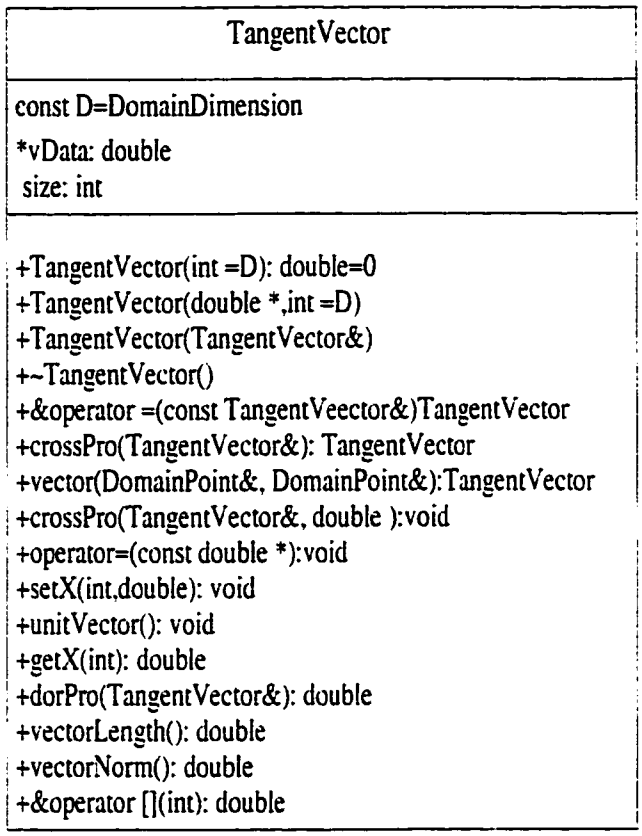


Figure 36: UML Representation of the TangentVector Class

The following function is a private member function: its task is to update the data members of an existing point.

```
void upDateTPP(double, double, Disk *);
```

Each of the following functions returns a coordinate of a point.

```
double getX1();
double getX2();
```

5. **Node Class**

This class is used in our implementation to build linked lists of different types such as *Disks* and *Arcs*. The Node class diagram is shown in Figure 38.

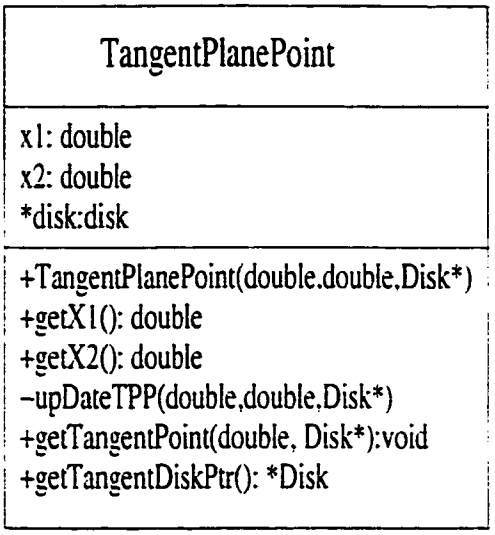


Figure 37: UML Representation of the TangentPlanePoint Class

- **Detailed Description of the Member Functions**

  The class constructor has two arguments. The first argument is a pointer to an object of generic type. and the second argument is a pointer to a node of the same type. Both parameters are given a default value (NULL).

```
Node<T>(T* =NULL, Node<T>* =NULL);
```

  The following are the prototypes of the get()'s and set()'s functions:

```
Node<T> *getNext();
void setValue(T *);
T *getValue();
```

6. **LList Class**

   Two data members of this class are pointers to objects of type *Node*. They maintain the references of the first and last nodes in a linked list. The other data member is a variable of type integer to store the size of a list. This class is a template class and, hence, all its member functions are template functions. See Figure 39 for the UML representation of this class.

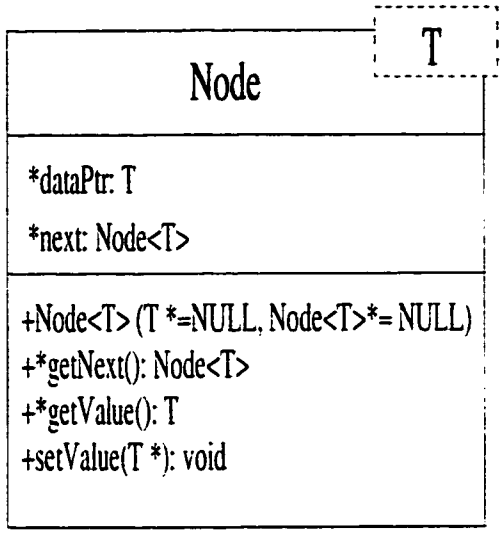


Figure 38: UML Representation of the Node Class

- **Detailed Description of the Member Functions**

  The class has a default constructor which is called to create a linked list.

  ```
  LList<T>();
  ```

  The following function has one argument; a pointer to an object of generic type. The function adds the pointer of a given object to the linked list. It first creates a new node which points to the given object, and then adds that node to the list.

  ```
  void addNewNode(T *);
  ```

  The task of the following two functions is similar. One function removes the front node in a list, while the second functions removes the last node in a list.

  ```
  void removeFrontNode();
  void removeRearNode();
  ```

  The class provides another function to remove a node from a list. This function has one argument which is a pointer to the node to be removed.

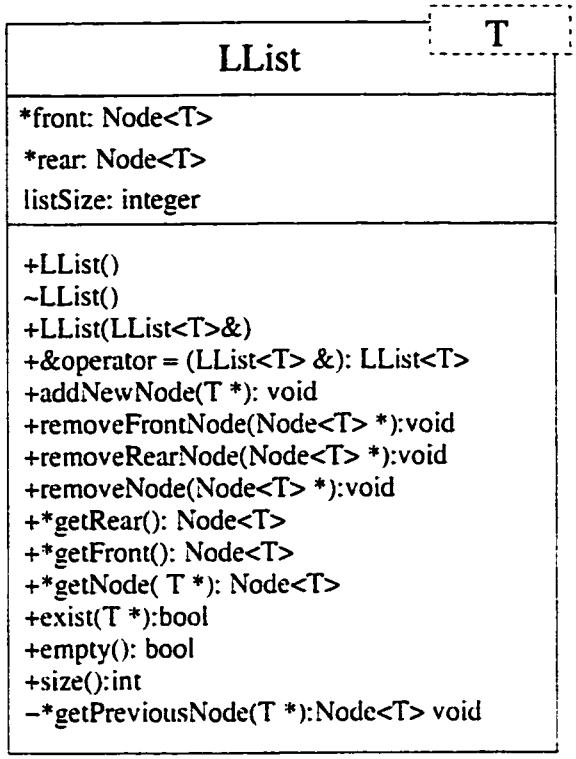


Figure 39: UML Representation of the Linked list Class

If the node to be removed is a front or a rear node, then this function calls one of the above two functions. Otherwise, it removes the node from the list by linking the preceding and the succeeding nodes together.

```
void removeNode(Node<T> *);
```

The following function has no arguments. It returns `true` if a linked list is empty; otherwise, it returns `false`.

```
bool empty();
```

The next function does not have an argument. It returns the number of nodes in a linked list.

```
int size();
```

The next function has one argument; a pointer to an object of generic

type. It checks whether a given object exists in a linked list or not. It returns true if the given object exists in a list; otherwise, it returns false.

```
bool exist(T *);
```

The following function has one argument; a pointer to the list to be deallocated. This function must be implemented in order to release the space which is allocated on the heap for the given list.

```
void deallocateList(LList<T> *);
```

The following are prototypes of other functions provided by the class:

```
LList(LList<T>);
LList<T> &operator = (LList<T> &);
Node<T> *getFront();
Node<T> *getRear();
Node<T> *getNode(T *);
Node<T> *getPreviousNode();
```

7. **Disk Class**

This class contains one object of type *DomainPoint* in order to store a disk's center, and two objects of type *TangentVector* in order to store the basis of the tangent plane associated with the disk. It also includes two pointers of type *Disk* to link disks of a manifold to each other (double-linked list). Moreover, the disk data structure includes two pointers to two different linked lists. One pointer points to a list of type *Disk*; this list is used to maintain the references of the disks that intersect a disk. The list is needed for the purpose of computing the triangulation of a manifold. The other pointer must point to a list of type *AnArc* in order to store the pointers of the remaining arcs of a disk at any given time. See Figure 40 for the class diagram.

- **Detailed Description of the Member Functions**

  The class does not have a default constructor. Thus, to create an object of this type, the disk's center and the basis of the tangent plane must be provided.

  ```
  Disk(DomainPoint &, TangentVector &, TangentVector &);
  ```

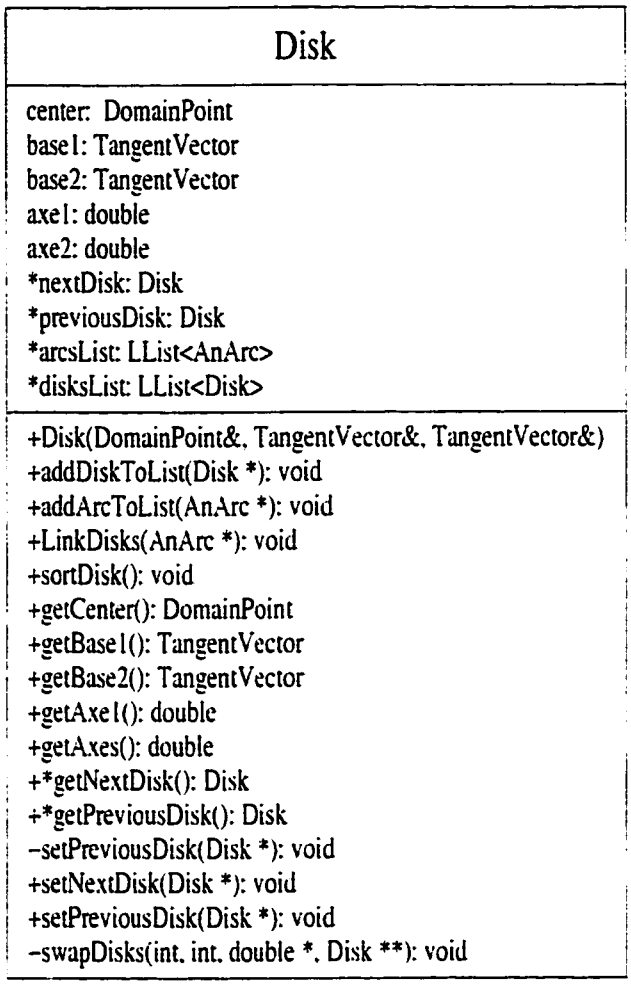


Figure 40: UML Representation of the Disk Class

The next function can be used to add a new arc to the list of the arcs in a disk data structure. It is usually called when an arc is split due to intersections of two disks.

```
void addArcToList(Arc *newArc);
```

The following function has one argument; a pointer to a new disk. If an intersection occurs between two disks, then this function adds the pointer of the new disk to the linked list in the other disk data structure and vice

versa.

```
void addDiskToList(Disk *);
```

To link two disks together, the following function, which takes two pointers of type disk as arguments, can be used:

```
void linkDisks(Disk *, Disk *);
```

The next function is called after the computation of a manifold is complete and before rendering the manifold. Its task is to provide the disks, which intersect a given disk, and their angles in two arrays to the function *orderList()*.

```
void sortDisk();
```

The two arguments of the following function are an array of type double which represents the angles made by the projection of the disks centers and an array that maintains the references of all the disks which intersect a given disk. The function's task is to order counterclockwise the disks in the given array based on the angles, and to store the ordered disks in a new linked list. Once all the disks are sorted, then the function deletes the pointer of the old linked list in that disk data structure, and replaces it with the pointer of the sorted linked list.

```
void orderList(double *, Disk **);
```

This function projects a point on a disk onto a tangent plane. A reference to the point to be projected must be passed as an argument. The function returns the projected point.

```
TangentPlanePoint* projectPoint(DomainPoint *);
```

The following function is a private member function. It is called by the *sortDisk()* function to swap two disks in an array. The position of the two disks to be swapped must be passed to this function.

```
void swapListElements(int,int,double *, Disk **);
```

The task of the remaining member functions in the class can be easily determined from their names. These functions are:

```
double getAxe1();
double getAxe2();
DomainPoint getCenter();
TangentVector getBase1();
TangentVector getBase2();
Disk* getPreviousDisk();
Disk* getNextDisk();
```

8. **An Arc Class**

An object of type *Arc* can be used to represent an entire boundary of a disk or a part of a disk. For example, if the starting point of an arc is 0.0 and the end point is 2π, then that arc represents an entire boundary of a disk; otherwise, it represents only part of it. The class diagram is given in Figure 41.

- **Detailed Description of the Member Functions**

The class has only one constructor which receives three arguments: a pointer to a disk and two variables of type double. The two variables are given default values $(0, 2\pi)$. This constructor can be called to create an arc which represent an entire boundary of a disk without passing the starting and the end points. However, to present part of the boundary of a disk as an arc, the endpoints of the arc have to be passed to the constructor.

```
AnArc(Disk *, double =0.0 , double = 2*PI);
```

The following member function has two arguments: a pointer to the last arc in a list and a pointer to a new arc. In this function, the new arc's pointer is assigned to the data member *next* of the last arc, and the last arc's pointer is assigned to the data member *previous* in the new arc.

```
void linkArcs(AnArc *, AnArc *);
```

The other member functions in this class are:

```
double getTheta1();
```

```
double getTheta2();
AnArc* getNextArc();
AnArc* getPreviousArc();
Disk* getDisk();
void setNextArc(AnArc *arcPtr);
void setPreviousArc(AnArc *arcPtr);
```

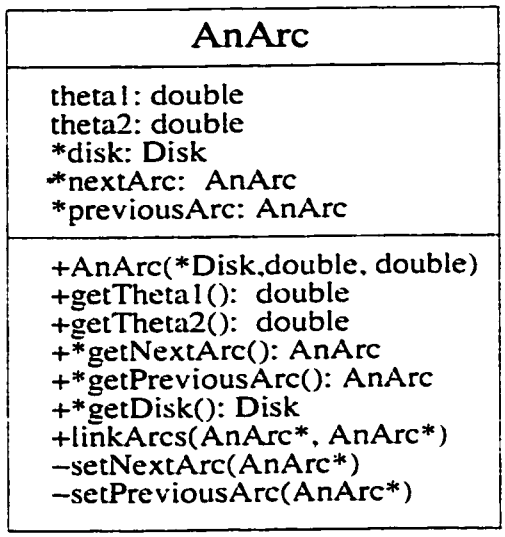


Figure 41: UML Representation of an Arc Class

9. **Manifold Class**

   Only one object of type manifold is needed, and it must be created at the beginning, after the first disk is computed. The object is used frequently in the computation component. It is also used in the rendering of the resulting manifold. See Figure 42 for the Manifold class diagram.

   - **Detailed Description of the Member Functions**

     The class does not have a default constructor, and so the first disk and its entire boundary must be provided in order to instantiate an object of this type.

     ```
     Manifold(Disk *fDisk,Arc *fArc);
     ```

The following function adds a new disk to a manifold data structure. The function must call the function *getLastDisk()* first to get the last disk. Then it adds the new disk to the list of the disks by assigning the pointer of the new disk to the data member *next* of the last disk, and assigning the pointer of the last disk to the data member *previous* of the new disk.

```
void addDisk(Disk *);
```

This function is called whenever a new disk is constructed. It adds the list of the arcs of the given disk to the manifold data structure.

```
void addArcList();
```

The task of the following function is to go over all the disks, which form a manifold, and call the member function *sortDisk()* of the class *Disk*.

```
void sortDisks();
```

The other member functions in this class are:

```
Disk* getFirstDisk();
Disk* getLastDisk();
Arc* getFirstArc();
Arc* getLastArc();
```

10. **Octree Class**

Since each node in an octree data structure represents a cube which contains a disk, one of the class data members must be a pointer of type disk. The other data members needed are variables of type double to store the boundary of a cube (Left, Right, Top, Bottom, Front, and Back sides). Also we need an array of pointers of size eight of type octree to store the references of the leaves of each node. Every cube (or node) in the Octree is divided into eight octants (subcubes), each of which is given an index as illustrated in Figure 43. These indices are used to determine the position of a disk (or a leaf node) in the octree. The Octree class diagram is given in Figure 44.

- **Detailed Description of the Member Functions**

The default constructor will be used at the beginning to create the octree

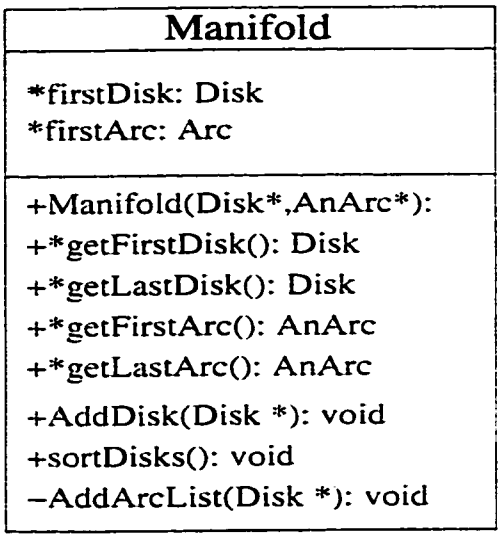


Figure 42: UML Representation of the Manifold Class

and initialize the root node with the entire computation space. If the user does not provide the size of the computation space, then the default value, which is a cube of size ($2 \times 2 \times 2$) centered at the origin, is used.

```
Octree(double =2.0);
```

The following function is a private member function which can be used to check if one cube is inside the other. It returns `true` or `false`.

```
bool inSideCube(Octree *);
```

The next function is also a private member function; its task is to determine the octant in which a given disk must be placed and to return that octant index. In other words, this function walks through the tree until it finds a parent for the new node. Then it determines the appropriate leaf for the new node and returns its index which ranges from 0 to 7.

```
int findPath(Octree *, Octree *);
```

The task of the next member function is to insert a new node into an

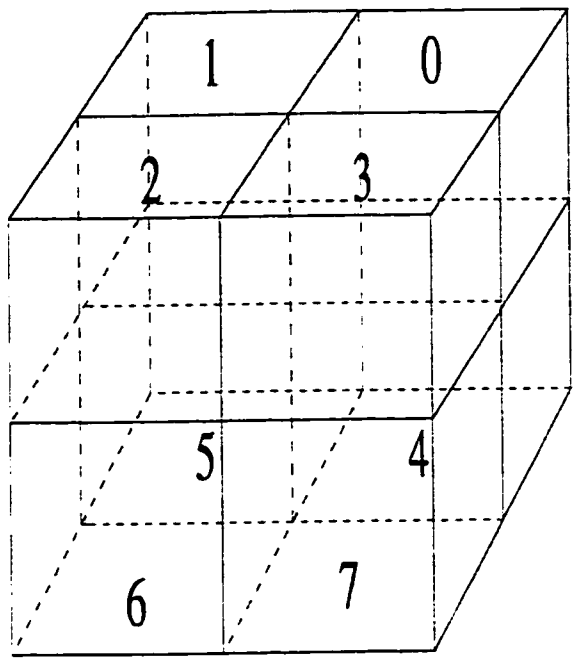


Figure 43: Region of Three-Dimensional Space

octree. It calls the *findPath()* function to determine the appropriate parent node and the index of the leaf in which the new node must be placed. If the insertion is done then it returns `true`; otherwise, it returns `false`.

```
bool insert(Octree *, Octree *);
```

This private member function checks if the calling cube overlaps with the passed cube. The function returns `true` if the two cubes overlap; otherwise it returns `false`.

```
cubeOverLap(double *);
```

The next function is recursive. It has three arguments: a pointer to the root node of a tree, a pointer to a new node, and a pointer to a linked list of type disk. Its task is to find all disks, which might intersect a given disk, and to provide a linked list of their references. It starts from the root node and walks through the tree to select the disks of interest.

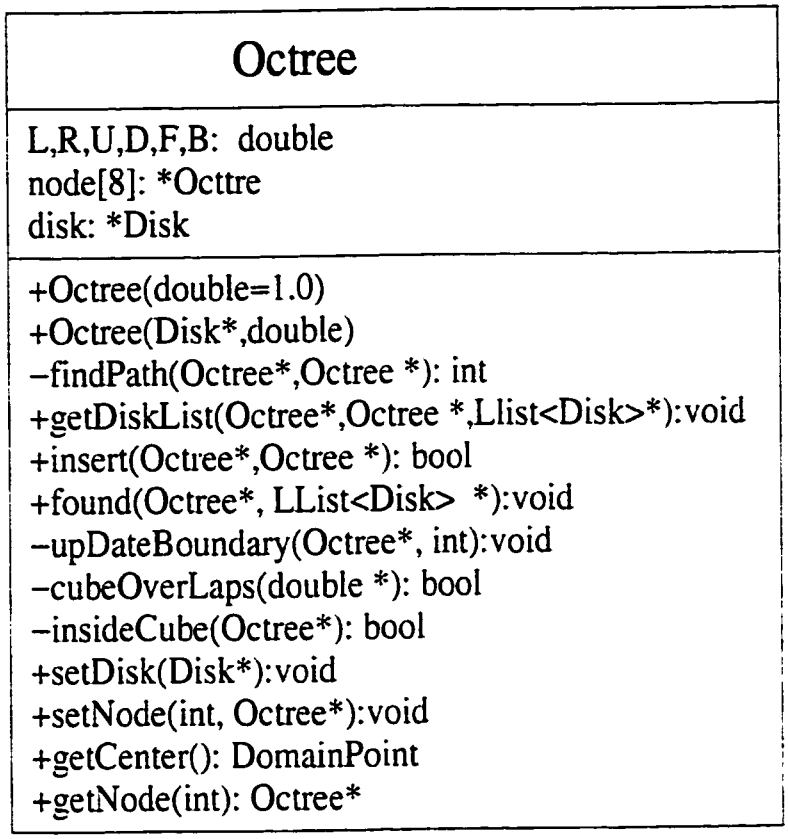


Figure 44: UML Representation of the Octree Class

`void getDiskList(Octree *, Node<disk> *).`

The other member functions, which can be used to get or update data members, are:

```
Octree *getNode(int);
double getSize(char);
DomainPoint getCenter();
setDisk(Disk *);
setNode(int, Octree *);
Octree* operator[](int);
```

11. **Matrix**

The UML representation of the Matrix class is given in Figure 45. Two data members, which represent the number of rows and columns, are of type integer, and other data member is an array of type double of size (number of Rows × number of Columns). The space for the elements of a matrix is allocated dynamically. This class includes only simple operations and functions; more functions are provided in the next class.

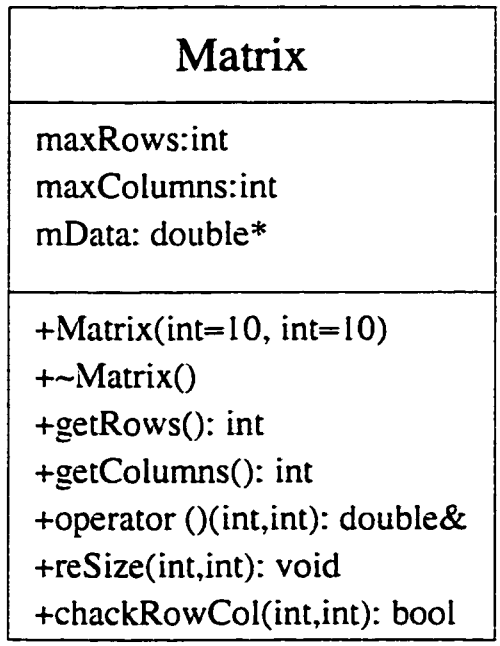


Figure 45: UML Representation of the Matrix Class

- **Detailed Description of the Member Functions**

  The class constructor has two integer arguments; their default value is 10. The constructor creates a matrix of any size. As default, it allocates space on the heap for a matrix of size (10 × 10).

  `Matrix (int =10, int =10).`

  The following function can be invoked to resize an existing array. The function, first maintains a reference to the space where the elements of the

existing matrix are stored. Second, it allocates new space on the heap. Then it copies the old matrix to the new one and, finally, it deletes the space which is reserved on the heap for the old matrix.
`reSize(int, int).`

The overloaded index operator *operator ()* can be used in a normal way to access the elements of an existing matrix using the following functions:
`double & operator () (int,int).`

The following functions can be used to obtain the number of rows and columns, respectively.
`int getColumns().`
`int getRows().`

12. **MatrixOp**
This class has no data members: it includes only operations and methods for matrix manipulations such as multiplication, addition, the Gauss Elimination method, LU-decomposition method, and so on.

- **Detailed Description of the Member Functions**
The task of the following class member functions can be determined from their names. For example, the function *AddMat(···)* adds two matrices; the function *LUD(···)* compute the LU-decomposition. For more details refer to [27].
`enum MatErr(matErr-None, matErr-size, matErr-Sigular,`
`matErr-IllCondition, matErr-Iterlimit),`
`MatErr Gauss(Matrix &, Matrix &, int, int).`
`MatErr LUD(Matrix &, Matrix &, intVect &,int,int).`
`MatErrType MatInverse(Matrix &, int).`
`void LUBackSubst(Matrix &, Matrix &, intVect &, int,`
`Vect &).`
`void LUInverse(Matrix &, Matrix &, intVect &,int).`
`double LUDeterminant(Matrix &, int, int).`
`double MatDeterminant(Matrix &, int).`
`void Copy(Matrix &, Matrix &).`

```
MatErr AddMat(Matrix &, Matrix &, int, int).
MatErr SubMat(Matrix &, Matrix &, int, int).
MatErr MulMat(Matrix &, Matrix &, Matrix &, int, int,
int, int).
```

13. **CEquation**

    This class can be used by the user to define the system of equations, which needs to be computed. An example on how to supply such equations is given Section 6.1.

    The class constructor has an integer argument which is used to determine the index of the manifold to be created.

    `CEquation (int).`

    The other member functions of this class are friend functions. They are used to evaluate the equations and their first and second derivatives at a given point. The prototypes of friend member function and their descriptions are given in the following section.

Free Functions

The functions, which will be described shortly, are free functions. Some of them are declared as friend functions in order to have access to private data members of the class in which they are declared. The other free functions, which are not friend functions, are coded in two separate header files. The free functions, which are implemented for the purpose of computation, are stored in a header file called *ComputationLib.h*. The free functions of the visualization component are stored in a header file called *graphicLib.h* .

The Detailed Descriptions of the Free Functions

`friend double` F`(Manifold & ,int, DomainPoint &)`

This function is declared as a friend free function in the class *CEquation*. It has three arguments: a reference to the computed manifold, an integer variable, and a reference to a domain Point. The integer variable determines which equation in the system is to be evaluated at the given domain point. It returns the result of evaluating the

equation(s) of the selected manifold.

`friend void` F_x`(Manifold &, int, DomainPoint &, Matrix &)`
This function is also a friend function of the *CEquation* class; it evaluates the Jacobian matrix of a given system. The function has four arguments; the second argument determines which system needs to be evaluated at the given domain point (the second argument); and, the last argument is a reference to a matrix in which the Jacobian matrix is to be stored.

`friend void` F_{xx}`(Manifold &,int, int, DomainPoint &, Matrix &)`
This is the third friend function in the class *CEquation*. It evaluates the second derivative of an equation at the given domain point(the second argument). and it stores the values in the matrix (Hessian matrix) which is passed as a reference.

```
bool getNewPoint (Manifold &, DomainPoint &,
TangentVector &, TangentVector &);
```
The argument list of this function consists of a reference to the computed part of a manifold, a reference to an object of type *DomainPoint*, which will be assigned the new computed point, and two references to the basis of a nearby Disk. The function returns `true` if a new point is found; otherwise, it returns `false`. The function starts with the first disk. If the list of the arcs in that disk is empty, it proceeds to the next disk in the linked list until it finds a disk for which the list of arcs is not empty. Then, it assigns one end of the first arc found in the list to the passed domain point object and returns `true`.

```
int solveNonLinearSys(DomainPoint *, TangentVector *,
TangentVector*, int, int);
```
This function solves a given nonlinear system. The first argument in the list is a pointer to an object of type *DomainPoint* which is used as a starting point and as an approximation solution in later iterations. The second and the third arguments are pointers to the basis of a tangent plane, and the fourth argument determines the maximum number of iterations allowed. The last argument returns the number of iterations that the function took to solve the system. The implementation of

this function is based on (a multi-dimensional version of) Newton's method which relies on solving linear systems. Thus, this function must call the member function *Gauss(Matrix*, Matrix*, int, int)* of the class *Matrix* to solve linear systems.

`void addNewDisk(Disk *, Manifold *);`

This function adds a new disk to a manifold data structure. The argument list of this function contains two pointers: a pointer to a new disk, and a pointer to the manifold being computed. The function first calls the member function *getDiskList()* of the class *Octree* to get a list of the old disks which most likely will intersect the new disk. Then, it calls the function *intersectDisks()* to check every disk in the list whether it intersects the new disk or not. If an intersection occurs between two disks, then this function calls the function *removeArcs()* to remove the interior parts. Once all disks in the list are checked, the new disk is added to the manifold data structure.

`DiskIntersection* interDisks(Disk *, Disk* );`

This function has two pointers of type *Disk* as arguments and returns a pointer to an object of type *DiskIntersection*. The intersection between any two disks is determined after solving two quartic equations. If no intersection occurs, then the function returns `NULL`; otherwise, it creates an object of type *DiskIntersection*, and returns its starting address; a pointer. The data members of the returned object are initialized using the pointers of the intersecting disks, the number of intersection points, and the angles where the intersections occur. Note that the object is allocated dynamically and, hence, it must be deleted.

`void removeArcs(DiskIntersection *);`

It has only one argument; a pointer to an object of type *DiskIntersection*. The task of this function is to remove the pieces which lie inside two intersecting disks. In order to remove these pieces this function first calls the function *splitArcs()* to split each arc according to the intersection points. Then, it removes the split parts which are interior to the union of both disks. It also updates the list of arcs in each disk. The list of arcs in both disks must include only the parts that form part of boundary of the computed manifold.

`void splitArcs(AnArc *, DiskIntersection *, AnArc *newList[]);`

This function is called by the function *removeArcs()*. It receives a pointer to an arc, a pointer to an object of type *diskIntersection*, and an array of pointers of type *AnArc*. Once an arc is split, the new pieces are stored in the array which is returned to the function *removeArcs()*.

`void computeManifoldCurvature(Disk *, TangentVector *, TangentVector *);`

This function computes the curvature of a manifold at a given point; a disk's center. The first argument is a pointer to a disk, and the other two arguments are pointers to two tangent vectors which will be assigned the manifold curvature. The basis for the tangent space as well as the point, where the curvature needs to be computed, can be obtained from the passed disk's pointer. The first step in this function is to construct the systems given in Equations 12 and 14. Then it calls the member function *Gauss(Matrix*, Matrix*, int, int)* to solve these linear systems.

5.3.3 Visualization Component

The Visualization component consists of several free functions which can be divided into two categories: functions deal with visualization properties such as setting the graphics window, a manifold colour, lighting, type of modeling view, etc. These functions are provided by the OpenGL graphics library and hence we do not give detailed descriptions of these functions here. The second category deals with rendering the resulting manifolds; it contains two functions for visualizing a manifold, and a function for clipping the plotted manifold.

1. **Visualizing**

   Computed manifolds can be visualized in two different ways: Wire-frame objects or Solid objects. Wire-frame manifolds are displayed using disks, while Solid objects are constructed using solid polygons.

   `void drawWireFrameManifold(Manifold *);`

   This function is a simple function. It has only one argument which is a pointer

to a manifold, and it returns nothing. Inside the function is a loop which goes over all the disks that form the resulting manifold. At the beginning, the function calls the member function *getFirstDisk()* of the class *manifold* to get the first disk, then it calls the function *drawArc()* to draw that disk. Once the first disk is plotted, a loop calls the member function *getNextDisk()* of the class *Disk* to get the successor disk in the linked list, and plot it. The loop continues until all disks are plotted.

`void drawSolidManifold(Manifold *);`

This function has only one argument which is a pointer to a manifold, and it returns nothing. The function must be called after all the disks, which intersect each disk, are sorted in counterclockwise order. It visualizes solid objects using solid polygons. The vertices of these polygons are in fact the center of the disks in the list. This function also computes the normal at each vertex of the polygons for lighting purposes. Once this information is available, the polygons are visualized using the OpenGL graphics function *glBegin(GL-POLYGON)*.

2. **Clipping**

`void clipping(double);`

The function has one argument: a double value which determines the size of the visualization space (a cube). The implementation of this function relies mainly on functions provided by the OpenGL graphics library, such as glEnable(GL_CLIP_PLANE) and glClipPlane(GL_CLIP_PLANE....). To use these functions for clipping an object, one must determine the coordinates and define the six faces of the visualization cube using the function glClipPlane(GL_CLIP_PLANE,...). Then, the function glEnable(GL_CLIP_PLANE) is called to clip the displayed object.

5.4 The Implementation Phase

5.4.1 Graphical User Interface (GUI)

The GUI is built using the MFC library which generates the framework of this component automatically such as header files, a skeleton code for the GUI, prototypes, and functions bodies for message handlers. The GUI, as shown in Figure 46, is divided into two parts. For the input section, the LEFT and RIGHT buttons of the mouse are used most frequently; the TAB key can also be used to move from one field to the next. On the top layout of the GUI are a number of edit controls and push buttons. Their description follows below:

1. **The Embedding Space (n)**
   The minimum value of the embedding space dimension(n) which is also given as a default value, is ($n = 1$). Thus, the user should enter the proper dimension if it is greater than the given default value.

2. **The Cube Size**
   The cube size determines the boundary of the space in which a manifold is to be computed and visualized. Note that the cube is centered at the origin and the default size is 2.

3. **Manifold No.**
   Here, the user can select one of the manifolds, which is expected to be defined inside the class *CEquation*.

4. **Iso-Value**
   This option allows the user to enter different iso-values.

5. **Type of Visualization**
   There are two buttons grouped under the type of visualization box: one button is labeled as *Solid Object* and the other as *Wireframe Object*. The user is allowed to select only one of the two buttons at a time; however, the Solid Surface button is selected by default.

6. **Starting Point.**
   There are two buttons grouped under the Starting Point box: one button is

labeled as *Random* and the other as *Specified.* The user is allowed to select only one of the two buttons at a time. The Random button is selected by default. If the Random button is selected, then the program randomly selects the starting point inside the given cube. However, if the other button is selected, then the user is required to enter the starting point.

7. **Computation Button**
   The user can start the computation by pressing the computation button which is located above the graphics window.

8. **Exit Button**
   This button can be used to exit from the program completely.

There are other keys, which can be used for the graphics window once the computed manifold is displayed. In order to use the keys indicated below, the user must activate the graphics window using the mouse.

1. *Up/Down Keys*
   The keys can be used to zoom in and out.

2. *Left/Right Keys*
   The keys can be used to rotate the displayed object.

3. *The keys I and i*
   Either key can be used to zoom in.

4. *The keys O and o*
   Either key can be used to zoom out.

5. *The keys S and s*
   Either key can be used to start/stop the rotation of the displayed object.

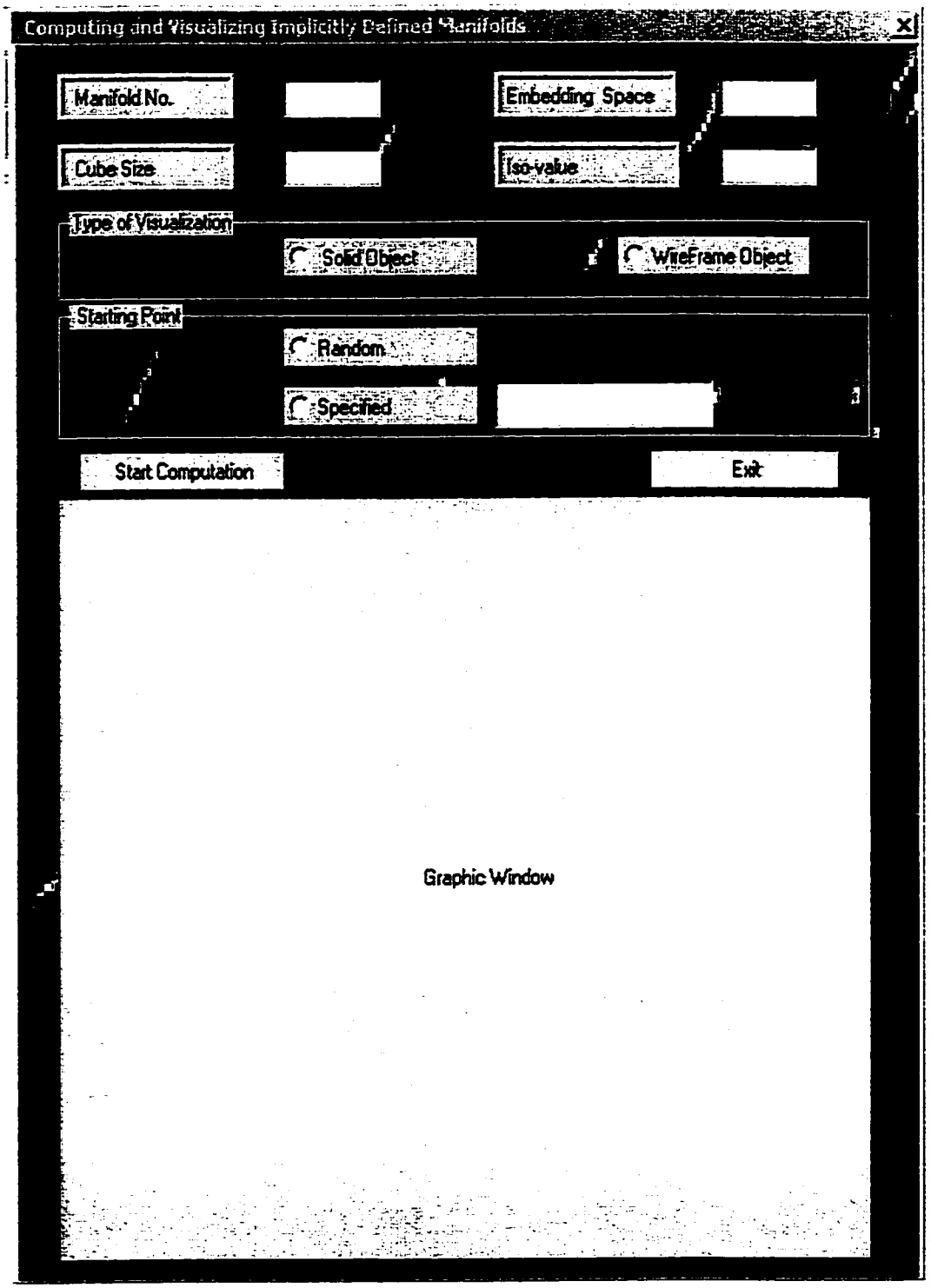


Figure 46: The Graphical User Interface (GUI)

5.4.2 Class Declarations

1. The RangePoint Class Declaration

```
class RangePoint
{
  public:      // Member Functions
   RangePoint(int = RANDIM);
   RangePoint(double *, int= RANDIM);
   RangePoint(double,int = RANDIM);
   RangePoint(RangePoint &);
   ~RangePoint();
   double & operator = (const RangePoint &);
   double & operator [](const int);
  private:     // Data Members
   double *coordinates;
   int size;
};
```

2. The DomainPoint Class Declaration

```
class DomainPoint
{
  public:      // Member Functions
   DomainPoint(int = DOMDIM);
   DomainPoint(DomainPoint &);
   ~DomainPoint();
   double & operator [] (int);
   DomainPoint & operator = (const DomainPoint &);
   bool operator == (const DomainPoint &);
   bool operator != (const DomainPoint &);
   DomainPoint & operator = (DomainPoint &);
   void setX(int,const double);
   double getX(int);
```

```
    int getSize();
  private:     // Data Members
    double *coordinates;
    int size;
};
```

3. The TangentVector Class Declaration

```
class TangentVector
{
  public:      // Member Functions
    TangentVector(int = DOMDIM);
    TangentVector(double *,int = DOMDIM);
    TangentVector(TangentVector &);
    ~TangentVector();
    TangentVector & operator = (TangentVector &);
    TangentVector crossPro(TangentVector &);
    TangentVector vector(DomainPoint &, DomainPoint &);
    void crossPro(TangentVector &,float *);
    void setX(int, const double);
    vodi unitVector();
    double getX(int index);
    double dotPro(TangentVector &);
    double vectorLength();
    double vectorNorm();
    double operator [](int index);
  private:     // Data Members
    double *vData; };
```

4. The TangentPlanePoint Class Declaration

```
class TangentPlanePoint
{
```

```
  public:      // Member Functions
   TangentPlanePoint(double = 0.0, double = 0.0, Disk* = NULL);
   double getX1();
   double getX2();
   void UpDateTPP(double, double, Disk *);
   void getTangentPoint(double &, double &);
   Disk* getTangentDiskPtr();
  private:     // Data Members
   double x1;
   double x2;
   Disk* disk;
};
```

5. The Node Class Declaration

```
template <class T>
class Node
{
  public:      // Member Functions
   Node<T>(T* = NULL, Node<T>* = NULL);
   ~Node<T>();
   Node<T>* getNext();
   T* getValue();
   void setValue(T*);
  private:     // Data Members
   T *dataPtr;
   Node<T> *next;
};
```

6. The LList Class Declaration

```
template <class T >
class LList
{
  public:      // Member Functions
   LList();
   ~LList();
   LList(LList<T> &);
   LList<T>& operator = (LList<T> &);
   void removeFrontNode();
   void removeRearNode();
   void removeNode(Node<T> *);
   void addNewNode(T *);
   bool exist(T *);
   bool empty();
   int size();
   Node<T> *getFront();
   Node<T> *getRear();
   Node<T> *getNode(T *);
   deallocateList(LList<T> *);
  private:
   // Member functions
   Node<T> *getPreviousNode(T *);
   // Data Members
   Node<T> front;
   Node<T> rear;
   int listSize;
};
```

7. The Disk Class Declaration

```
class Disk
{
  public:      // Member Functions
   Disk(DomainPoint &, TangentVector &, TangentVector & );
   void addArcToList(AnArc *);
   void addDiskToList(Disk *);
   void linkDisks(Disk *, Disk *);
   void updateArc(AnArc *,Node<AnArc> *);
   void sortDisk();
   void setAxes(double, double);
   Disk* getPreviousDisk();
   Disk* getNextDisk();
   DomainPoint getCenter();
   TangentVector getBase1();
   TangentVector getBase2();
   double getAxe1();
   double getAxe2();
  private:
  // Member Functions
   void setNextDisk(Disk *dPtr);
   void setPreviousDisk(Disk *dPtr);
   TangentPlanePoint* projectPoint(DomainPoint *);
   void swapListElements(int,int);
  // Data Members
   DomainPoint center;
   TangentVector base1, base2;
   double axe1, axe2;
   Disk *nextDisk, *previousDisk;
   LList<AnArc> *arcsList;
   LList<Disk> *disksList;
};
```

8. An AnArc Class Declaration

```
class AnArc
{
  public:      // Member Functions
   AnArc(Disk *, double, double);
   void linkArcs(AnArc *, AnArc *);
   double getTheta1();
   double getTheta2();
   AnArc* getNextArc();
   AnArc* getPreviousArc();
   Disk* getDisk();
  private:
  // Member functions
   void setNextArc(AnArc *arcPtr);
   void setpreviousArc(AnArc *arcPtr);
  // data members
   Disk *disk;
   double theta1, theta2;
   AnArc *nextArc, *previousArc;
};
```

9. The Manifold Class Declaration

```
class Manifold
{
  public:      // Member Functions
   Manifold(Disk *, Anarc *);
   Disk* getFirstDisk();
   Disk* getLastDisk();
   AnArc* getFirstArc();
   AnArc* getLastArc();
   void addDisk(Disk *);
```

```
    void sortDisks();
  private:
  // Member Functions
    void addArcList(Hdisk *);
  // Data Members
    Disk* firstDisk;
    AnArc* firstArc;
};
```

10. The Octree Class Declaration

```
class Octree
{
  public:                                  // Member Functions
    Octree(double = 1.0);
    Octree(Disk *, double);
    Octree* getNode(int);
    void getDiskList(Octree *, Octree *, LList<Disk> *);
    bool insert(Octree*, Octree *);
    void found(Octree*, LList<Disk> *);
    DomainPoint getCenter();
    void setDisk(Disk *);
    void setNode(int, Octree*);
  private:
  // Member Functions
    void upDateBoundary(Octree*, int);
    bool insideCube(Octree*);
    bool cubeOverLap(double *);
    int findPath(Octree*, Octree *);
  // Data Members
    double L,R,U,D,F,B;
    Octree *node[8];
    Disk *diskPtr;
};
```

11. The Matrix Class Declaration

```
class Matrix
{
  public:      // Member Functions
   Matrix(int = 10, int = 10);
   ~Matrix();
   void reSize(int, int);
   int getRows();
   int getColumns();
   double&operator()(int, int);
   bool chackRowCol(int, int);
  private:     // Data Members
   int maxRows;
   int maxColumns;
   double *mData;
};
```

Chapter 6

Running and Testing the CVIDM

This chapter consists of two sections. The steps that are required to enter a system of equation(s) of a manifold are given in the first section. The second section provides several examples of computing manifolds, with other information, such as CPU time and number of disks needed to construct these manifolds.

6.1 Running the Software

The following example illustrates how to enter a system of equations and its first and second derivatives. The manifold to be computed in this example is a sphere which is implicitly defined by

$$F(X) = X_1^2 + X_2^2 + X_3^2 - R^2 = 0. \tag{24}$$

The radius of the sphere is R, the size of the embedding space in this example is $(n = 1)$, and the size of the domain space is $n + 2 = 3$. The first derivative, F' or F_x, is a Jacobian matrix of size 1×3 given by

$$F' = F_x = \begin{pmatrix} 2X_1 & 2X_2 & 2X_3 \end{pmatrix}.$$

The second derivative, F'' or F_{xx} , is a Hessian matrix of size 3×3 with components given by

$$F'' = F_{xx} = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{pmatrix}.$$

1. **Labeling the Equations**

   The user can define more than one system of equations in the class *CEquation*. Each system of equations must be labeled using indices (positive integers). The user can choose one of the labeled systems by entering its index via the GUI.

2. **Supplying the Equations**

   The class *CEquations* has three friend free functions:

   - F(Manifold &, int , DomainPoint & X),
   - F_x(Manifold &,int , DomainPoint & X, Matrix & J),
   - F_{xx}(Manifold &,int , DomainPoint & X, Matrix & H).

   The above three functions evaluate the function(s) that define a manifold, and their Jacobian matrix and the Hessian matrix, respectively. The first parameter in the argument list of the three functions is a reference to the manifold being computed. The second parameter is an integer variable which is used to determine which system of equations is to be evaluated. The third parameter in the three function is a reference to a domain point used in the evaluation. The coordinates of the domain point can be accessed using the ordinary index "[ ]" operator. For example, X[0] refers to the first coordinate, X[1] refers to the second coordinate, and so on. Thus, to define the equation of a sphere, the user must add to the function, `double F(···)`, the following statements:

   ```
   case 1:
   F[0] = pow(X[0], 2) + pow(X[1], 2) + pow(X[2], 2) − pow(R, 2);.
   ```

   The user must also add to the function, $F_x(\cdots)$, the following statements:

   ```
   case 1:
   J(0, 0) = 2.0 * X[0];
   J(0, 1) = 2.0 * X[1];
   J(0, 2) = 2.0 * X[2];
   ```

   in order to evaluate the Jacobian matrix (J) at a point X.

   The third function, $F_{xx}(\cdots)$, evaluates the Hessian matrix (H) at a point X. To get the total second derivatives of a given system, this function needs to be called n times where n is the number of the equations in the system. At each call, the function returns the Hessian matrix of one the equations in the system. The second derivative of the sphere's equation is coded as follows:

```
case 1:
H(0,0) = 2.0;
H(0,1) = H(1,0) = 0.0;
H(0,2) = H(2,0) = 0.0;
H(1,1) = 2.0;
H(1,2) = H(2,1) = 0.0;
H(2,2) = 2.0;
```

After the equations of the desired manifold are coded as explained above, the user can compile and execute the program. Once the program executes, the GUI will appear for the user to enter the required data: the manifold to be computed, the computation space, the Iso-value, type of visualization, and so on.

6.2 Testing the Software

CVIDM has been tested on a PC computer with the following specifications:

1. **Hardware**

Processor:	Intel Pentium III.
CPU Speed:	800 Mhz.
RAM:	128 MB.

2. **Software**

Platform:	Windows NT 4.0.

The following sections provide examples of different manifolds that have been computed and visualized using CVIDM. The constant values used in the examples, unless explicitly stated otherwise, are:

- An epsilon (ϵ) = 0.1.
- The maxAxesLength = 0.1.
- Cube of size $(2 \times 2 \times 2)$.

6.2.1 Compact Manifolds

Compact manifolds are more difficult to compute than unbounded manifolds because they are finite and have no edges. When computing a compact manifold, one wants to avoid computing it multiple times. This is achieved by updating and maintaining the boundary of the computed manifold whenever a new part of the manifold is explored. The boundary is updated by deleting the arcs which lie inside the computed part. Eventually, there will be no arcs left, and the program terminates. The stages of constructing such manifolds are shown in the first two examples (a sphere and a torus).

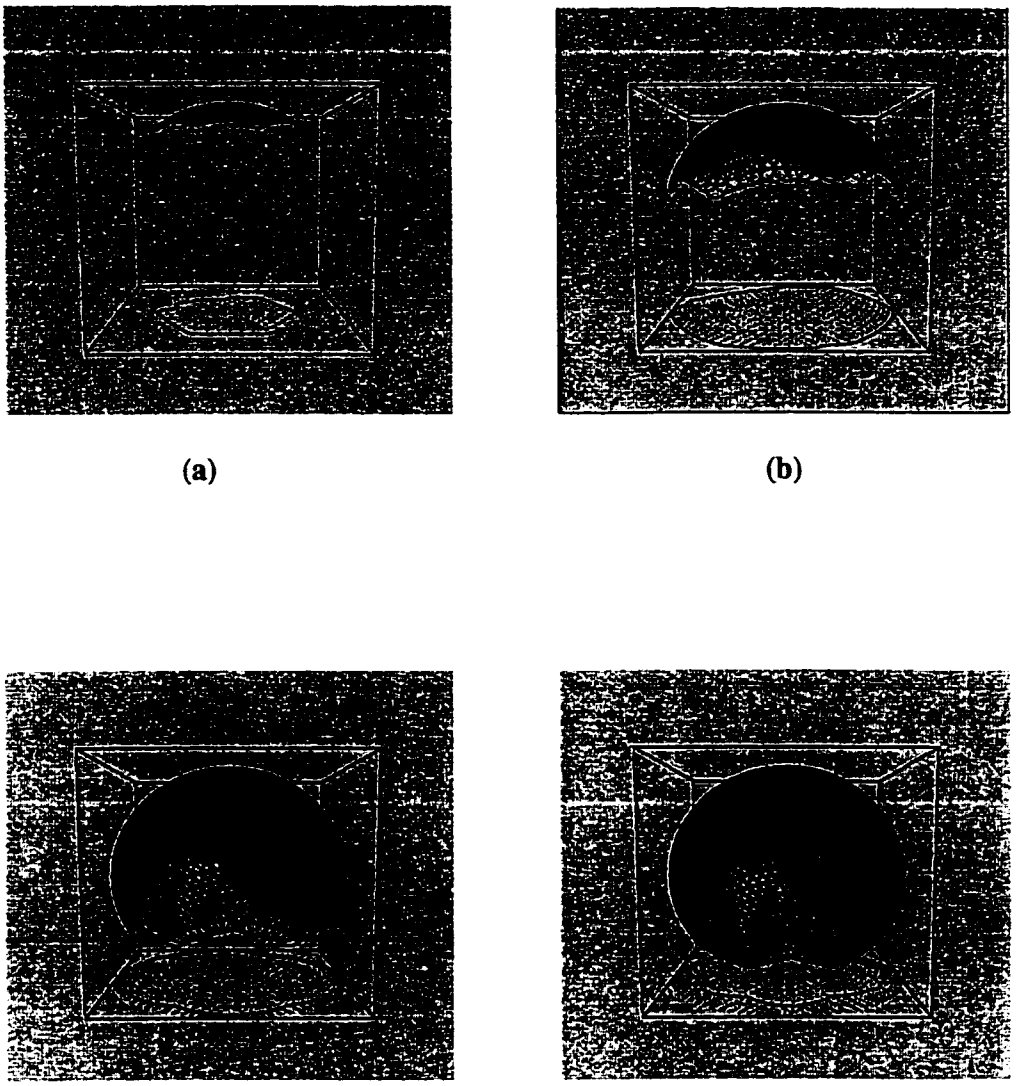
1. **Sphere**

   This example illustrates how a compact manifold such as a sphere is constructed.

The pictures given in Figure 47 show how the surface of the sphere is built using wireframe disks. The boundary of the computed part at different stages and the projection of the centers of the disks are also shown in these pictures. Pictures (a) and (b) in Figure 47 show that the boundary of the computed part is getting larger and larger, and it attains its maximum when half of the sphere is computed. Then it starts to shrink as part of the second half of the sphere is explored as shown in pictures (c) and (d). Figure 48 shows parts of the surface constructed using solid polygons. The solid polygons, which are used to visualize manifolds, are constructed based on the triangulation of the surface as shown in Figure 49. The complete surface of the sphere is shown in Figure 50 which illustrates that there is no boundary left and hence the computation of the sphere terminates. Figure 51 shows the complete surface of the sphere constructed using solid polygons. Data on the computation and visualization are given in Table 1.

Total computation time	**2.266** sec.
Selecting new Points	0.064 sec.
Computing the basis	0.015 sec.
Computing manifold curvature	0.048 sec.
Solving linear systems	0.798 sec.
Computing disk intersections	0.326 sec.
Splitting and Removing Interior Pieces	0.359 sec.
Visualization time	0.047 sec.
Number of disks used	**1122**

Table 1: Timing and Memory Usage for Constructing a Unit Sphere



(a) (b)

(c) (d)

Figure 47: Constructing a Sphere Using Disks

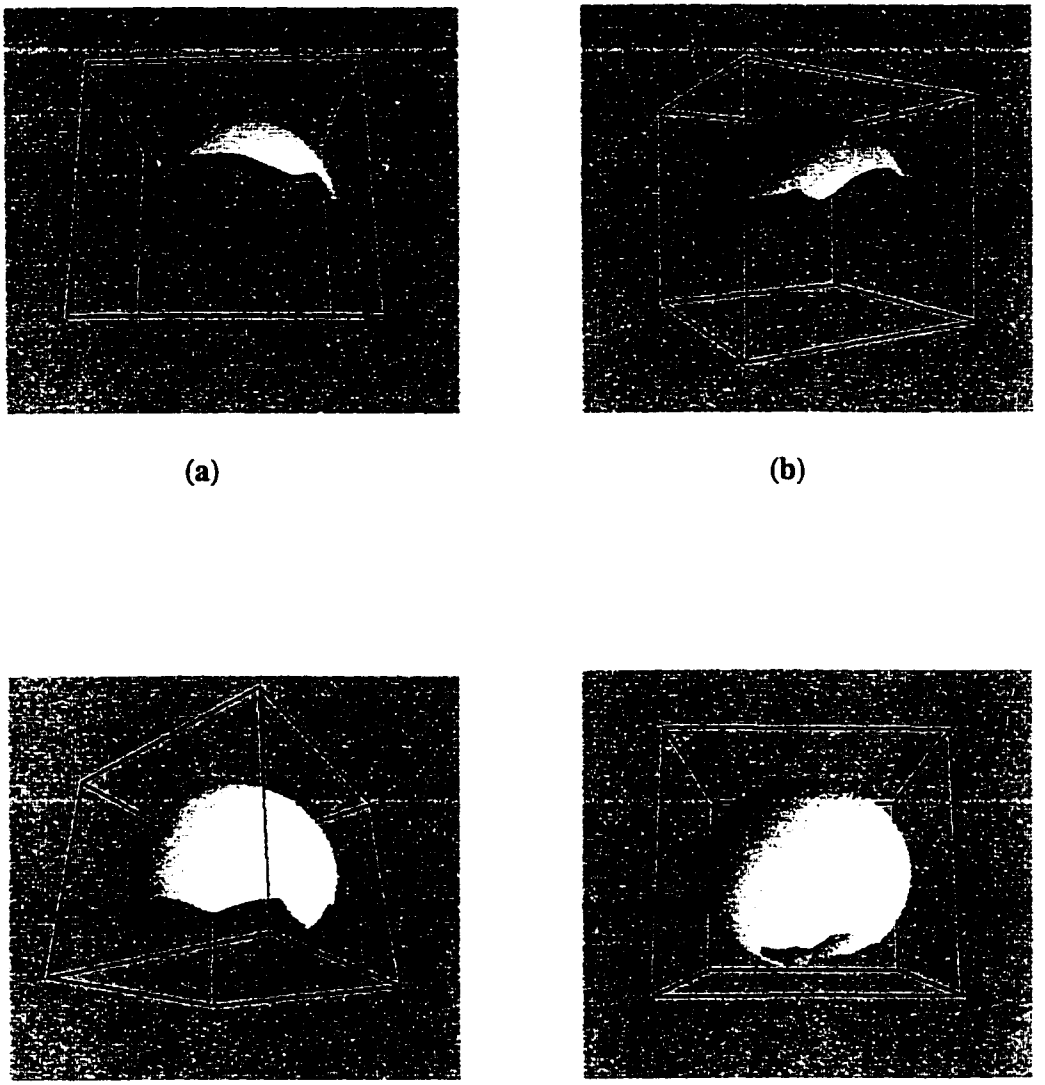
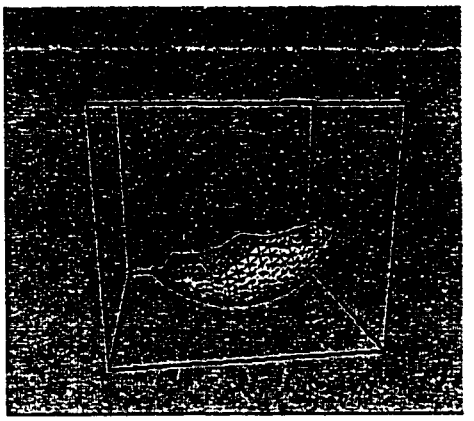
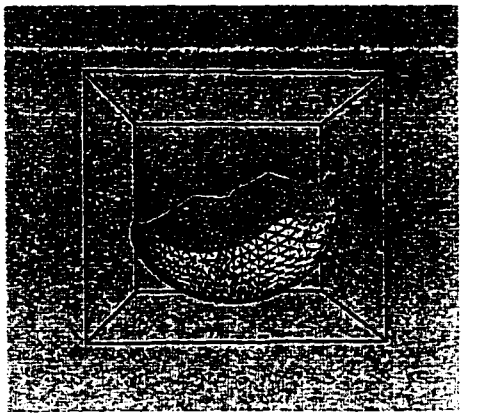


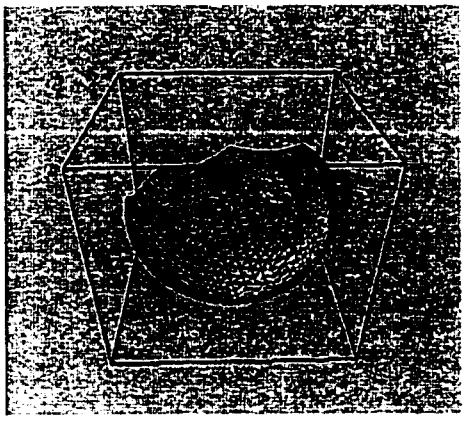
Figure 48: Constructing a Sphere Using Solid Polygons



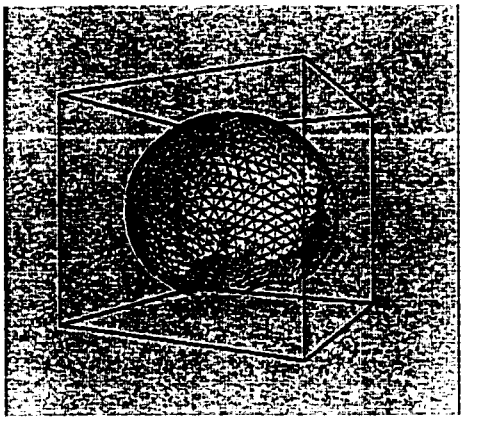
(a)



(b)



(c)



(d)

Figure 49: Triangulation of a Sphere at Different Stages

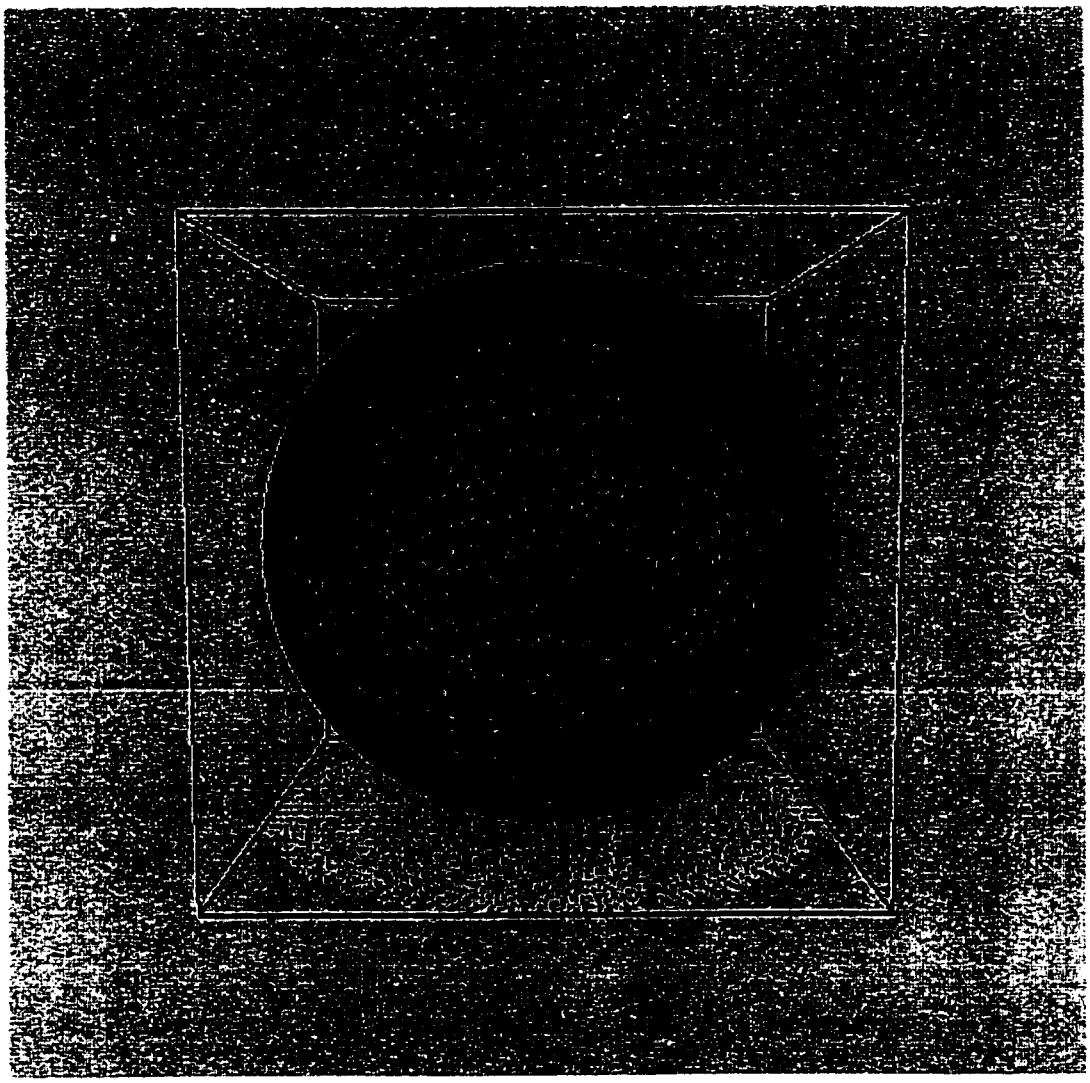


Figure 50: Wire-frame Sphere Built Using Disks

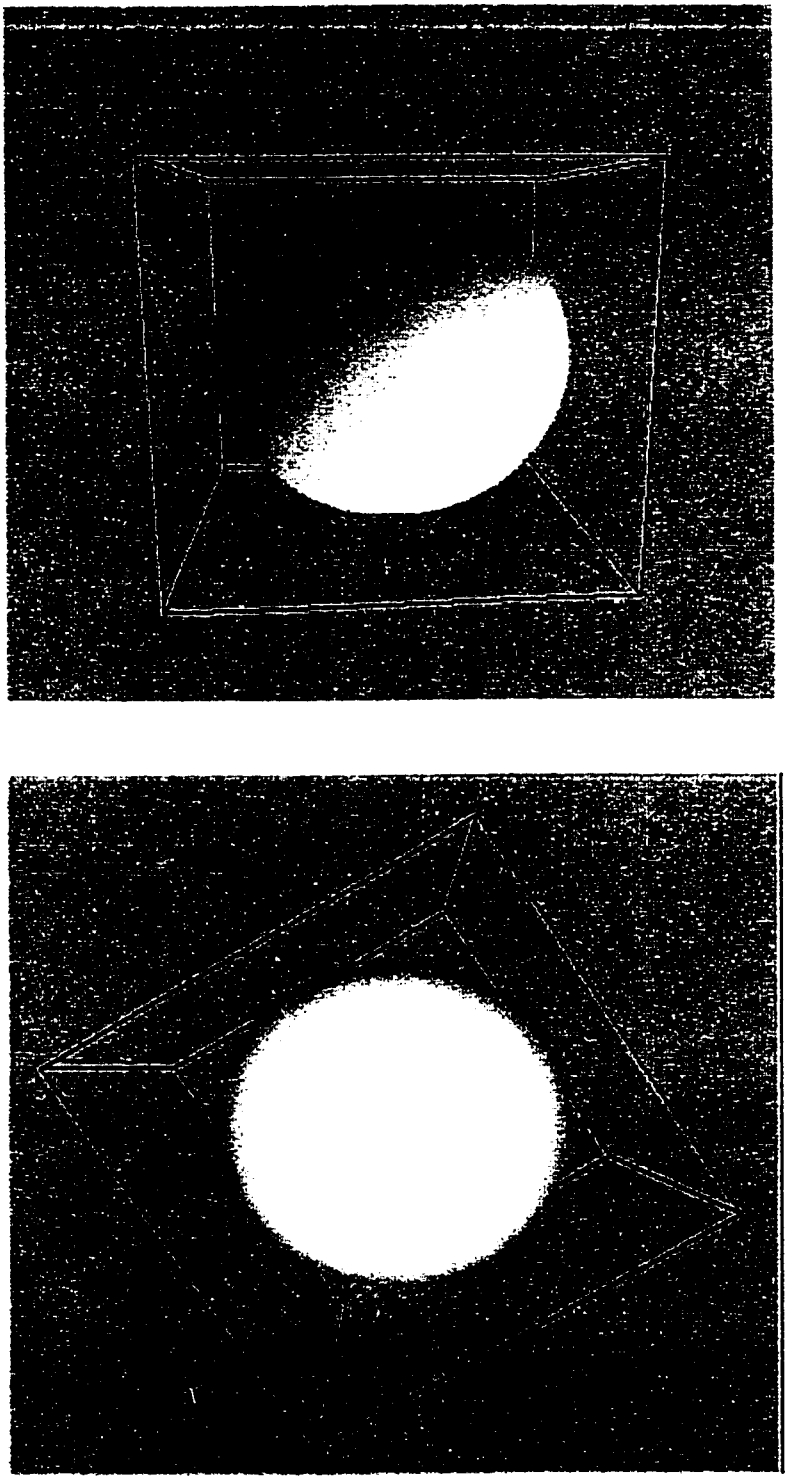


Figure 51: Solid Sphere Using Solid Polygons

2. **Torus**

The computation of a torus is similar to the computation of a sphere except that the torus has an opening in the middle which makes the computation more difficult and, more time and disks are needed to construct it. The implicit equation of a torus is

$$F(X) = 4.0\ R_1^2\ (X_2^2 + X_3^2) - (X_1^2 + X_2^2 + X_3^2 + R_1^2 - R_2^2)^2\ .$$

In this example, $R_1 = 0.75$, and $R_2 = 0.2$. Figure 52 shows how the torus is constructed. It also shows the boundary of the computed part of the manifold at different stages. Figure 53 shows solid parts of the surface at different stages. The complete surface of the torus using wire-frame disks is shown in Figure 54. and the complete solid surface of the torus is given in Figure 55; see Table 2 for timing and memory usage for computing and visualizing this example.

Total computation time	**2.828** sec.
Selecting new Points	0.061 sec.
Computing the basis	0.015 sec.
Computing manifold curvature	0.064 sec.
Solving linear systems	0.923 sec.
Computing disk intersections	0.546 sec.
Splitting and Removing Interior Pieces	0.425 sec.
Visualization time	0.063sec.
Number of disks used	**1215**

Table 2: Timing and Memory Usage for Constructing a Torus

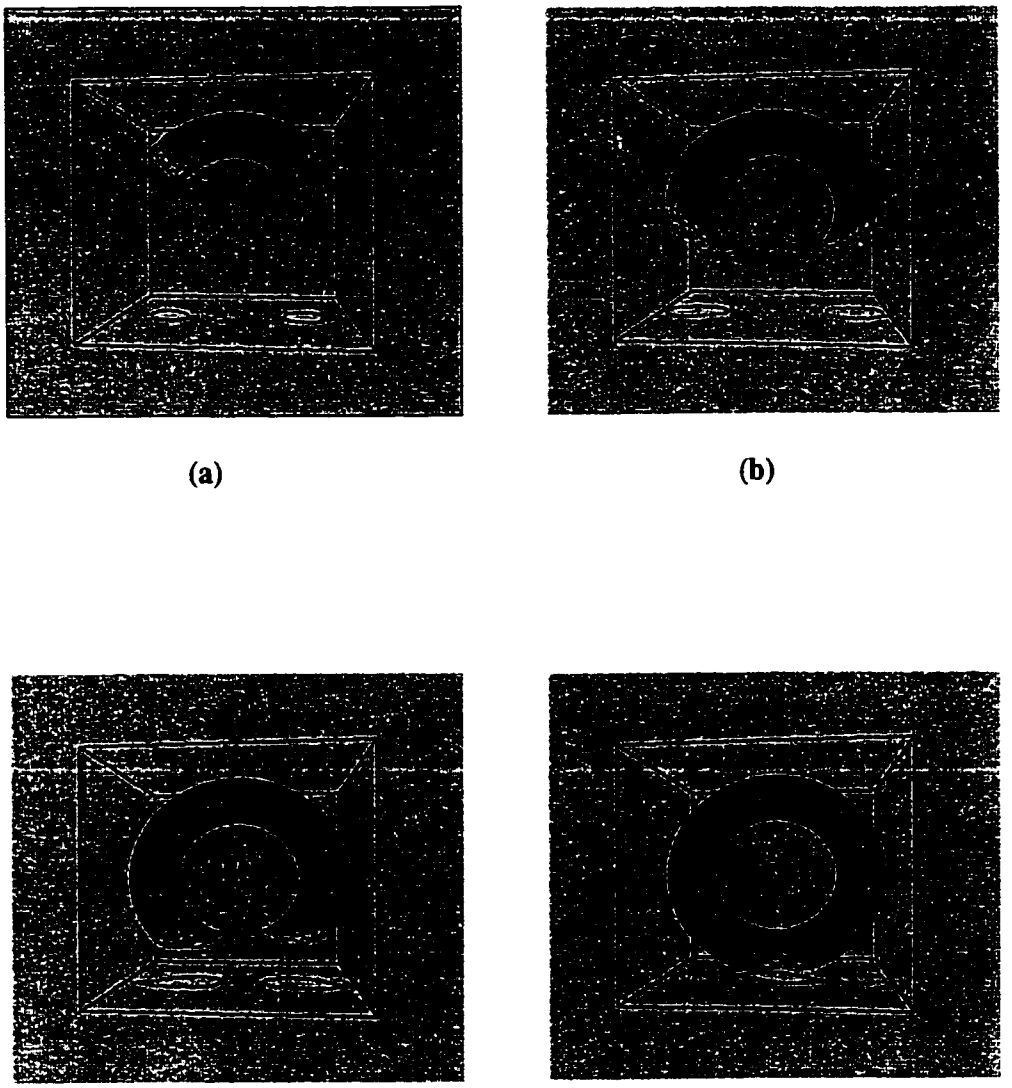


Figure 52: Constructing a Torus Using Disks

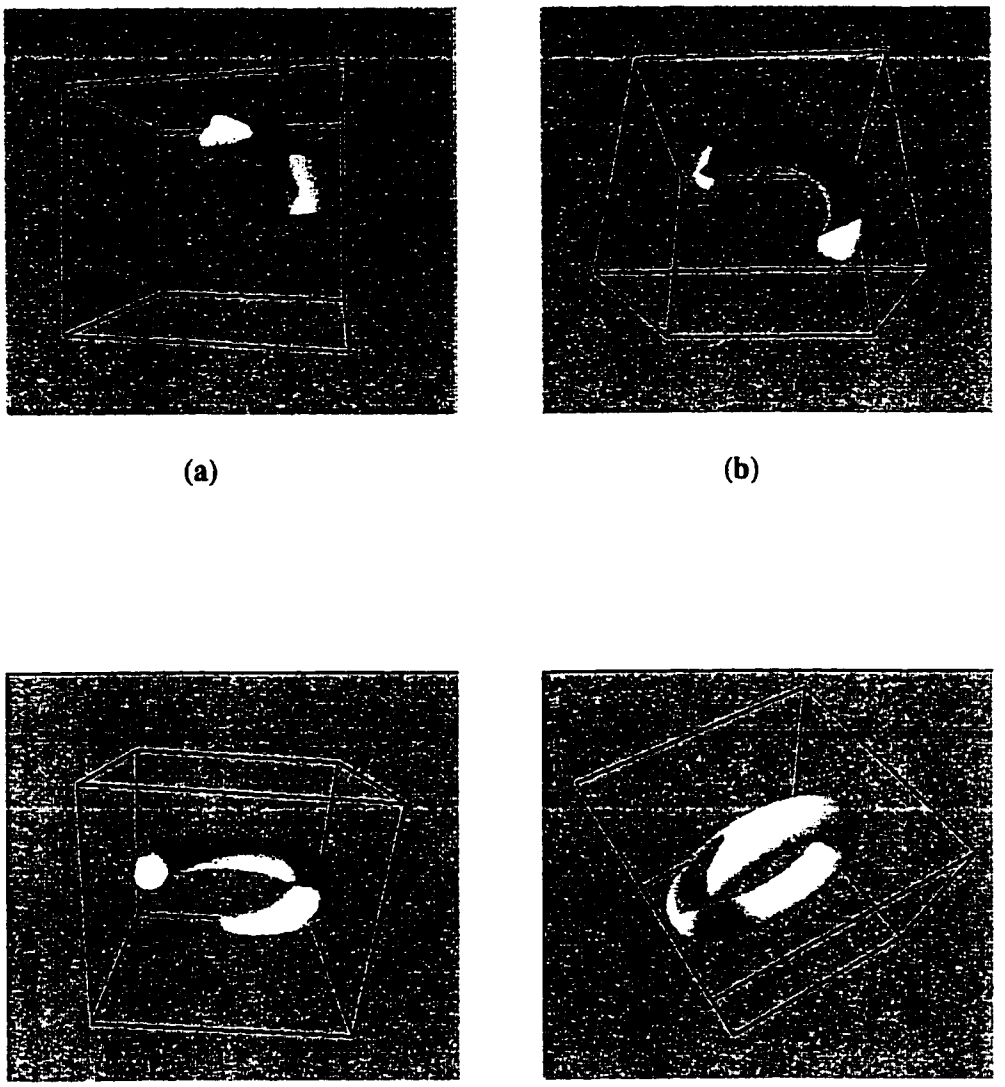


Figure 53: Constructing a Torus Using Solid Polygons

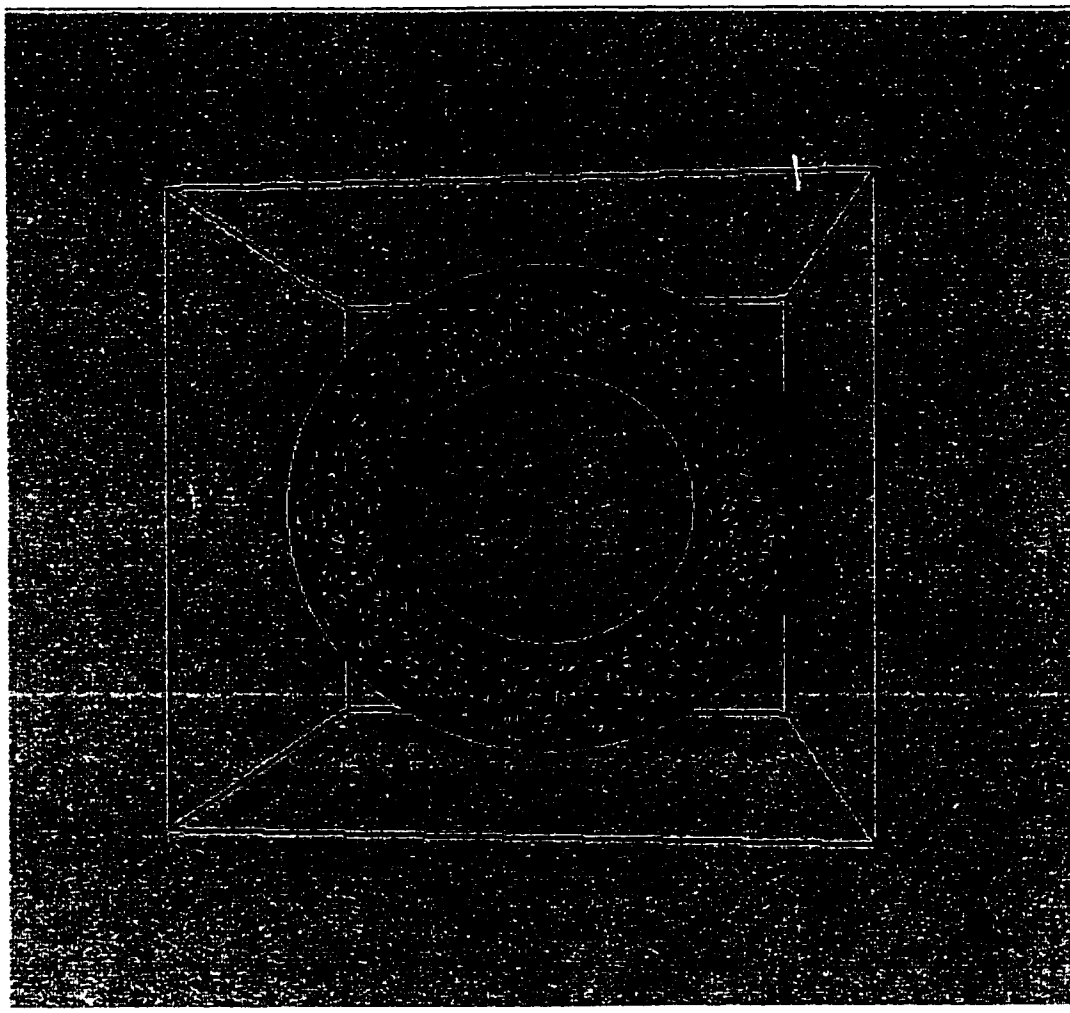


Figure 54: Wire-frame Surface of a Torus

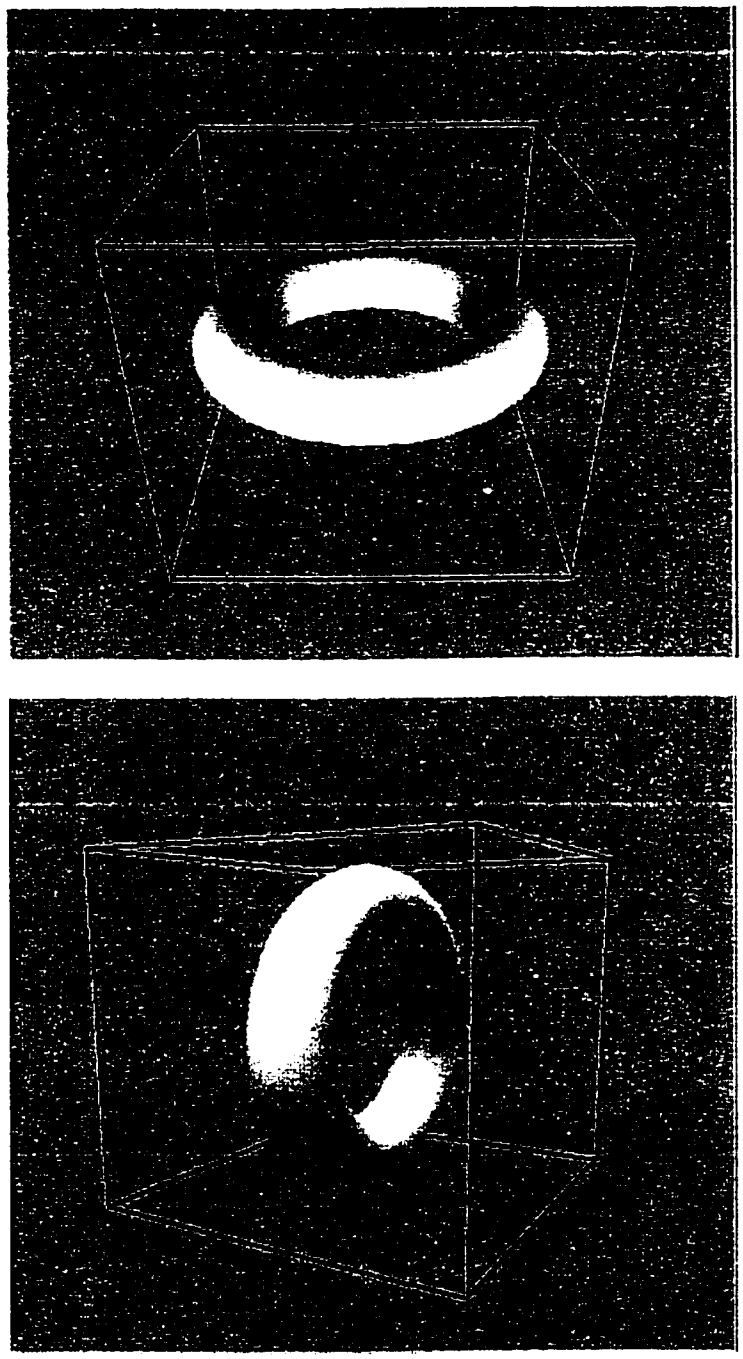


Figure 55: Solid Torus

6.2.2 Unbounded Manifolds

For unbounded manifolds, the algorithm discards any part of the boundary that lies outside the computation space (the given cube), and it terminates if no more points are found inside the cube.

- **Example 1**

  The manifold defined by the equation

$$F(X) = X_1^2 * X_2^2 + X_1^2 * X_3^2 + X_2^2 * X_3^2 - Iso - value = 0.$$

  is shown in Figure 56 for different values of *Iso-value*. Note that as the *Iso-value* gets smaller, the six openings get smaller too. See Table 3 for the computation and visualization performance.

Total computation time	**2.031** sec.
Selecting new Points	0.031 sec.
Computing the basis	0.046 sec.
Computing manifold curvature	0.002 sec.
Solving linear systems	1.043 sec.
Computing disk intersections	0.22 sec.
Splitting and Removing Interior Pieces	0.219 sec.
Visualization time	0.046sec.
Number of disks used	**841**

Table 3: Timing and Memory Usage for Constructing Example 1

- **Example 2**

  The function defining this example is given by

$$F(X) = X_1^2 + X_2^3 + X_3^3 - X_2^2 - X_3^2 - Iso - value = 0.$$

  This manifold has three openings as shown in the pictures in Figure 57. The shape of these openings and the smoothness of the manifold, especially near the openings, show how accurate the CVIDM is in computing and visualizing such complex manifolds.

Total computation time	**2.313** sec.
Selecting new Points	0.078 sec.
Computing the basis	0.061 sec.
Computing manifold curvature	0.031 sec.
Solving linear systems	0.55 sec.
Computing disk intersections	0.419 sec.
Splitting and Removing Interior Pieces	0.298 sec.
Visualization time	0.046sec.
Number of disks used	**1115**

Table 4: Timing and Memory Usage for Constructing Example 2

- **Example 3**

  The implicit function defining the manifold shown in Figure 58 is

$$F(X) = (X_1 + X_2)(X_1X_2 - X_3^2) + X_1X_2X_3 - Iso - value = 0.$$

  The pictures in (a) and (b) show the complete manifold using different *Iso-values*. Notice here that the opening in the manifold gets smaller as the Iso-value gets smaller. However, when the absolute value of the *Iso-value* is very small, the manifold splits into two parts, and only one part is computed. The reason for this is that the opening no longer exists, and the manifold has split into two parts. As a result, the list of the arcs, which form the boundary of the computed manifold, will become empty, and the computation is then terminated. For example, if the $Iso - value = 0.000125$, and we start the computation of the manifold from the point; the initial point, (0.2919,0.0820,0.1911), we get the "upper" part of the manifold; see Figure 58 (c). Whereas, if we start from the point (-0.0218,0.0214,0.5), we get the "lower" part of the manifold: see Figure 58 (d). The timing and memory usage for computing and the visualizing the manifold shown in picture (a) is given in Table 5.

- **Example 4**

  The function defining the manifold in this example is

$$F(X) = X_1^3 - 3X_1X_2^2 + X_3^2 - X_1^2 - X_2^2 - Iso - value = 0.$$

Total computation time	**1.890** sec.
Selecting new Points	0.015 sec.
Computing the basis	0.016 sec.
Computing manifold curvature	0.062 sec.
Solving linear systems	0.658 sec.
Computing disk intersections	0.248 sec.
Splitting and Removing Interior Pieces	0.28 sec.
Visualization time	0.047sec.
Number of disks used	**922**

Table 5: Timing and Memory Usage for Constructing Example 3

Total computation time	**2.609** sec.
Selecting new Points	0.016 sec.
Computing the basis	0.005 sec.
Computing manifold curvature	0.015 sec.
Solving linear systems	0.878 sec.
Computing disk intersections	0.403 sec.
Splitting and Removing Interior Pieces	0.294 sec.
Visualization time	0.062sec.
Number of disks used	**1282**

Table 6: Timing and Memory Usage of Example 4 (Cube of Size = 2.0)

The computed manifold is shown in Figure 59. The timing table is shown in Table 6.

The pictures in Figure 60 show the same manifold given in Figure 59. but this time the computation space is different. The Cube size used in Figure 59 was $(2 \times 2 \times 2)$. but the Cube size in Figure 60 is $(4 \times 4 \times 4)$. Table 7 shows the timing and memory usage for computing and visualizing the manifold in the bigger Cube. Comparing Table 6 and Table 7, we see that more time is needed for computing and visualizing the manifold as the size of the region increases.

- **Example 5**

  The manifold defined by the equation

$$F(X) = X_1^2 + X_2^2 + X_3^2 - (-X_1^3 - X_2^2 + X_3^3)^2 - Iso - value \ ,$$

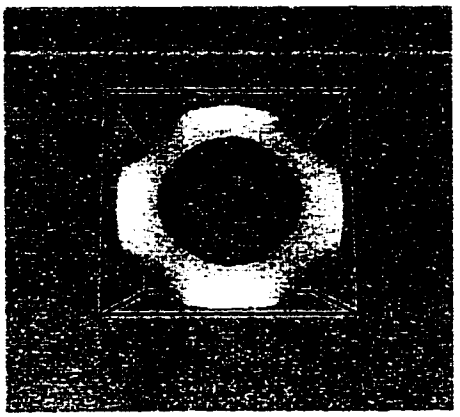
Total computation time	**13.203** sec.
Selecting new Points	2.353 sec.
Computing the basis	0.095 sec.
Computing manifold curvature	0.109 sec.
Solving linear systems	4.332 sec.
Computing disk intersections	1.686 sec.
Splitting and Removing Interior Pieces	1.383 sec.
Visualization time	0.25sec.
Number of disks used	**5253**

Table 7: Timing and Memory Usage of Example 4 (Cube of Size = 4.0)

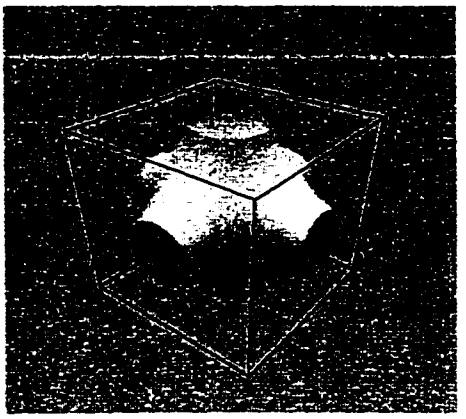
where $F : R^3 \rightarrow R$, is shown in Figure 61. Table 8 shows the number of disks and the time needed for computing and visualizing this manifold. In fact, this manifold can also be computed using a system of two equations as shown in the following section.

Total computation time	**3.437** sec.
Selecting new Points	0.047 sec.
Computing the basis	0.06 sec.
Computing manifold curvature	0.062 sec.
Solving linear systems	1.736 sec.
Computing disk intersections	0.344 sec.
Splitting and Removing Interior Pieces	0.345 sec.
Visualization time	0.063 sec.
Number of disks used	**1214**

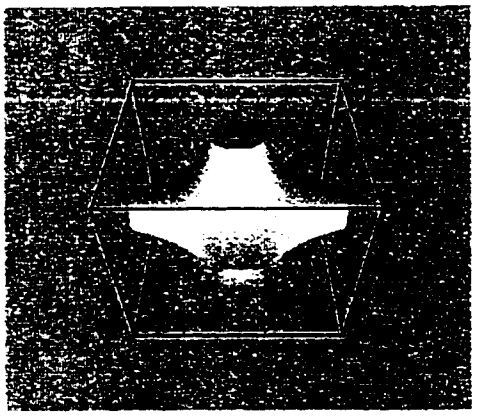
Table 8: Timing and Memory Usage for Constructing Example 5 in R^3



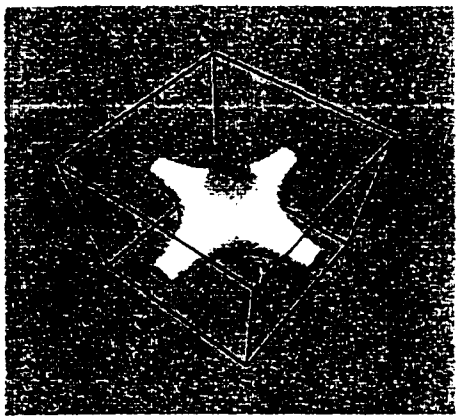
(a) Iso_value = 0.2



(b) Iso_value = 0.08

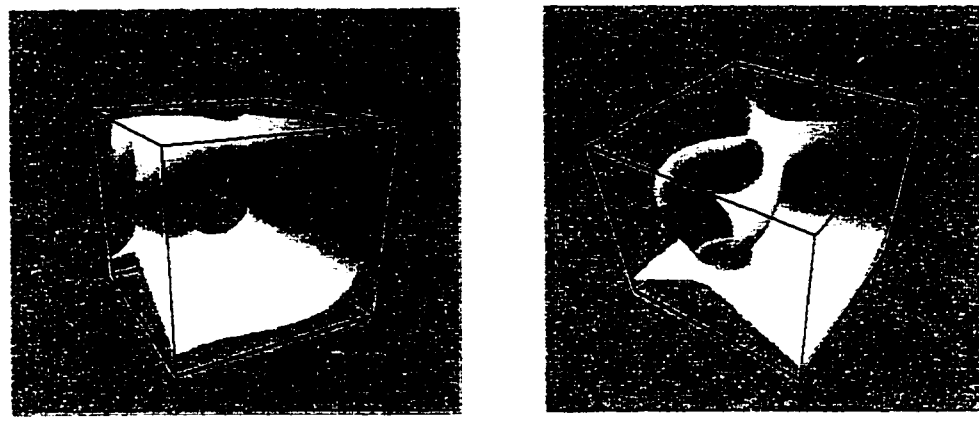


(c) Iso-value =0.0035

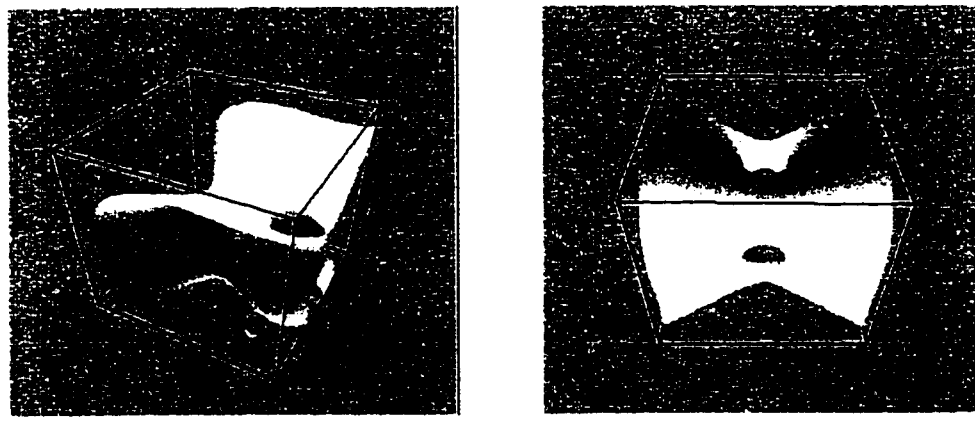


(d) Iso-value = 0.001

Figure 56: Example 1 Using Different Iso-values

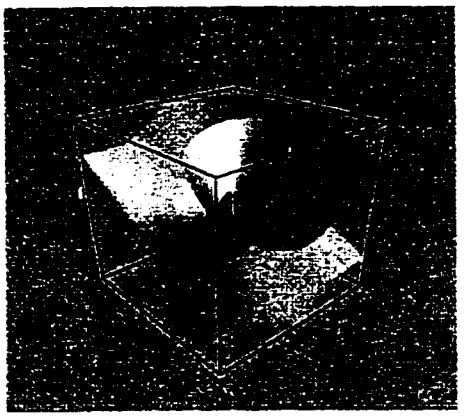


(a) (b)

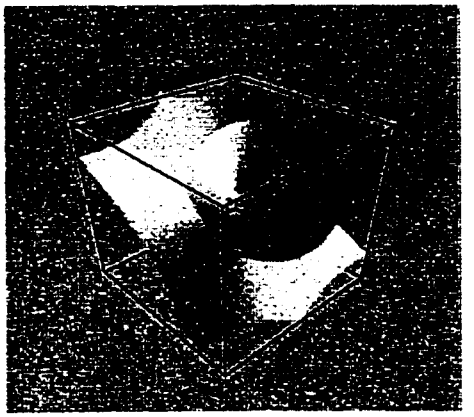


(c) (d)

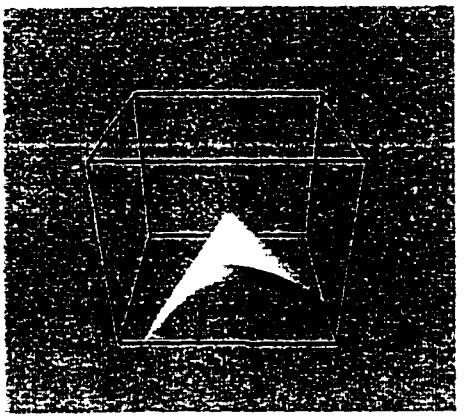
Figure 57: Different Views for Example 2 (Iso-value = 0.1)



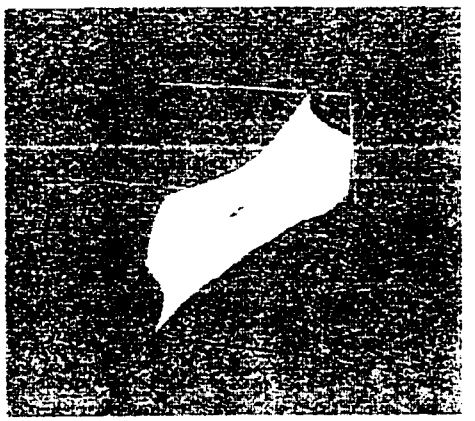
(a) Iso-value = -0.0009



(b) Iso-value = -0.0005

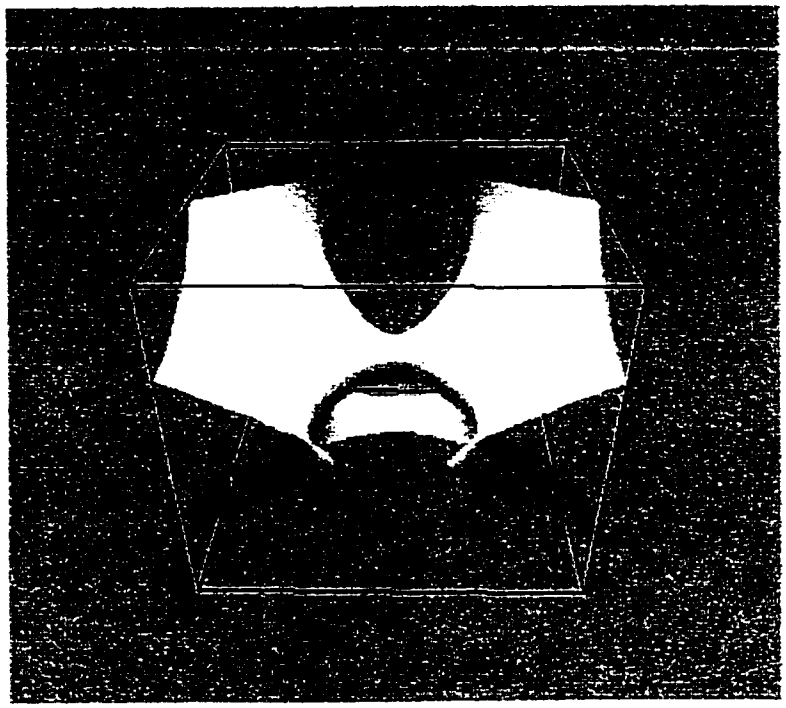


(c) Iso-value = -0.000125

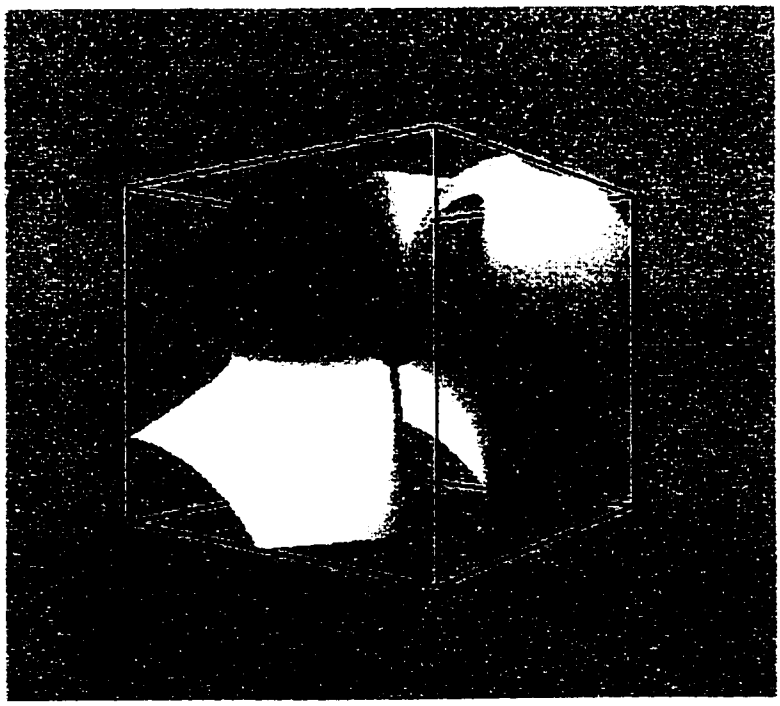


(d) Iso-value = -0.000125

Figure 58: Example 3 Using Different Iso-values



(a)



(b)

Figure 59: Example 4 (Iso-value = -0.01 and Cube Size = 2.0)

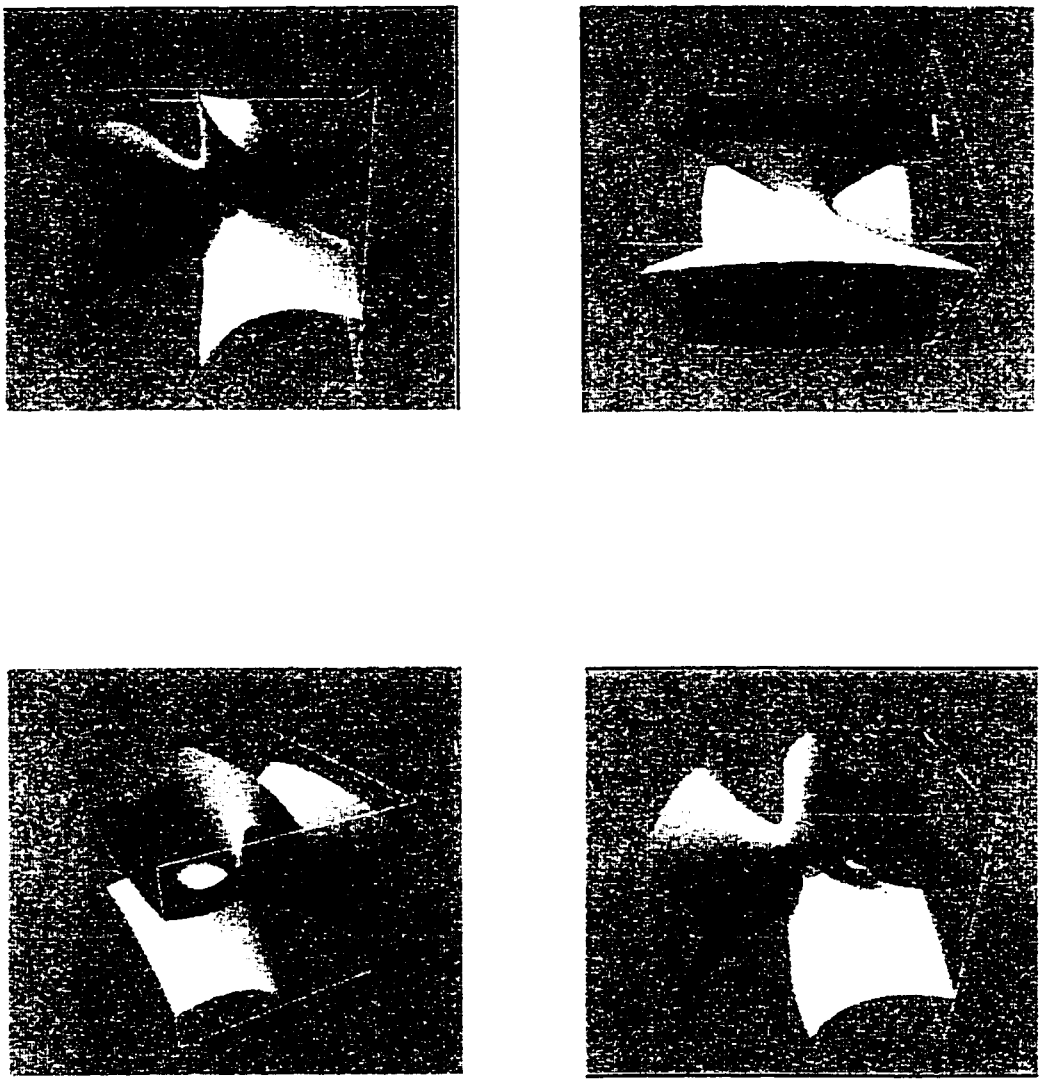


Figure 60: Example 4 (Iso-value = -0.01 and Cube Size = 4.0)

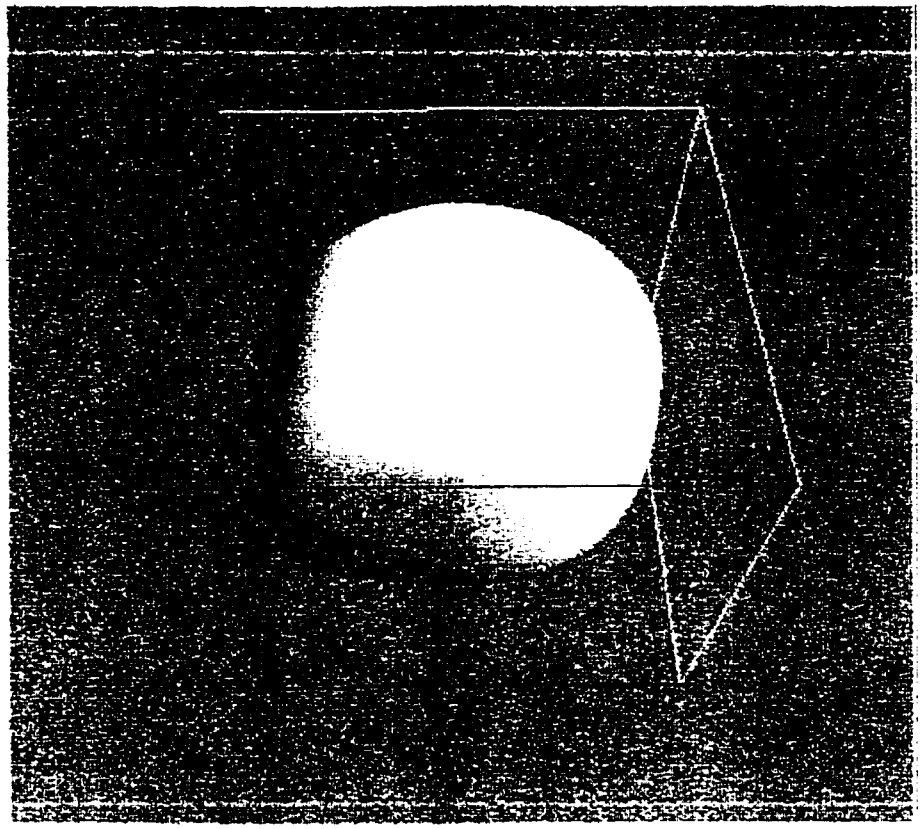


Figure 61: Example 5 (Iso-value = 1.0 and Cube Size = 2.0)

6.2.3 Manifolds Embedded in High Dimensional Spaces

- **Example 1**

  The system of the equations

$$
\begin{aligned}
&F_1(X) = X_1^2 + X_2^2 + X_3^2 + \cdots + X_n^2 - R^2 \ , \\
&F_2(X) = X_4 \ , \\
&F_3(X) = X_5 \ , \\
&\vdots \\
&F_{n-2}(X) = X_n \ .
\end{aligned}
$$

  defines a manifold (sphere) embedded in a space of dimension n. This example shows the capability of our software in computing manifolds embedded in higher dimensional spaces. In this example, the size of the dimension space, in fact, does not effect the shape of the manifold (See Figure 51 for a picture). However, the computation time is different as shown in Tables 9-12 which report the timing and memory usage for computing and visualizing several examples. These tables also state the percentage of the total computation time spent in computing different tasks.

  It can be seen from Tables 9-12 that the total computation time increases as the embedding dimension space gets higher, while the number of disks and the time needed for visualizing the manifold are the same in the four examples. We can also conclude that most of the computation time is spent on solving the linear systems, and that as the embedding space gets higher, the percentage of time spent on solving such systems gets higher. For example, solving the linear systems takes around 42% of the total time when $n = 3$, but 64% when $n = 7$, and 83% when $n = 25$. The tables also show that computing a manifold curvature takes considerable time, and that its percentage of the total time also increases as the embedding space gets higher. Computing the manifold curvature requires repeated evaluation of the *Hessian Matrix*, the cost of which increases as the system of equations gets larger.

Total computation time	**2.922** sec.	Per.
Selecting new Points	0.077 sec.	2.6%
Computing the basis	0.064 sec.	2.1
Computing manifold curvature	0.031 sec	1.06
Solving linear systems	1.192 sec.	40.79
Computing disk intersections	0.392 sec.	13.4
Splitting and Removing Interior Pieces	0.415 sec.	14.2
Visualization time	0.047sec.	
Number of disks used	**1122**	

Table 9: Timing and Memory Usage for Constructing a Sphere in R^5

Total computation time	**5.766** sec.	Per.
Selecting new Points	0.077 sec.	1.6%
Computing the basis	0.127 sec.	2.2
Computing manifold curvature	0.218 sec.	3.78
Solving linear systems	3.706 sec.	64.27
Computing disk intersections	0.34 sec.	5.9
Splitting and Removing Interior Pieces	0.437 sec.	7.5
Visualization time	0.047sec.	
Number of disks used	**1122**	

Table 10: Timing and Memory Usage for Constructing a Sphere in R^9

- **Example 2**

  The manifold defined by the equations

$$F_1(X) = X_1^2 + X_2^2 + X_3^2 - X_4^2 = 0 \ ,$$
$$F_2(X) = X_1^3 + X_2^2 - X_3^3 + X_4 - Iso - value = 0 \ ,$$

  where F is a mapping from $R^4 \rightarrow R^2$, has already been computed in Example 5 of section 6.2.2 using the mapping $F : R^3 \rightarrow R$. Refer to Figure 61 for a picture of this manifold. The timing is given in Table 13.

  Even though Example 5 of section 6.2.2 and the current define the same manifold, the timing tables shows that the manifold in higher dimension takes more time to be compute because it contains more equations and hence more computation is involved.

Total computation time	**9.485** sec.	Per.
Selecting new Points	0.093 sec.	1.0%
Computing the basis	0.205 sec.	2.1
Computing manifold curvature	0.45 sec.	4.7
Solving linear systems	6.974 sec.	73.5
Computing disk intersections	0.342 sec.	3.6
Splitting and Removing Interior Pieces	0.546 sec.	5.7
Visualization time	0.047sec.	
Number of disks used	**1122**	

Table 11: Timing and Memory Usage for Constructing a Sphere in R^{12}

Total computation time	**62.797** sec.	Per.
Selecting new Points	0.109 sec.	0.17%
Computing the basis	2.779 sec.	4.4
Computing manifold curvature	4.79 sec.	7.6
Solving linear systems	52.268 sec.	83.23
Computing disk intersections	0.448 sec.	0.7
Splitting and Removing Interior Pieces	1.305 sec.	2.0
Visualization time	0.047sec.	
Number of disks used	**1122**	

Table 12: Timing and Memory Usage for Constructing a Sphere in R^{27}

- **Example 3**

  The manifold defined by the following equations

$$F_1(X) = 4.0\ R_1^2\ (X_2^2 + X_3^2) - (X_1^2 + X_2^2 + X_3^2 + R_1^2 - R_2^2)^2 + X_4,$$
$$F_2(X) = X_1^2 + X_2^2 + X_3^2 - X_4\ ,$$

  is shown in Figure 62. In this example, F is a mapping from $R^4 \rightarrow R^2$.

- **Example 4**

  The system of the equations defining this example is

$$F_1(X) = 4.0\ R_1^2\ (X_2^2 + X_3^2) - (X_1^2 + X_2^2 + X_3^2 + R_1^2 - R_2^2)^2 + X_4 + X_5,$$
$$F_2(X) = X_1^2 + X_2^2 + X_3^2 - X_4\ ,$$
$$F_3(X) = X_1^2 + X_2^2 - X_5\ ,$$

Total computation time	**3.468** sec.
Selecting new Points	0.079 sec.
Computing the basis	0.048 sec.
Computing manifold curvature	0.061 sec.
Solving linear systems	1.811 sec.
Computing disk intersections	0.394 sec.
Splitting and Removing Interior Pieces	0.297 sec.
Visualization time	0.063sec.
Number of disks used	**1142**

Table 13: Timing and Memory Usage for Constructing Example 5 in R^4

where F is a mapping from $R^5 \rightarrow R^3$. The manifold is shown in Figure 63.

- **Example 5**

  The manifold shown in Figure 64 defined by the following equations

$$F_1(X) = 4.0\ R_1^2\ (X_2^2 + X_3^2) - (X_1^2 + X_2^2 + X_3^2 + R_1^2 - R_2^2)^2\ + X_4 + X_5 + X_6.$$
$$F_2(X) = X_1^2 + X_2^2 + X_3^2 - X_4\ .$$
$$F_3(X) = X_1^2 + X_2^2 - X_5\ .$$
$$F_4(X) = X_1^3 + X_2^3 - X_6\ .$$

  The computation space in this example is a cube of size $(4 \times 4 \times 4)$. and F is a mapping from $R^6 \rightarrow R^4$. See Table 14 for the computation and visualization report.

Total computation time	**18.281** sec.
Selecting new Points	1.089 sec.
Computing the basis	0.221 sec.
Computing manifold curvature	0.311 sec.
Solving linear systems	10.894 sec.
Computing disk intersections	1.403 sec.
Splitting and Removing Interior Pieces	1.688 sec.
Visualization time	0.187 sec.
Number of disks used	**4073**

Table 14: Timing and Memory Usage for Constructing a Manifold in R^6

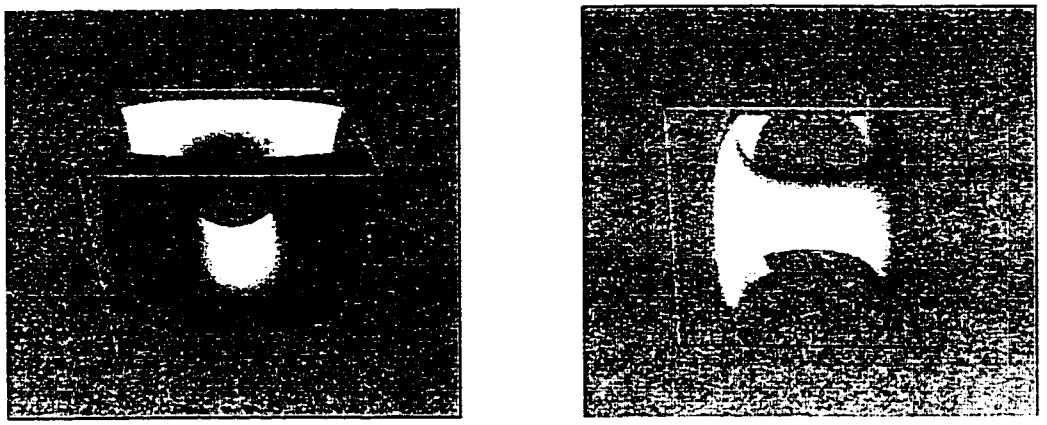


Figure 62: Manifold Embedded in R^4 (Example 3)

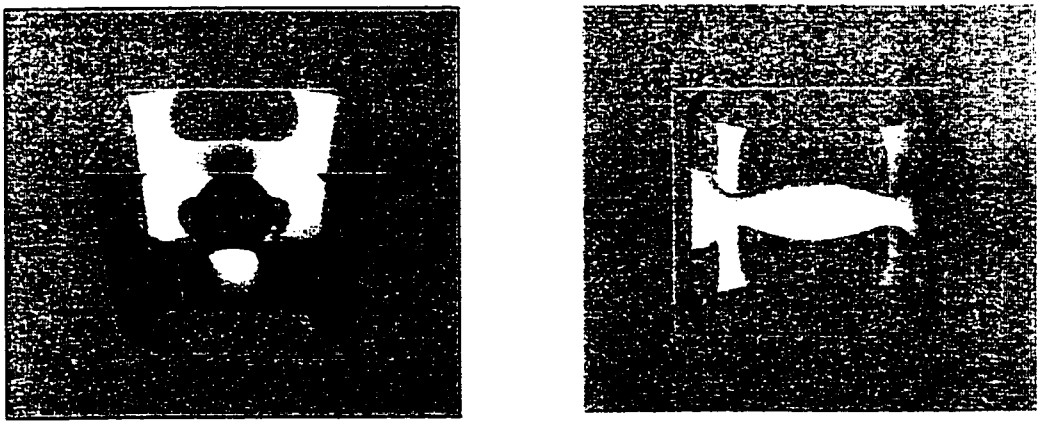


Figure 63: Manifold Embedded in R^5 (Example 4)

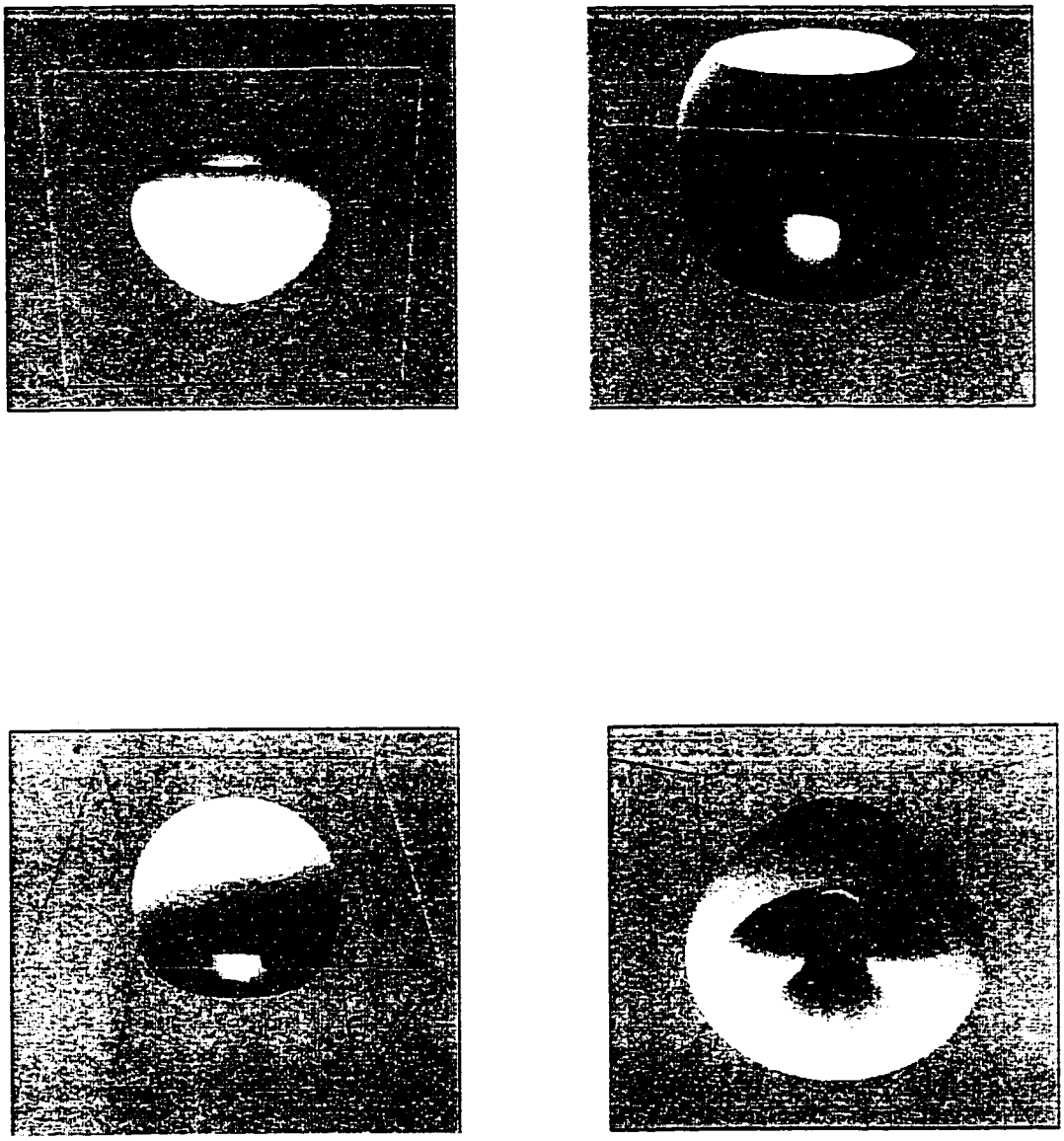


Figure 64: Manifold Embedded in R^6 (Example 5)

6.3 Conclusion

CVIDM has been developed for Computing and Visualizing Implicitly Defined Manifolds (2-manifolds) using the Windows system. It has been designed and implemented using object-oriented techniques, which make it distinct from existing software for computing such manifolds. Also, the algorithm adopted for the computation part gives our software other features that cannot be found in existing software, such as the handling of compact manifolds. Our software has been tested on several different examples of 2-manifolds. The tests illustrate the accuracy of the CVIDM in computing complicated manifolds, and its ability to handle compact manifolds.

This thesis focussed primarily on computing and visualizing implicitly defined manifolds giving rise to "regular" shapes. An extension of interset would be to compute "irregular" shapes such as Klein Bottle, a non-orientable surface which can not be embedded in three dimensions without intersecting surfaces but can be realized in four dimensions. See, for example, *http://www.math.rochester.edu/misc/klein-bottle.html.*

Another direction would be to add a new component that allows user to enter the equations directly via the GUI. Also the component could automatically generate the first and second derivatives of the equations. Highly optimized programs written in ANSI C for automatic differentiation (in forward mode) exist. See, for example. *http://www-fp.mcs.anl.gov/adic/.*

Bibliography

[1] E. L. ALLGOWER and K. GEORG. Simplicial and continuation methods for approximation, fixed points and solutions of systems of equations. *SIAM Review*, 22:28-85, 1980.

[2] E. L. ALLGOWER and K. GEORG. *Numerical continuation methods: An Introduction. Springer Series in Computational Mathematics.* Springer-Verlag, New York, 1992.

[3] E. L. ALLGOWER and K. GEORG. Continuation and path following. *Acta Numerica*, 2:1-64, 1993.

[4] E. L. ALLGOWER and S. GNUTZMANN. An algorithm for piecewise linear approximation of implicitly defined two-dimensional surfaces. *SIAM J. Numer. Anal.*, 24:452-469, 1987.

[5] E. L. ALLGOWER and P. H. SCHMIDT. An algorithm for piecewise-linear approximation of an implicitly defined manifold. *SIAM J. Numer. Anal.*, 22:322-346, 1985.

[6] J. BLOOMENTHAL. Polygonalization of implicit surfaces. *Computer Aided Geometry Design*, 5:341-355, 1988.

[7] M. L. BRODZIK and W. C. RHEINBOLDT. The computation of simplicial approximations of implicitly defined 2-dimensional manifolds. *Computers Math. Applic.*, 28:9-21, 1994.

[8] C. DEN HEIJER and W. C. RHEINBOLDT. On steplength algorithms for a class of continuation methods. *SIAM J. Numer. Anal.*, 18:925-948, 1981.

[9] E. J. DOEDEL. Numerical analysis, lecture notes. Concordia University, 1998.

[10] E. J. DOEDEL. Lecture notes on Numerical Analysis of Bifurcation Problems. Survey Lectures on Nonlinear Systems of Equations, Spring School on Numerical Software, Hamburg, Germany, March 1997, 132 pages. (Available from `http://indy.cs.concordia.ca`).

[11] E. J. DOEDEL and J. P. KERNEVEZ. *AUTO*, Software for Continuation and Bifurcation Problems in Ordinary Differential Equations. Applied Mathematics Report, California Institute of Technology, 1986.

[12] E. J. DOEDEL, H. B. KELLER and J. P. KERNEVEZ. *Numerical analysis and control of bifurcation problems.* 1991a. Part I: Bifurcation in finite dimensions. Int. J. of Bifurcations and Chaos.

[13] E. J. DOEDEL, H. B. KELLER and J. P. KERNEVEZ. *Numerical analysis and control of bifurcation problems.* 1991b. Part II: Bifurcation in infinite dimensions. Int. J. of Bifurcations and Chaos.

[14] M. HENDERSON. Computing implicitly defined surfaces: Two parameter continuation. Technical report, IBM Research Division, 1993.

[15] G. BOOCH, J. RUMBAUGH and I. JACOBSON. *The Unified Modeling Language User Guide.* Addison Wesley Longman, Inc, 1999.

[16] H. B. KELLER. Numerical solution of bifurcation and nonlinear eigenvalue problems. *in: Applications of Bifurcation Theory. Academic Press.* 1977.

[17] H. B. KELLER. *Lectures on Numerical Methods in Bifurcations Problems.* Springer Verlag, 1987.

[18] D. J. KVUGLINSKI. *Inside Visual C++.* Microsoft Press. 1997.

[19] W. L. LORENSEN and H. E. CLINE. Marching cubes: a high resolution 3d surface construction algorithm. *Computer Graphics*, 21:163–170, 1987.

[20] H. MARK and J. WARREN. Adaptive polygonalization of implicitly defined surfaces. *IEEE Computer Graphics and Applications*, 28:33–42, 1990.

[21] R. MELVILLE and D. S. MACKAY. A new algorithm for two-dimensional numerical continuation. *Computers Math. Applic.*, 30:31–46, 1995.

[22] W. C. RHEINBOLDT. Solution fields of nonlinear equations and continuation methods. *SIAM J. Numer. Anal.*, 17:221-237, 1980.

[23] W. C. RHEINBOLDT. Numerical analysis of continuation methods for nonlinear structural problems. *Computer and Structure*, 13:103-113, 1981.

[24] W. C. RHEINBOLDT. On a moving frame algorithm and triangulation of equilibrium manifolds. *in: Bifurcation: Analysis, Algorithms, Applications ISNM*, 29:256-267, 1987.

[25] W. C. RHEINBOLDT. On the computation of multi-dimensional solution manifolds of parametrized equations. *Numer. Math. Appl.*, 53:165-181, 1988.

[26] W. C. RHEINBOLDT and J. V. BURKARDT. A program for a locally parameterized continuation process. *ACM Transactions on Mathematical Software*, 9:236-241, 1983.

[27] N. C. SHAMMAS. *C/C++ Mathematical Algorithms for Scientists and Engineers*. McGraw-Hill, Inc., 1995.

Extendable Design of a Cellular Network Simulator

Bamdad Ghafari

A Major 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

Montreal, Quebec, Canada

June 2001

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the major report prepared

By: Mr. Bamdad Ghafari

Entitled: Extendable Design of a Cellular Network Simulator

and submitted in partial fulfillment of the requirement for the degree of

Master of Computer Science

Complies with the regulations of the University and meets the accepted standard with respect to originality and quality.

Signed by the final examining committee:

_____ Examiner

_____ Supervisor

Approved by _____

Chair of Department or Graduate Program Director

JUL 13 2001 _____

Dr. Nabil Esmail, Dean

Faculty of Engineering and Computer Science

Abstract

Extendable Design of a Cellular Network Simulator

Bamdad Ghafari

Cellular Network Simulator (CNS) is a stand-alone application, intended to study the behavior and compare the performance of frequency channel assignment and handoff algorithms in different types of cellular networks. A key requirement of such a simulator is that it should be easy to implement and integrate new types of cells or networks, methods for mobility, and algorithms for channel assignment and handoff. Thus the design of CNS was required to be extendable, and particularly in the specific ways mentioned above.

CNS was designed and implemented by using object-oriented design and programming techniques. The design of CNS utilized both *Unified Modeling Language (UML)* and design patterns, while it was implemented in Java. CNS provides a convenient graphical user interface as well as a command line interface. CNS was designed and implemented in a manner to completely separate its GUI from the rest of the application. This enables CNS to run without need of any human intervention when this feature is needed. CNS is a multi-platform application that could run on any standard platform (Unix, Windows, and DOS). CNS also provides a record and playback mechanism in order to save and re-run a simulation. An in-depth description of the basic technology used in the design and implementation of CNS is provided in this report.

Acknowledgements

I would like to take this opportunity to express my sincere thanks to my supervisor, professor Lata Narayanan, for her support throughout my studies at Concordia University. Without her extraordinary guidance, this work would not have been done.

My special thanks are also due to Asmaa Alsumait. She joined me later in the project and made a big contribution to the project, by implementing methods and algorithms for frequency channel assignment, handoff, heap structure, mobile movement and also by helping me during the integration phase. Finally, thanks to all the professors and staff in the department of computer science, especially Halina Monkiewicz, for their excellent support during my studies.

Table of Contents

1 Introduction

This report will describe the detailed design and implementation of a *Cellular Network Simulator (CNS).* CNS is a utility which simulates mobile traffic and management of calls arising from this traffic in a cellular network. CNS can be used by researchers and scientists who study the behavior of different types of cellular networks, and algorithms for frequency channel assignment and handoff. In this chapter, we give a brief introduction to cellular networks, channel assignment algorithms and the motivation to develop CNS, as well as review some related work.

1.1 Introduction to Cellular Networks

Cellular telephone systems are a way of providing portable telephone services. Each telephone is connected by a radio link to a base station, which is linked to the landline telephone network. A cellular mobile communications system uses a large number of low-power wireless transmitters to create *cells.* A cell is the basic geographic service area of a wireless communications system. Variable power levels allow cells to be sized according to the subscriber density and demand within a particular region. In order to simplify planning and designing of a cellular network a hexagonal type of cell is usually used. Hexagons tile the plane without any overlap and provide a close enough approximation to the circular shape that best describes the coverage area of a transmitter. But it is important to understand that the hexagonal shaped cells are artificial and cannot be generated in the real world and also they are not suitable for planning all kinds of networks. For instance, if a cellular network is supposed to give coverage to a city, a grid of rectangular cells may be more suitable.

In order to increase capacity, the key concepts used by cellular networks are *frequency reuse* and *cell splitting*. Frequency reuse refers to the idea that the same frequency channel can be reused in different cells of the network. However, if the same channel is used in adjacent cells, this can lead to radio interference, called *co-channel interference*. Thus, there is a limitation in the extent of frequency reuse. Generally, frequency channels may not be reused in adjacent cells. Cell splitting refers to the idea that a cell may be split into several smaller cells, with lower power transmitters in each, in order to increase the amount of frequency reuse. Cells can also be added to accommodate growth, creating new cells in unserved areas, or overlaying cells in existing areas. Usually a base station covers a cell and is responsible for managing ongoing calls in the cell. When a mobile user tries to initiate a call, it requests a frequency channel from the closest base station which assigns a frequency channel to the mobile. Each mobile uses a separate, temporary radio channel to talk to the base station. However, the base station talks to many mobiles at once, using one frequency channel per mobile. It is common to pre-assign a set of frequency channels to each base station in advance, where the assignment is pre-computed to avoid interference with neighboring cells [2,4,5,6]. It is important to do this in such a way that each cell site can support the maximum possible number of mobile users. Then, when a call arrives, the base station has to make a decision about which of its available channels to assign to the call.

The mobility of users adds an extra complication to channel assignment strategies. In particular, users may move between several cells during the duration of a call. When moving to an adjacent cell a mobile would generally lose contact with the original base station. Thus, the call is "*handed off*" to the base station in the adjacent cell. If the call

were to continue on the original channel despite moving to a new cell this could cause *co-channel interference*. Thus the channel is usually released, and the new base station assigns a new channel to the call, if one is available.

A successful handoff is subject to the availability of a frequency channel in the adjacent cell. Some channel assignment algorithms give priority to handoff calls (ongoing calls) rather than new calls [1,7]. This can be done by reserving frequency channels for handoff calls. Base stations can estimate the likelihood of a handoff based on cell size, speed and direction of mobile stations. More details on cellular networks can be found in [2].

1.2 Frequency Channel Assignment Problem

Mobile telecommunication systems establish a large number of communication links with a limited number of available frequency channels. Furthermore, reuse of the same frequency channels in neighboring cells causes interference. The task of assigning frequency channels with minimal interference is the channel assignment problem. The channel assignment problem is usually treated as a "*graph multicoloring*" problem where the number of colors is minimized [6]. The phenomenal growth of wireless services has fuelled the demand for efficient assignment of frequency channels, and thus, more sophisticated algorithms are required.

1.3 The Motivation for CNS

By implementing several types of frequency channel assignment algorithms that could be run on various types of cellular networks, CNS becomes a very useful tool to study the behavior and efficiency of each channel assignment algorithm.

CNS has been designed in a manner to keep the door open for new implementations of new frequency channel assignment algorithms, network types and mobile movements. In particular, CNS has been designed with an open architecture so that any researcher with some knowledge of computer programming can add his or her own implementations of frequency channel assignment algorithms, network types or mobile movements to CNS.

1.4 Related Work

Like any dynamic simulation architecture, CNS models real world objects such as mobiles, base stations, and the geographic network area. Thus, it is important to: (a) identify the actors who represent the real world objects (b) identify discrete events (c) identify continuous dependencies, and (d) provide adequate performance [8]. In many simulators, entities communicate with each other by sending messages through a centralized messaging/events distributor object. They also create different processes or threads for each simulator object in order to improve the performance and to simulate concurrent events. In this section, we will provide several examples of such simulators. We will briefly describe each simulator and will compare it with CNS.

Simjava is a series of libraries that can be used to write stand-alone simulations developed by Institute for Computing Systems Architecture at the University of Edinburgh [10]. Simjava is a process-based discrete event simulation package for Java. A simjava simulation is a collection of entities each running in its own thread. These entities are connected together by ports and can communicate with each other by sending and receiving event objects. A central system class controls all the threads, advances the

simulation time, and delivers the events. The progress of the simulation is recorded through trace messages produced by the entities and saved in a file. CNS uses the same techniques. CNS objects communicate with each other through a centralized event distributor object. CNS also creates new threads for those objects that need extended CPU power. For example, a new thread is created for each object representing a mobile unit. CNS has different functionality compared to SimJava. SimJava is a collection of general-purpose libraries that can be used to develop a simulator, while CNS is a special-purpose simulator for cellular networks. While Simjava could have been used to develop our cellular network simulator, one of the purposes of this project was to understand how design patterns could be applied in the design of such a simulator. Therefore, we chose not to use Simjava but instead to do the design "from scratch."

MobSim++ is an object-oriented cellular network simulator, developed by the Department of Teleinformatics at the University of Stockholm [11]. MobSim++ runs on top of a parallel simulation executive, the Parallel Simulation Kernel (PSK). PSK is an implementation of the Time Warp mechanism for optimistic synchronization of Parallel Discrete Event Simulations and is also written in C++. MobSim++ like CNS simulates mobile movement, hand-off and also collects statistics for the simulation but MobSim++ does not implement any specific network type, mobile movement or frequency channel assignment algorithms. Thus MobSim++ does not meet the desired objectives of our project.

CellSim is a tool to simulate GSM and CDMA mobile cellular networks [12]. It is developed by Australia's Commonwealth Scientific and Industrial Research Organisation (CSIRO). CellSim performs static GSM or CDMA simulations generating

multiple snapshots in time. Each snapshot represents a particular arrangement of mobiles in cells and thus a particular realization of signal propagation arising from base station and mobile transmission and reception powers. Thousands of snapshots can be quickly run, with mobile positions varying to simulate full cell coverage. Histograms of Carrier-to-Interference ratios accumulated over the snap-shots reveal network performance. CellSim also offers dynamic simulations that are event-driven. A run represents a period of time in which mobiles may enter the network, initiate calls, handover to adjacent cells, terminate calls or leave the network. Thousands of such events can be quickly simulated representing full network operation with network performance measures accumulated. Unlike CNS that is not tied to any specific wireless technology, CellSim is specific to GSM and CDMA networks. CellSim is also a commercial application, and the source code is not freely available.

NS is a discrete event simulator developed in the Department of Computer Engineering at Bogazici University [13]. It provides support for wired/wireless networking with multicast capabilities and satellite networks. Mobility features include node movement, periodic position updates, and maintenance of topology boundary. These are implemented in C++. Plumbing of network components like classifiers, dmux, Link Layer, Media Access Control, channel within Mobile node have been implemented in OTCL. NS focuses on the impact of the mobile movement on routing algorithms in a large network combined from wired, wireless and satellite nodes, while CNS focuses on frequency channel assignments.

As shown in the above examples, although CNS is in many respects similar to other simulators, it provides a unique functionality tailored to our needs. In general, we emphasize three points:

(a) At the time that we started the development of CNS, there were no publicly available cellular network simulators. They were restricted to the private sector.

(b) Even now, there are several network simulators are available, but we have not found any that satisfy the requirements for CNS. Most of these products are for commercial use with a focus on simulating the network traffic. The main purpose of these products is to provide a test tool for network engineers to evaluate the capacity of the network. This differs from the CNS objective to provide a research tool to evaluate algorithms for frequency channel assignment.

(c) One of the objectives of CNS was to study and understand the use of design patterns in this application domain.

2 System Requirements

This section will describe the general, hardware, and software requirements of the cellular network simulator.

2.1 General Requirements

First, we describe the domain-related requirements, and then the user related requirements. CNS should be able to simulate relevant events in the geographic area that is covered by a cellular network. In a real life situation, usually the geographic area has been divided into a network of cells. Depending on the distribution and movement of mobile traffic, the network will be assembled in various ways and by using various cell shapes. For instance in a city, it is desired to have rectangular cells between street blocks and buildings; so a network of rectangular cells should be used. In a remote area, a hexagonal type of cell is more desirable; so a network of hexagonal cells should be used. In order to have a realistic simulation, CNS should be able to simulate different types of networks such as hexagonal and rectangular networks.

One of the main characteristics of a cellular network is that its users are not stationary. In particular, during the period of a call, a user may move to several different cells. The way that a mobile user moves can follow different patterns and could depend on several parameters such as network type. For instance in a city, a rectangular type of network is usually used and the mobile can change direction only at the intersections and in a very specific way. On the other hand, a hexagonal type of network is more likely to cover areas including highways and the mobile movement usually follows a straight line

pattern. It is important that CNS should be able to simulate different type of mobile movements.

Another important aspect of a cellular network is to understand when a mobile user tries to use the system and for how long. Mobile arrivals are generally modeled by a Poisson process, while the duration of a call generally follows an exponential distribution. CNS should follow the same conventions for call arrivals and duration.

One of the main reasons behind developing CNS is to design a tool to simulate different frequency channel assignment algorithms, in order to compare their performance in a specific type of network. Thus, it is obvious that CNS should be able to simulate different types of frequency channel assignment algorithms. Furthermore, it should be easy to add new algorithms at a later date. Finally, the channel assigned to a call may change while crossing a cell boundary. Thus, the appropriate base station must be notified when this occurs so that a handoff procedure may be called into effect.

The main motivation behind the development of CNS is to simulate channel assignment algorithms, in order to study their behavior and compare their performance. Thus appropriate statistics should be collected, and CNS must provide a mechanism to perform this function.

Next, we describe the requirements that arise from the types of expected users. CNS is expected to have two classes of users. The first set may use CNS to demonstrate channel assignment algorithms, for example, for teaching purposes. Thus, a good user interface will be required for this set of users. The second set is expected to use CNS to experiment with newly designed algorithms. This will require stepping through a simulation and probably a record mechanism. To do performance evaluations, the same

set of users is expected to set up and run several simulations to gather data. In this anticipated use, it is desired to provide a means of running CNS without a graphical user interface.

Finally, CNS should be accessible and portable. The best way to make a software tool accessible is to make it Web-enabled. In order to achieve the portability, CNS should be able to run on both Windows and Unix platforms.

2.2 Hardware Requirements

CNS can run on any standard UNIX or PC station, but due to the demanding nature of any graphical application, it is highly recommended to run CNS on a powerful station with at least 32-MB of RAM.

2.3 Software Requirements

CNS is a Java applet-based application, which operates in two modes, with graphical user interface or in the background and without graphical user interface. In both modes, Java Virtual Machine is required (VM). VM is part of JDK (Java Development Kit); to run CNS, JDK version 1.1.3 or higher is required. To run an applet, Java delivers in two ways: using appletviewer, included in JDK, or a Java-enabled web browser (for example, Netscape's Navigator or Microsoft Explorer version 4 or later). Theoretically, other Java-enabled web browsers could be used, but they have not been tested.

3 System Analysis and Design

The main objective of the design of CNS was to create a reusable extendable system with an open architecture for future enhancements. This leads CNS 's designers with no choice rather than practicing object-oriented modeling. The next section gives a brief introduction to the ideas we use, including UML and design patterns. Next, we give some use cases for our application, followed by a preliminary analysis of the system and our use of pertinent design patterns.

3.1 Design Patterns

Object-oriented modeling and design is a way of thinking about problems using models organized around real-world concepts. The fundamental construct is the object, which combines both data structures and behavior in a single entity. Object-oriented models are useful for understanding problems, communicating with application experts, modeling enterprises, preparing documentation, and designing software [8].

In late 1994. Grady Booch, Jim Rumbaugh, and Ivar Jacobson decided to join forces and create a standard set of graphical notations for object-oriented design. The result was the *Unified Modeling Language (UML)* [9]. UML is a commonly understood notation for describing object-oriented systems. UML notation has been used to describe the relationship between objects throughout this report.

Designing object-oriented software is hard, and designing reusable object oriented software is even harder. Finding pertinent objects, factoring them into classes at the right granularity, defining class interfaces and establishing key relations among them is a

tough job to do. Many experts feel that getting a reusable design right the first time is an extremely difficult task [3]. At the same time, several design problems tend to recur in applications, and experienced designers have found good solutions for them. These solutions ought to be reusable. This point of view has been explored by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides in their book [3]. In their words,

> *"Design patterns make it easier to reuse successful designs and architecture. Expressing proven techniques as design patterns makes them more accessible to developers of new systems. Design patterns help you choose design alternatives that make a system reusable and avoid alternatives that compromise reusability. Design patterns can even improve the documentation and maintenance of existing systems by furnishing an explicit specification of class and object interactions and their underlying intent. Put simply, design patterns help a designer get a design "right" faster."*

Several design patterns were studied for this project, but only those which were found applicable, have been described in this report (see Appendix A).

3.2 Use Cases

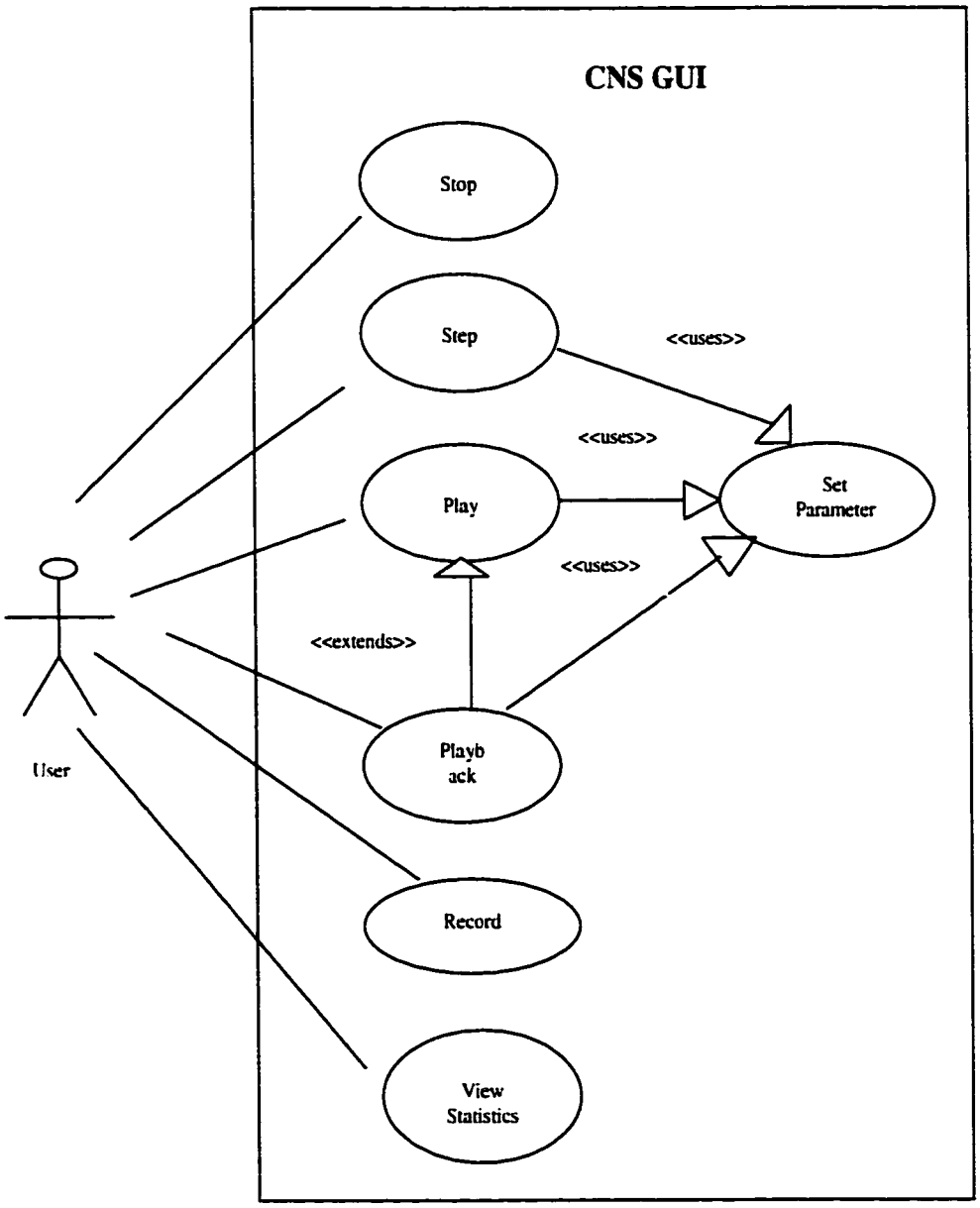


Figure 3.1

Figure 3.1 describes the available CNS's functionality from its GUI interface. The <<uses>> relationships from "step", "play", and "playback" to "set parameter" indicate

that they use the simulation values set by the "set parameter" use case. The <<extend>> relationship from "playback" to "play" indicates that "playback" is a specific case of "play". The "playback" task is a "play" of the previous simulation.

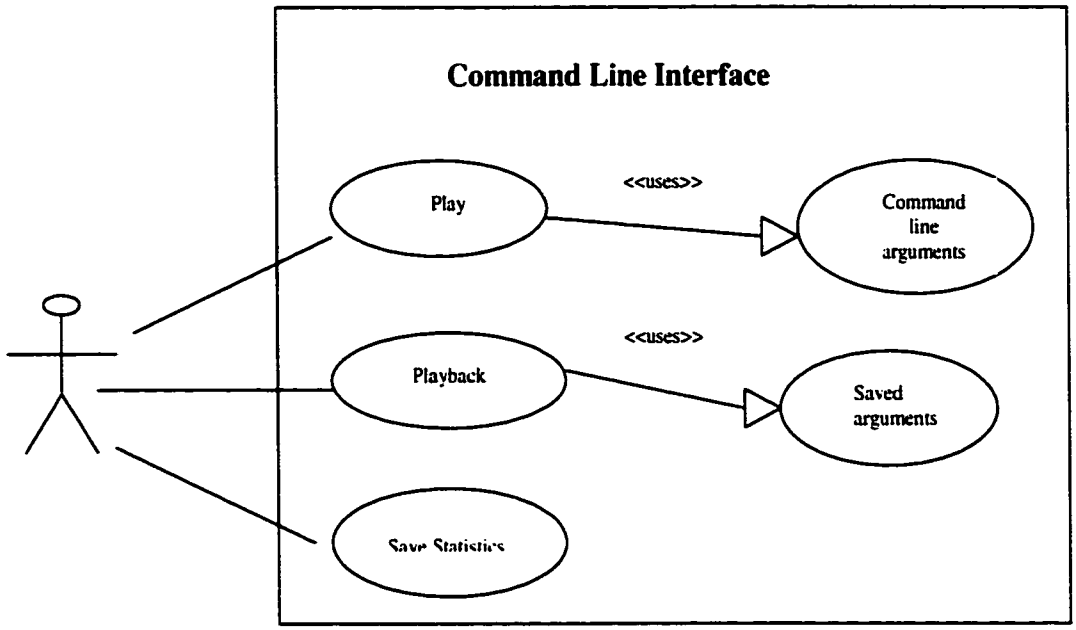


Figure 3.2

The above diagram describes the CNS functionality available from its command line interface. The <<uses>> relationships from "play", to "Command line arguments" indicate that the "play" use case uses the command line arguments values. The <<uses>> relationships from "playback" to "Saved arguments" indicate that the "playback" use case uses the simulation values saved in a simulation file. The use case also indicates that the user can save the simulation statistics.

3.3 Preliminary Analysis

One of the first steps in designing CNS was to identify the objects that can describe the characteristics of the cellular network simulator as well as establish the relationship between those objects.

From the requirements. three fundamental objects appear to be: An object representing a cell (***Cell***), an object representing a group of cells (***Network***) and an object in charge of the simulation (***Sim***). In addition to those objects there is an obvious need for an object representing a mobile station (***Mobile Station***). A base station object (***Station***) assigns frequency channels for mobile stations.

To run the simulation itself, using a scheduler that extracts simulation events from a priority queue looked reasonable; this necessitates a scheduler object (***Scheduler***), an event object (***Event***) and a heap manager object (***Event Queue***) .

Figure 3.3 gives an idea of the objects identified so far and their relationships.

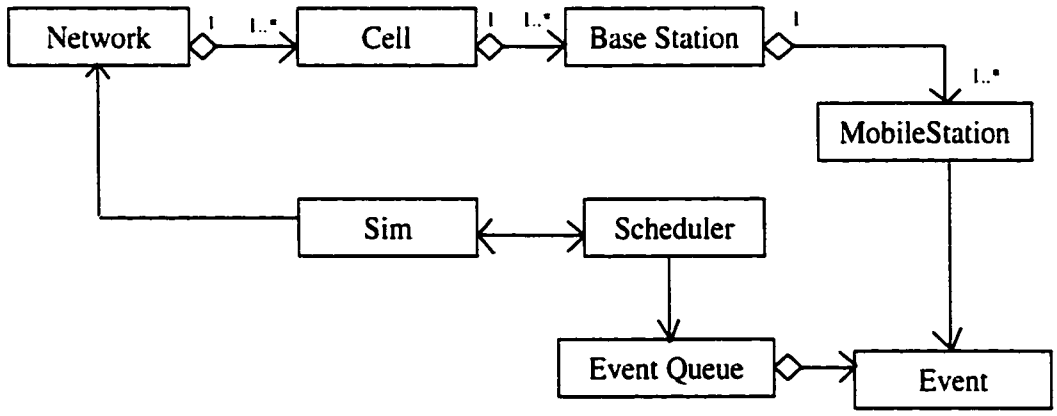


Figure 3.3

3.4 Use of Design Patterns

In this section we identify the design patterns applicable to our problem. CNS should be able to create different types of networks by using different type of cells, or using the same cell type but assembling them in a different fashion. This is exactly the job of the

Builder pattern. Builder separates the construction of a complex object from its representation so that the same construction process can create different representations. For instance the only difference between the hexagonal overlap network and hexagonal network is the way that hexagonal cells will be assembled together. Thus building the CNS network is a very good candidate for applying the builder pattern.

Some CNS objects should only have one instance such as ***Event Queue*** object, which collects all the events. In order to make sure a class has only one instance, the **Singleton** pattern is used. This can also be used to implement objects such as ***Sim*** and ***Scheduler.***

One of the key requirements is that CNS should be able to use different types of frequency channel assignment algorithms and should be able to implement different types of movements. All frequency channel assignment algorithms are related, just as all of the mobile movement algorithms are related; they only differ in their behavior. The **Strategy** pattern is useful for those classes because implementations of different algorithms are needed.

When a mobile station changes its states (created, moves, etc.) it will have an impact on other objects such as interface object and the object that collects statistics. The mobile object need not be aware of how many other objects needed to be changed. Also the mobile station objects should not be attached to the interface object in order to enable the design of dual mode operation (with or without GUI Interface). The ***Observer*** pattern provides a solution for this situation.

3.5 System Design

In this section, we will provide a graphical notation of CNS classes and their relationship with each.

Figure 3.4 describes the generalization relationship between the ***Network*** classes. All ***Network*** classes share the same methods and properties. They only differ in the way that they construct a network. Each class implements its own version of the *Build Network* method by overwriting its parent *Build Network* method.

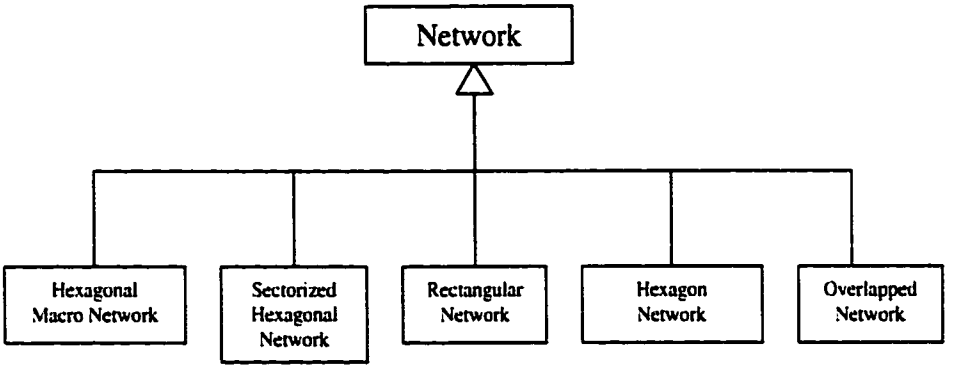


Figure 3.4

Figure 3.5 describes the generalization relationship between the ***Cell*** classes. All ***Cell*** classes share the same methods and properties. They only differ in the way that they draw themselves. Each class implements its own version of the *paint* method by overwriting its parent *paint* method.

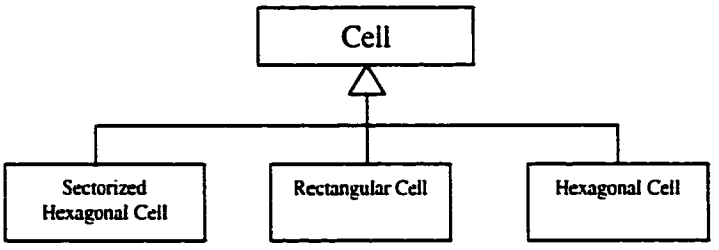


Figure 3.5

Figure 3.6 describes the generalization relationship between the ***Channel Assignment*** classes. They only share the properties and methods related to spectrum setup. Each class has its own implementation of channel assignment.

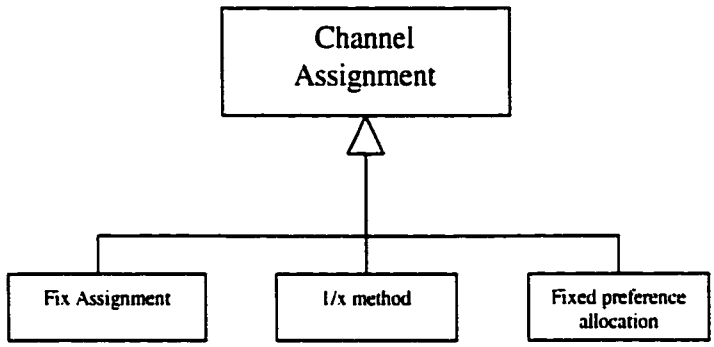


Figure 3.6

Figure 3.7 describes the generalization relationship between the ***Mobile Movement*** classes. All ***Mobile Movement*** classes share the same methods and properties. They only differ in the way that they set the direction of the next move *setDirection* and the way that they calculate the next mobile location *moveLocation.*

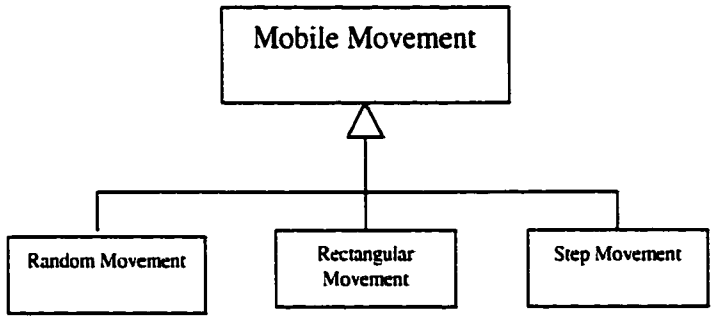


Figure 3.7

Figure 3.8 describes the relationship between the most important objects of CNS. The objects in CNS fall in four major categories: objects implementing the geographical simulation area, objects implementing frequency channel assignment, objects implementing mobility, and objects managing the simulation.

The geographical simulation area has been implemented by using the ***Network*** (which is a base class for different type of networks) and the ***Cell*** objects (Cell is a base class for different type of cells). Each network consists of several cells. There are different types of cells, and additionally there are different types of networks. This means that a network could be constructed by different types of cell objects and assembled in various ways to represent different type of networks. In order to separate construction of a network from its representation and provide a generic solution for assembling various types of networks the **Builder** pattern is used. Each Network class has a method called *BuildNetwork* which constructs a representation of the network geographic area for a specific network. The network object is also an aggregation of mobile objects. This is to enable the network object to keep track of the status of each mobile station.

Each cell has a ***Base Station*** (Base Station is a class representing a base station and is responsible for assigning frequency channels to mobile stations). Each base station has a frequency assignment algorithm. The ***Channel Assignment*** object (Channel Assignment is a base class for different type of frequency channel assignment algorithms) is responsible to find an available frequency channel and then the frequency channel will be assigned to a ***Mobile Station*** by the ***Base Station*** object. It is desired to use a single interface to access various channel assignment algorithms. The **Strategy** pattern is used

to fullfill this goal. The Channel Assignment class provides a single interface for different implementations of channel assignment algorithms.

A mobile is randomly instantiated somewhere in the network geographical area. It finds its base station and requests a frequency channel from its base station. Then it starts moving within the network. As with the channel assignment, the **Strategy** pattern is used to provide a single interface for various algorithms that implement the mobile movement. In this case, the ***Mobile Movement*** class provides this interface. If a mobile moves to a new cell, a handoff procedure is initiated and a new frequency channel will be assigned to the mobile. Change of states within a ***Mobile Station*** object will have an impact on the objects that have dependencies on it such as the ***Interface*** object and the object responsible for collecting statistics. In order to notify and update those objects automatically, the **Observer** pattern is used. The Java built-in ***Observable*** and ***Observer*** classes are used for the implementation of the **Observer** pattern.

Each action from the beginning until the end of simulation is an event. ***Events*** are instantiated by CNS's objects and managed by a priority queue. ***Heap Manager*** is a class implementing a heap data structure. The ***Scheduler*** object retrieves events from the priority queue and assigns them to the appropriate object to handle them. The ***Sim*** object is responsible for setting up, starting, and stopping the simulation.

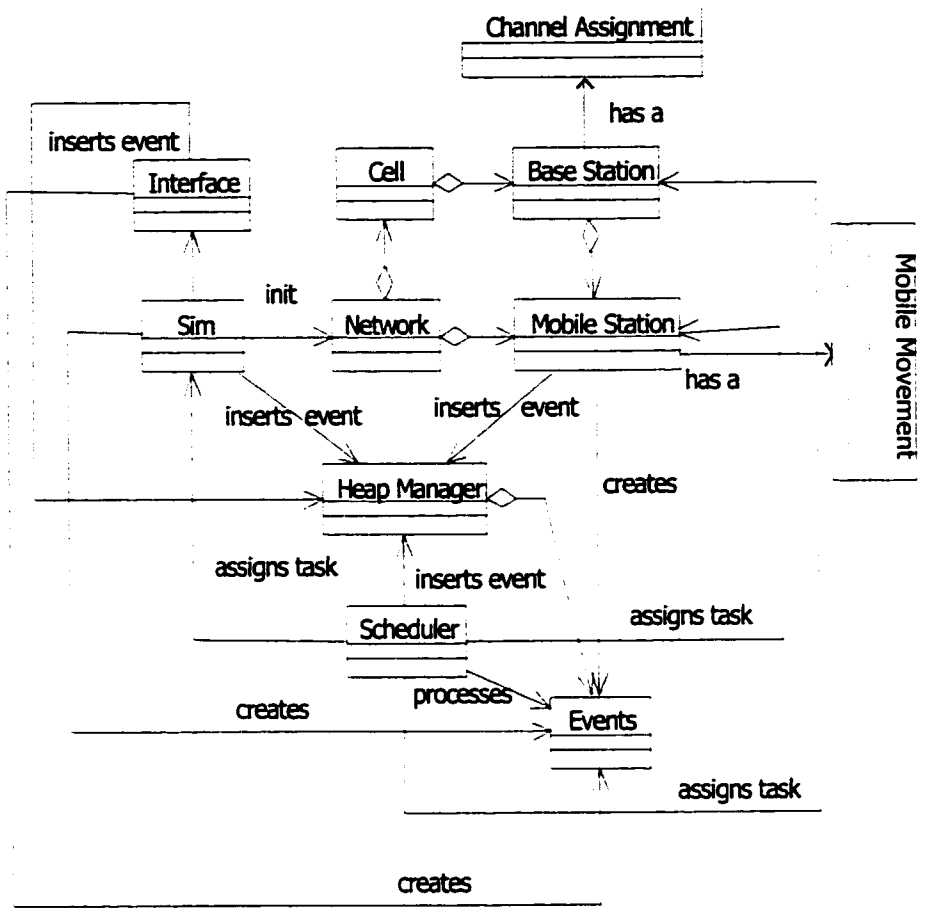


Figure 3.8: Class Diagram

3.6 Detailed Class Diagrams

The following diagrams will provide the detailed design of CNS's classes.

3.6.1 Sim

The Sim object is the CNS's main class. It is responsible for setting up and starting the simulation and stopping it. It also provides methods (*getTime* and *setTime*) for other objects to synchronize the event creation activities. Events are processed based on their event time and their priorities. It is very important for all objects to use the same clock source.

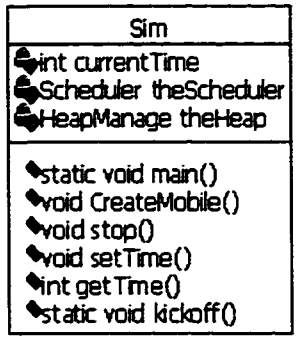


Figure 3.9

3.6.2 Cell

Cell is a base class for different types of cells. A cell has a station and it is identified by its Id (CellId). A ***Cell*** object is able to draw itself by using the *paint* method. The *paint* method would be implemented differently in different cell types. A ***Cell*** object also can determine if it contains a specific point by using its *contains* method. This is used by mobile objects to identify the cell id of their current location.

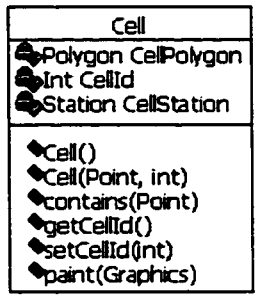


Figure 3.10

3.6.3 Hexagonal Cell

HexagonalCell is derived from the Cell class and implements a hexagonal cell (see Figure 3.12). This type of cell can be used in a buildup process of any network that uses

hexagonal cells. ***Hexagonal Cell*** inherits all properties and methods of its parent class ***Cell*** and only the *paint* method is overwritten.

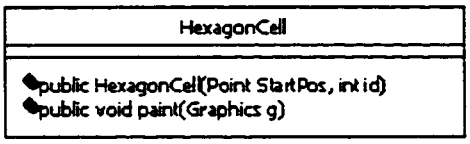


Figure 3.11

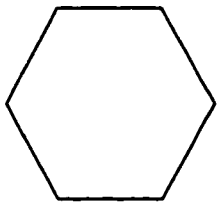


Figure 3.12

3.6.4 Rectangular Cell

RectangularCell is derived from the ***Cell*** class and implements a rectangular cell (see Figure 3.14). This type of cell can be use in a buildup process of any network that uses rectangular cells. ***Rectangular Cell*** inherits all properties and methods of its parent class ***Cell*** and only the *paint* method is overwritten.

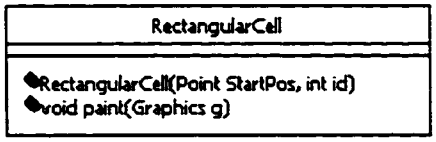


Figure 3.13

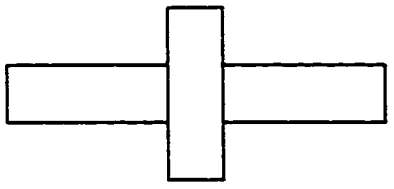


Figure 3.14

3.6.5 Network

Network is the base class for different types of networks (see Figure 3.4). It is an aggregation of cells and mobile stations. The **Builder** pattern has been used to implement different types of networks. The *BuildNetwork* method assembles cells together and builds different types of networks. The ***Network*** object also keeps track of all mobile stations within the network, and calculates the number of active mobile stations and number of hand off. There are methods for adding (*AddMobileToNetwork*), deleting (*DelMobileInNetwork*) and also locating (*FindIndexOfMobile*) mobile objects. The ***Network*** class also has a *ResetNetwork* method to destroy the network in order to build a new one.

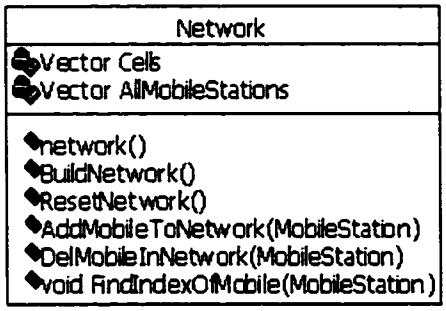


Figure 3.15

3.6.6 HexagonalNetwork

HexagonalNetwork is derived from the ***Network*** class and implements a hexagonal network by assembling non-overlapping hexagonal cells together. ***Hexagonal Network*** inherits all properties and methods of its parent class ***Network*** and only the *BuildNetwork* method is overwritten. This is where it constructs a hexagonal type of network.

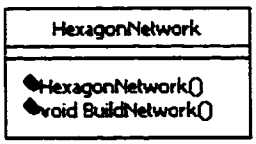


Figure 3.16

3.6.7 RectangularNetwork

RectangularNetwork is derived from the ***Network*** class and implements a rectangular network by assembling rectangular cells together, in a non-overlapping fashion. ***Rectangular Network*** inherits all properties and methods of its parent class ***Network*** and only the *BuildNetwork* method is overwritten. This is where it constructs a rectangular type of network.

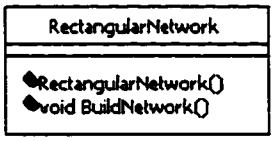


Figure 3.14

3.6.8 Base Station

Base Station objects are responsible for assigning channels to mobile stations within the cell area that they are located. A ***Base Station*** object has a ***Channel Assignment Algorithm*** object; which in different simulations, can be instantiated to different types of channel assignment algorithms. Station color is an attribute used by certain channel assignment algorithms, and is set by the network. A base station also maintains a list of all the mobile stations it is currently handling using *AddMobile* and *DelMobile*. A station object maintains a list of available, used and reserved frequency. channels using *AddToFrequencyList* can be used to add frequency channel to any of the frequency lists (ListOfAvailabeFrequency,ListOfReservedFrequency,ListOfUsedFrequency).

RequestFrequency returns an available frequency channel to a mobile request. *FrequencyRelease* is to delete a frequency channel from a frequency list. Reserved frequency channels may be used for handoff purposes because it is generally more desirable to drop a new call rather than terminate an ongoing call. *AddHandOff* will add

the time of the next hand-off to the HandOffQueue List. NewCallQueue maintains a list of new calls using *AddNewCall* .The ***Base Station*** object will decide to drop or accept a new call by consulting with these two lists. When a ***Base Station*** object is instantiated, it sets all its available frequency channels using its frequency channel assignment algorithm. When a mobile requests a frequency channel, the base station assigns a channel to the mobile using its channel assignment algorithm. If there are no more available frequency channels, the call will be dropped.

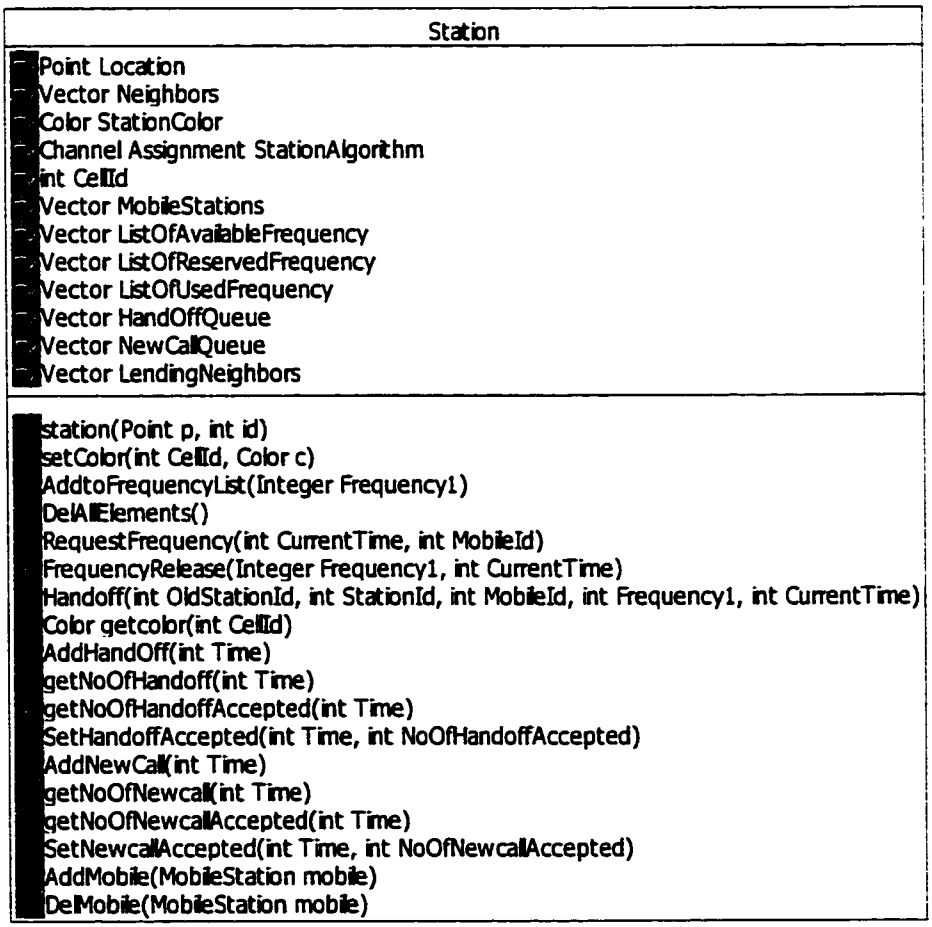


Figure 3.18

3.6.9 Mobile Station

Mobile Station objects simulate the mobility in the system. They are randomly instantiated in a location within the network geographical area. The mobile inter-arrival time is based on a Poisson distribution and its duration is based on an exponential distribution. The parameters for these can be set by the user. Upon creation, a mobile station tries to find the nearest base station by scanning the network's cells using *FindStation* method. After determining the base station, it requests a frequency channel from the station and sets its frequency channel using *setFrequency* then starts to move. The move algorithm is based on network type and also simulator parameters. When the mobile duration is finished, the mobile releases its frequency channel using the *Endcall* method. If the mobile wants to move to a new cell, it notifies the neighboring cell in advance. This information may be used to reserve a frequency channel for future handoff. The creation of a mobile station sets up the creation of the next mobile as well as all the mobile's activities during its lifetime by inserting appropriate events in the event queue.

Mobile Station also implements the **Observer** pattern by using the Java built in *Observer*. When a ***Mobile Station*** changes its states (move, hand-off, end of call, etc.), it has to notify the ***Interface*** object and the object in charge of collecting statistics. This is automatically done by implementing the **Observer** pattern. The ***Mobile Station*** adds all objects that have to be notified of its state's change to its list of observers and then notifies every object that exists on that list. The ***Mobile Station*** provides several interfaces to enable other CNS objects to retrieve the mobile's run time parameters. Those interfaces are *getTime, getDuration, getMobileId, getStation and getOldStation.*

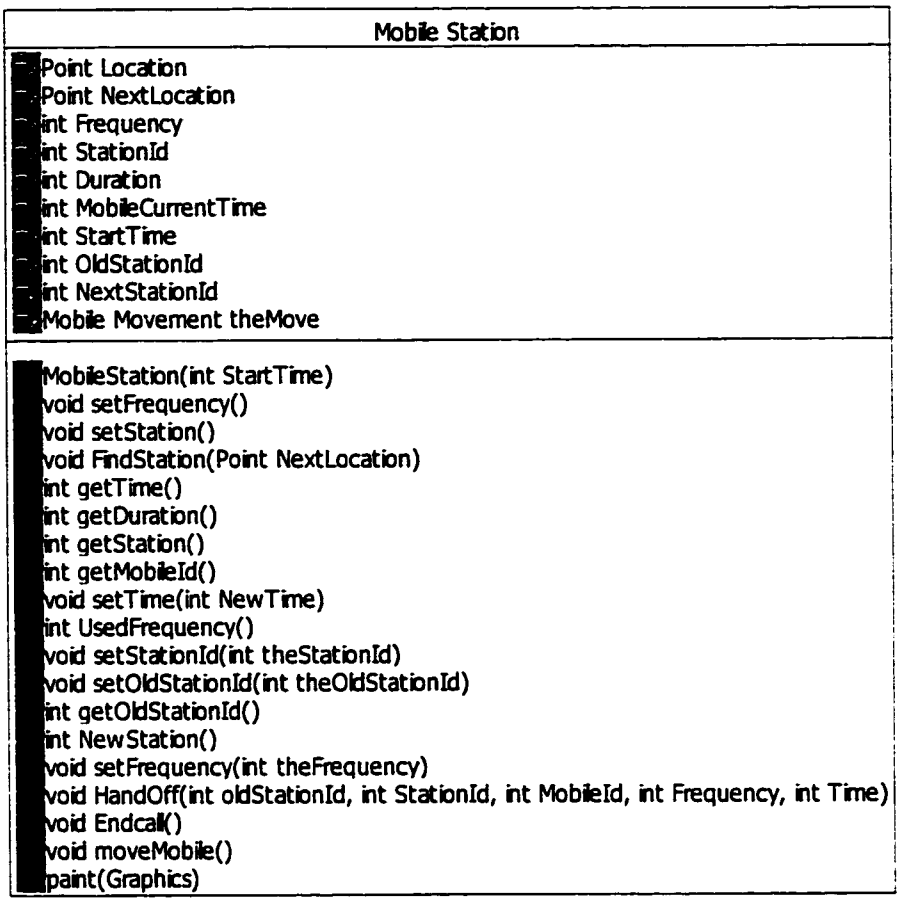


Figure 3.19

3.6.10 Channel Assignment

This is a base class for different frequency channel assignment algorithms. Each ***Channel Assignment*** has a spectrum which is a list of its available frequency channels. The ***Channel Assignment*** class is responsible to set up the spectrum. It performs this function by analyzing the network structure and considering its own specific implementation. The ***Channel Assignment*** class is implemented by using the **Strategy** pattern. The ***Channel Assignment*** class is an abstract class, meaning there is no object instantiated by this class and no method that is actually implemented. Different ***Channel Assignment*** classes are

derived from this class (see Figure 3.6). Those classes include the implementations of the channel assignment algorithms.

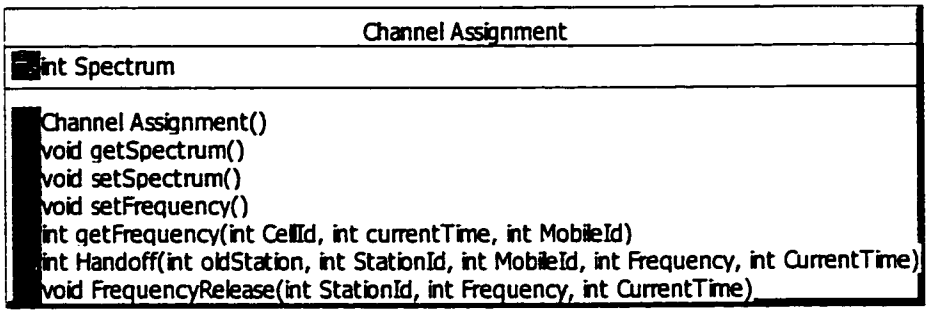


Figure 3.20

3.6.11 Fixed Assignment

This is derived from the ***Channel Assignment*** class and implements the Fixed channel Assignment. This algorithm is described in [4].

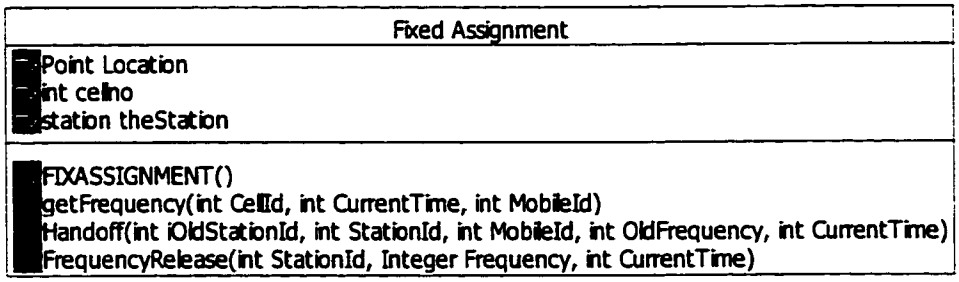


Figure 3.21

3.6.12 Mobile Movement

This is the base class for different types of movement algorithms. Mobile Stations can move in different patterns. For each specific type of movement, one sub-class is inherited from the Mobile Movement class (see Figure 3.7). Mobile Movement

implements the *getLocation* and *move* methods that are common to all movement algorithms. The implementation of the rest of methods is left for the sub-classes.

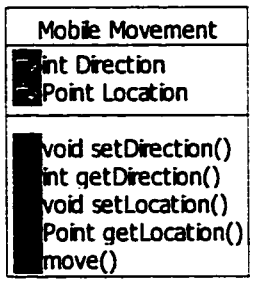


Figure 3.22

3.6.13 Random Movement

RandomMovement is derived from ***Mobile Movement*** and it is used in the ***Hexagonal Network***. In this method a mobile randomly chooses a direction at the start of its movement and keeps moving in the same direction for the rest of its duration.

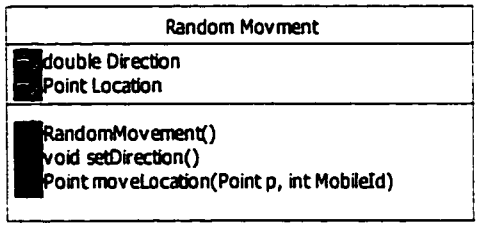


Figure 3.23

3.6.14 Rectangular Movement

RectangularMovment is derived from ***Mobile Movement*** and it is used in the ***Rectangular Network***. In this type of movement, a mobile first examines its location within the cell. If the mobile is located in the vertical part of the cell it moves either upward or downward; otherwise it moves either right or left. When the mobile crosses an intersection, it randomly decides to move up, down, right or left. Unlike ***Random***

Movement, in this type of movement, a mobile has to examine its location after each step to check for an intersection.

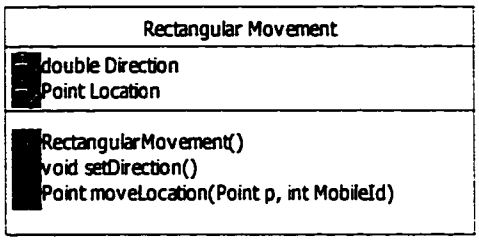


Figure 3.24

3.6.15 Step Movement

StepMovement is derived from ***Mobile Movement*** class and it is used in a hexagonal type of network. It is similar to ***Random Movement***, the only difference is that the mobile station changes its direction after a number of steps specified by the user.

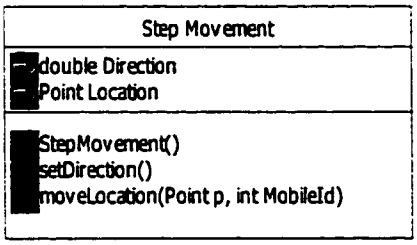


Figure 3.25

3.6.16 Event

Events for CNS are as fuel for a car. ***Event*** objects are created and handled based on the event map described in Table 3.1. Each event has a creator, who creates the event, a handler which is the object that performs the action expected by the event, a time when it must be executed, and a priority. ***Events*** are created and inserted into the priority queue by ***Mobile Station*** objects and the ***Sim*** object. The ***Scheduler*** object removes them from

the priority queue and assigns them to an appropriate object to handle the event. Table 3.1 describes different types of events and their attributes.

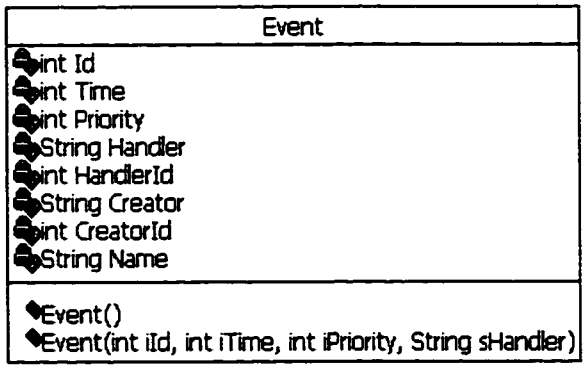


Figure 3.26

Event Name	Creator	Event Id	Handler	Priority	Time
Start	Interface	1	Simulator	1	
Stop	Interface	2	Simulator	1	
Step	Interface	3	Simulator	1	
Create Mobile	Simulator	4	Mobile Station	2	
Request Frequency	Mobile Station (I)	5	Base Station(J)	4	
Move	Mobile Station (I)	6	Mobile Station (I)	4	
End	Mobile Station (I)	7	Base Station(J)	2	
Hand Off	Mobile Station (I)	8	Base Station(K)	2	
Reserve Frequency	Mobile Station (I)	9	Base Station(K)	3	

Table 3.1

3.6.17 Heap Manager

The ***HeapManager*** object implements a heap data structure and it provides methods to store, retrieve and manage the events based on their priority and the time.

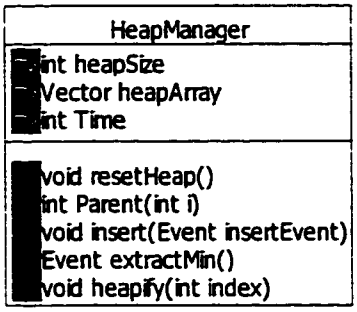


Figure 3.27

3.6.16 Scheduler

The ***Scheduler*** object is responsible for retrieving the events from ***HeapManager***, analyzing them and assigned them to a proper handler object. Each ***Event*** object has an Id and also the identity of who is responsible for this event. The ***Scheduler*** object retrieves this information from the event object and then based on the event Id calls the appropriate method of the handler object. For more information please look at Table 3.1.

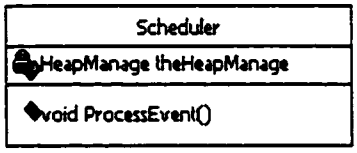


Figure 3.28

3.6.18 Interface

The ***Interface*** object provides the graphical user interface for CNS. Besides providing the standard graphical user interface functionality such as push buttons and panels, CNS is also able to respond to user requests while proceeding with the simulation. This means that CNS should have multithreading functionality. In addition, the user interface also needs to keep track of all mobile objects all the time in order to reflect a valid view of the

simulation. In order to achieve the above-mentioned goals, CNS uses Java built-in ***Thread*** and ***Observer*** functionality.

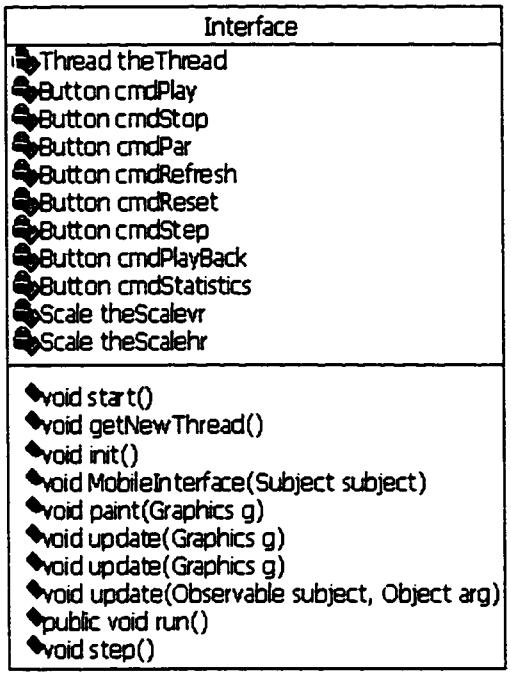


Figure 3.30

3.6.18 Global Constants & Global Variables

The ***namedConstants*** class is used to hold all named constants for CNS. HEXAGON_NETWORK, RECTANGULAR_NETWORK, MOBILE_MOVE_STATE, and MOBILE_HANDOFF_STATE are examples of named constants. The ***GlobalVar*** class is used to hold all global variables SIM_DURATION, CELL_SIZE, ALGORITHM_TYPE, MOBILE_SPEED are examples of global variables.

3.7 Sequence Diagrams

The following sequence diagrams will describe the typical use of the simulator. Figure 3.28 describes the basic use of the CNS. The ***Interface*** object inserts the start event into the priority queue. The ***Scheduler*** object takes the event from the queue and assigns it to the ***Sim*** object. The ***Sim*** object inserts the mobile event into the heap. The ***Scheduler*** takes the event out of the queue and invokes the mobile station constructor. The ***Mobile Station*** adds all of the events needed during its lifetime. This means that the ***Mobile Station*** will find out all the necessary events in advance and add them into the queue. The ***Scheduler*** takes the events one by one out of the queue and invokes the appropriate methods. Those events include requests for frequency channels, requests for movement, requests for hand off and end of call. The same scenario happens repeatedly.

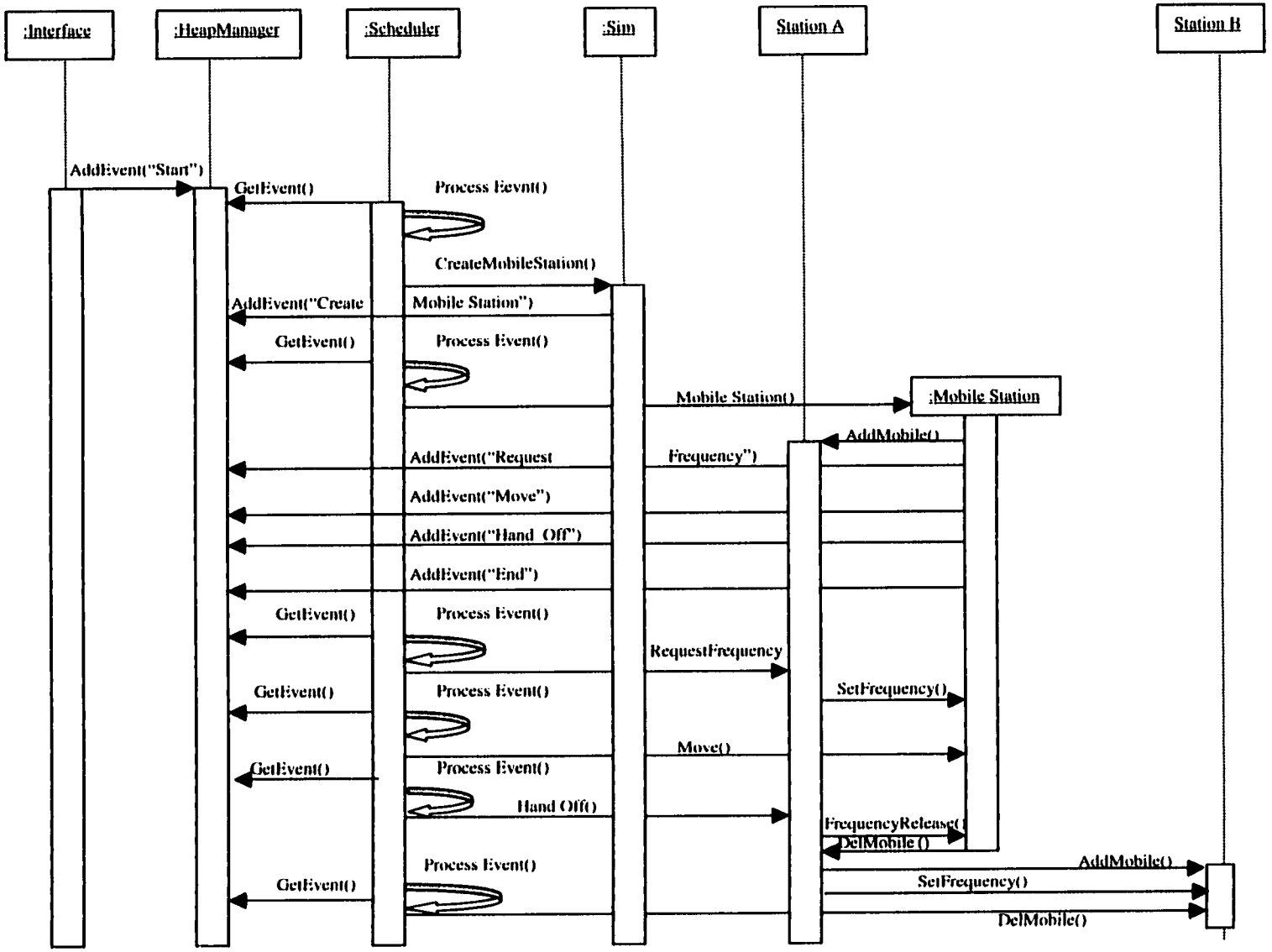

:Interface
:HeapManager
:Scheduler
:Sim
Station A
Station B
:Mobile Station
AddEvent("Start")
GetEvent()
Process Eevnt()
CreateMobileStation()
AddEvent("Create
Mobile Station")
GetEvent()
Process Event()
Mobile Station()
AddMobile()
AddEvent("Request
Frequency")
AddEvent("Move")
AddEvent("Hand_Off")
AddEvent("End")
GetEvent()
Process Event()
RequestFrequency
GetEvent()
Process Event()
SetFrequency()
Move()
GetEvent()
Process Event()
Hand Off()
FrequencyRelease()
DelMobile ()
Process Event()
AddMobile()
GetEvent()
SetFrequency()
DelMobile()

Figure 3.31

The next sequence diagram (Figure 3.32) describes the Playback mechanism in CNS. CNS events are created randomly using a random generator algorithm. The value of the seed completely determines the sequence of random numbers that will be generated. Thus, instead of recording the states of all objects during the whole simulation, CNS only records the seed of the previous simulation and feeds the random algorithm with the same seed to generate the exact same simulation. The user pushes the playback button, the ***Interface*** sets the simulation seed with the saved seed value of the last simulation and then calls the *kickoff* method of the ***Sim*** object and the simulation starts.

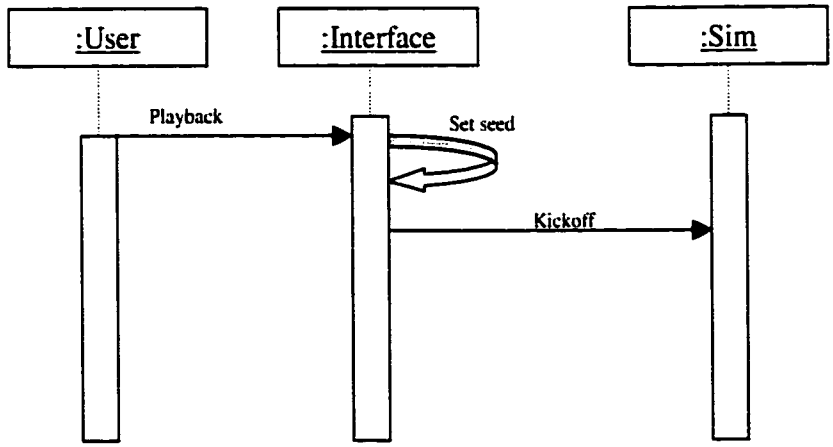


Figure 3.32

The last two sequence diagrams, Figure 3.33 and 3.34, describe the use of CNS from its command line interface. Figure 3.33 demonstrates the sequence of events when the user passes all simulation parameter as command line arguments, while Figure 3.34 shows the sequence of events when a user only passes a file name that contains the simulation

parameters. This case is very useful because with a simple script, the user can run several simulations at the same time. The user also does not need to interact with the system.

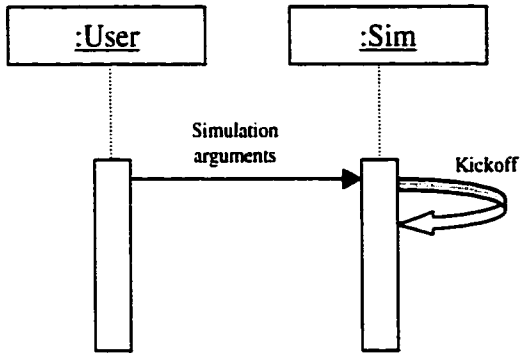


Figure 3.33

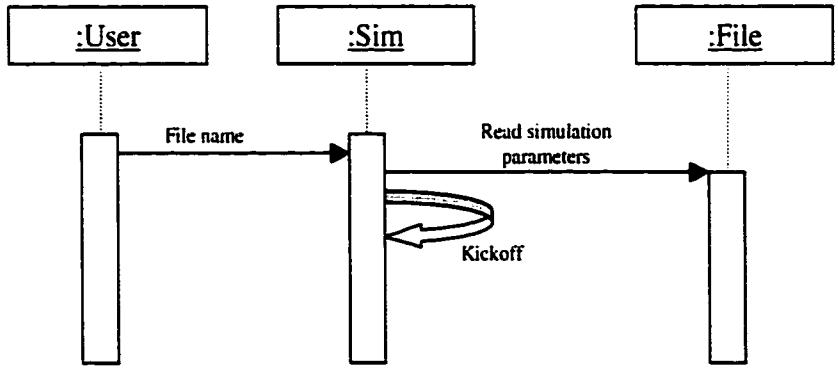


Figure 3.34

4 Interface Design

CNS has two interfaces, a graphical user interface and a command line interface. The graphical user interface should be used when the user wants to see and observe the simulation, but on the other hand, the command line user interface is useful when the user wants to run many simulations and collect statistics. The GUI interface can be run both by appletviewer or any internet browser.

4.1 Graphical User Interface

4.1.1 Main Window

The main interface of CNS is shown in Figure 4.1. In order to start the simulation the **Play** button should be pushed. The **Step** button can be used to run a simulation step by step. This is useful when the user wants to follow the simulation step by step. In this mode, the user has to push the **Step** button each time to proceed to the next step, which means to the next event. The **Stop** button will stop the simulation. The user can set the simulation parameters by pushing the **Parameters** button. This activates the parameter window (see Figure 4.2). After each simulation, the user needs to refresh the screen. This can be done by pushing the **Refresh** button. The user can change simulator parameters by using the **Reset** button. Simulation statistics can be viewed at any time by pushing the **Statistics** button. The user can play back the previous simulation by pushing the **Playback** button.

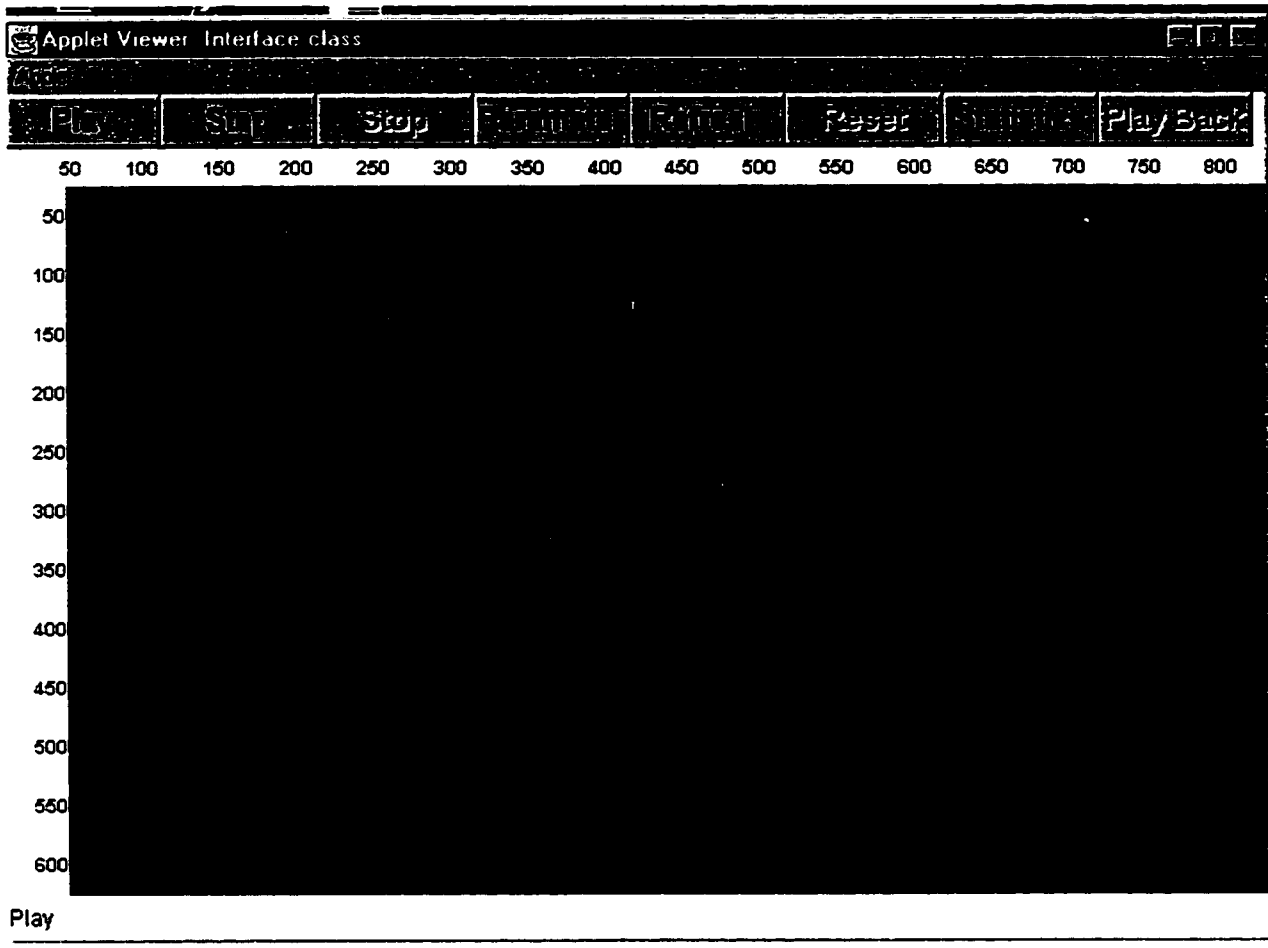


Figure 4.1

4.1.2 Parameters window

The values of **Start Col, Start Row, Stop Col, and Stop Row** will determine the simulation network boundaries. The value of **Call Duration** represents the average duration of a call. The value of **Call Inter-Arrival** is the average call inter-arrival time. The value of **Simulation Duration** is the total simulation duration. The value of **Cell Size** is the radius of a cell. **Mobile Speed** is the speed of a mobile based on the cell size. For instance, a value of 1 means that in each movement, a mobile moves half of the cell

size. **Step Move** is specific to Step Movement and identifies after how many steps a mobile will change direction. It is used when a mobile needs to change direction. **Max Spectrum** is the **maximum** number of available frequency channels. **Rectangular Size** is the width of a rectangular cell and is specific to the rectangular network. The value of **Network Type** represents the kind of network that will be simulated. The value of **Algorithm Type** represents the kind of frequency channel assignment algorithm that will be used. The value of **Cell Type** is of no use for this version but has not been omitted because it might be needed in the next version. The value of **Movement Method** represents the kind of movement algorithm that will be used. The value of **Seed** will be used for generating the first random number and will be used only once at the start of the simulation.

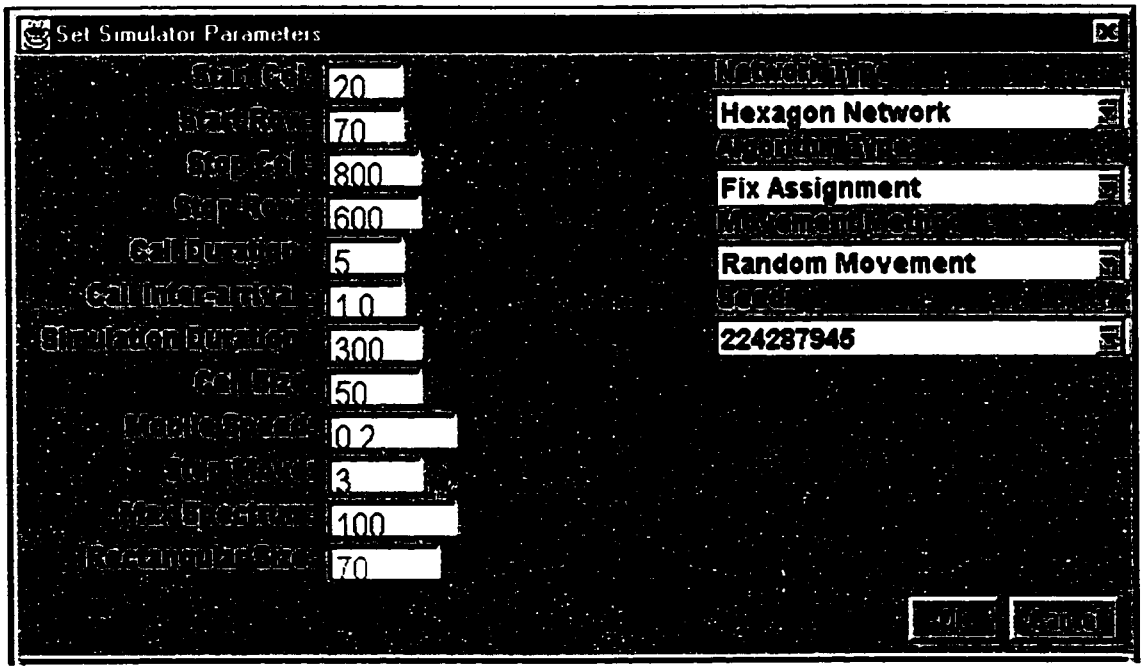


Figure 4.2

4.1.3 Statistics window

The parameters are self–explanatory. Other parameters can easily be added as needed.

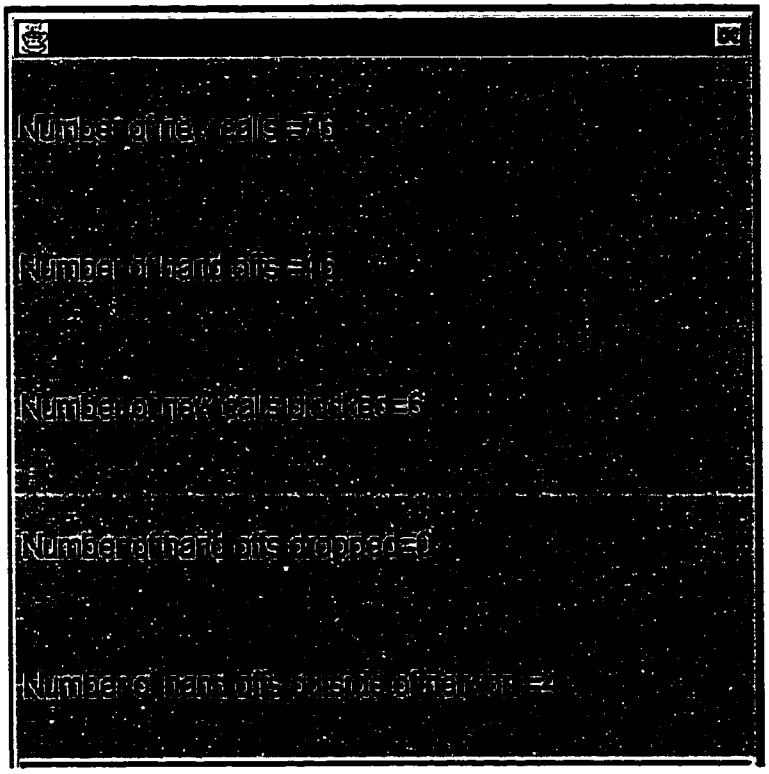


Figure 4.3

4.2 Command Line Interface

The command line user interface should be used when it is desired to run a simulation without need of any interaction. This is a very useful option to run long simulations or several simulations at the same time. All simulation parameters can be set by the users; otherwise the default parameters will be taken. For more information, please see Appendix B.

5 Conclusions

5.1 Design Lessons

Designing and implementing a robust object-oriented application is a demanding and challenging job to accomplish, and almost impossible to get it 'right' the first time.

Making mistakes is unavoidable and there is a need for constant review and correction of the design. However, the most important thing is to learn a lesson from each design project and try to use it in the future.

The design and implementation of CNS was not an exception. After several months of design and even implementing part of the application we had to stop and redesign almost every thing from scratch. It is important to identify those mistakes in order to avoid them in the future. Here are the two of the most important factors that led us to redesigning the system.

1- **Unclear Requirements:**

The system requirements were unclear in the beginning of the project, partly because of miscommunication leading to different understanding of the same problem, and partly because of undecided implementation decisions that led the programmer to make a decision by his own and assume that this is what is required but ended up to be wrong. It is important to spend some time and make sure that the problem has been understood by everybody in the project and prepare a detailed written list of all requirements.

2- Lack of experience applying Design Patterns:

For solving the same problem sometimes it is possible to apply different patterns but choosing the right pattern that suits the problem the best needs experience and deep

understanding of the patterns. Designers should try to understand a pattern and its use in depth before applying them in a design model.

5.2 Future Enhancements

The following areas could be considered for the future enhancement of CNS.

Network Type: Currently two network types have been implemented, Hexagon Network and Rectangular Network, other types of Networks should be implemented in the future such as macro/micro and sectorized networks.

Movement Type: Random Movement, Rectangular Movement and Step Movement have been implemented, and more types of movements could be added to the system.

Channel Assignment: There is definitely a need to add more frequency channel assignment algorithms to the system.

GUI Enhancement: There are a few areas regarding the graphical user interface that, could be improved such as: adding diagrams to the Statistics window, and making the Parameters window more user friendly. In addition, some cosmetic changes could be considered to make the interface more elegant.

5.3 Extendability

The design of CNS was intended to be extendable. In particular, one of the requirements was that it should be easy to add new types of networks, channel assignment algorithms and mobile movements. While complete extendability is difficult or impossible to achieve in any design, we believe the more limited extendability mentioned above has been achieved by our design. In general, to add a new network (or channel assignment algorithm or movement type), a new class for the new network type

should be created which is derived from the ***Network*** class, some methods should be overwritten, and the Parameters window in the ***DataEntry*** class should be modified. The other classes remain unaffected. In this section we describe in detail how CNS can be extended as well as limitations to the extendability.

Network Type

In order to add a new network the following steps should be taken:

1- The ***Network*** class which is the base class for all network classes should be modified to be able to identify the new network type and invoke the constructor of the new network class.

2- A new class should be created. This class should inherit from the ***Network*** base class. The new class should inherit all attributes and methods of its parent ***Network*** class. The only method that must be overwritten is *BuildNetwork*, which should construct the specific implementation of the new network. The construction algorithm is specific to the new network type. However the construction algorithm for different networks may be similar. For example, the algorithm for overlapped hexagonal network will be similar to hexagonal network, except that the boundaries of cells should be calculated in such a way that the neighboring cells overlap.

3- If the new network type includes a cell type that has not defined yet (cell type beside Hexagonal and Rectangular) then the new cell type should be created. The steps are as follows: (a) Modify the GUI to add new cell type (b) Create a new cell type inherited from the ***Cell*** base class. The only methods needed to be implemented are the class constructor and the *paint* method.

4- In order to modify the CNS GUI, the Parameters window (see Figure 4.2) should be modified to include the new network type. The ***DataEntry*** class includes the implementation of this window.

5- The new network's name should be added to the ***namedConstants*** class.

It is important to understand that any new implementation of network type might have a direct impact on other components of the system such as frequency channel assignment algorithm. This is because the distributions of spectrum and hand-off algorithms in some of the frequency channel assignments are closely related to the way that cells are organized in the network. For instance with the current implementation of CNS, a mobile queries the network in order to find its cell number. The first cell that contains the mobile will reply. Although with the network types already implemented, this is an acceptable mechanism, it may not be sufficient for a macro/micro cellular network, or a network where cells overlap, as a mobile can potentially belong to more than one cell in such networks. What mechanism to adopt may then depend on the channel assignment algorithm used.

Movement type

In order to add a new movement type the following steps should be taken:

1- The ***Mobile Movement*** class, which is the base class for all movement classes should be modified to be able to identify the new movement type and invoke the constructor of the new movement class.

2- A new class must be created and derived from the ***Mobile Movement*** base class. The *setDirection* and *MoveLocation* methods must be implemented.

3- In order to modify the CNS GUI, the Parameters window (see Figure 4.2) should be modified to include the new movement type. The ***DataEntry*** class includes the implementation of this window.

4- The new movement type's name should be added to the ***namedConstants*** class.

Movement types are closely tied to the network type. Thus any new type of movement should be implemented in a manner suitable for the specific type of network. For instance in the rectangular network mobiles are not allowed to randomly change their direction so we can not have random movement type in a rectangular network.

Frequency Channel Assignment

In order to add a new frequency channel assignment algorithm the following steps should be taken:

1- The ***Channel Assignment*** class, which is the base class for all frequency channel assignment classes should be modified to be able to identify the new frequency channel assignment type and invoke the constructor of the new frequency channel assignment class.

2- A new class must be created and derived from the ***Channel Assignment*** base class. The *getFrequency, Handoff and FrequencyRelease* methods should be implemented.

3- In order to modify the CNS GUI, the Parameters window (see Figure 4.2) should be modified to include the new frequency channel assignment type. The ***DataEntry*** class includes the implementation of this window.

4- The new channel assignment algorithm's name should be added to the ***namedConstants*** class.

As mentioned earlier, it is important to understand that not all frequency channel assignments can be used for all network types. For a specific network, an appropriate frequency channel assignment algorithm should be selected. This is because the distributions of spectrum and hand-off algorithms are based on the specific behavior and construction of the network.

Validation

In this section, we have described how CNS can be extended in order to add new network types, movement types and channel assignment types. In fact, the extendibility of the design was tested in practice in two ways. First, in the course of our implementation, we started by implementing one network type (Hexagonal Network), one movement type (Random Movement), and a simple frequency channel assignment and hand-off algorithm (Fixed Assignment). We then added a new network type (Rectangular Network) and new movement types (Step Movement and Rectangular Movement) by using the procedures described above. Secondly, CNS has already been used as a simulation tool in [1]. Al-Sumait added a new channel assignment and handoff algorithm as part of her work in [1]. This provides some direct evidence towards the extendibility of our simulator.

Appendix A: Design Patterns

The following has been extracted from the Design Patterns book, by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides [3], with minor amount of editing and rewording.

A.1 Builder

Intent

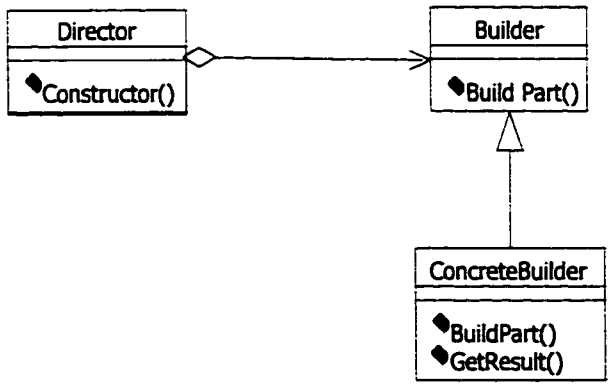
Separate the construction of a complex object from its representation so that the same construction can create different representations.

Applicability

The Builder pattern should be used when:

- the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.
- the construction process must allow different representations for the object that's constructed.

Structure



Participants

- **Builder**
  - specifies an abstract interface for creating parts of a Product object.
- **ConcreteBuilder**
  - constructs and assembles parts of the product by implementing the Builder interface.
  - defines and keeps track of the representation it creates.
  - provides an interface for retrieving the product.
- **Director**
  - constructs an object using the Builder interface.

Description:

- The client creates the Director object and configures it with the desired Builder object.
- Director notifies the builder whenever a part of the product should be built.
- Builder handles requests from the director and adds parts to the product.
- The client retrieves the product from the builder.

A.2 Singleton

Intent

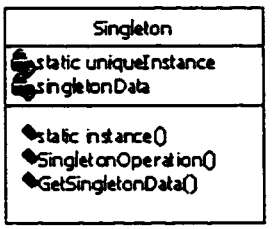
Ensure a class only has one instance, and provide a global point of access to it.

Applicability

The Singleton pattern should be used when:

- there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
- the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

Structure



Participants

- **Singleton**
  - defines an Instance operation that lets clients access its unique instance.
  - may be responsible for creating its own unique instance.

Description:

- Clients access a singleton instance solely through Singleton's Instance operation.

A.3 Observer

Intent

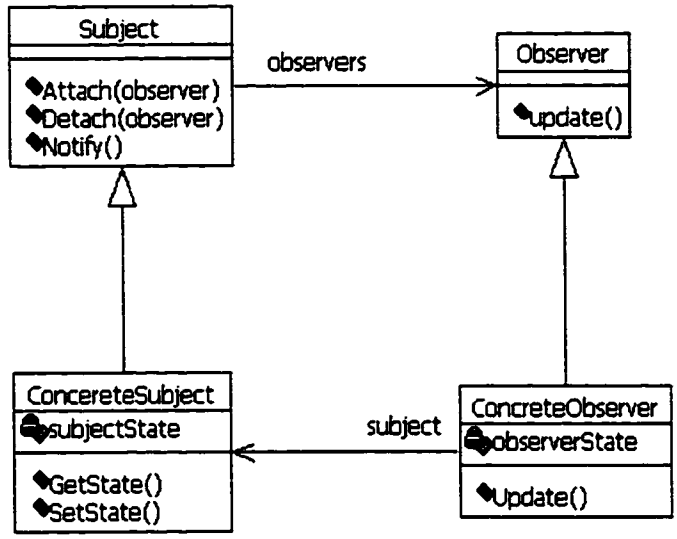
Defines a one-to-many dependency between objects so that when one object changes states, all its dependents are notified and updated automatically.

Applicability

The Observer pattern should be used when:

- an abstraction has two aspects, one dependent on the other. Encapsulating these aspects makes it possible to reuse them independently.
- a change to one object requires changing others, do not know how many objects need to be changed.
- an object should be able to notify other objects without making assumptions about who those objects are.

Structure



Participants

- **Subject**
  - knows its observers. Any number of Observer objects may observe a subject.
  - provides an interface for attaching and detaching observer objects.
- **Observer**
  - defines an updating interface for objects that should be notified of changes in a subject.

- **ConcreteSubject**
  - stores state of interest to ConcreteObserver objects.
  - sends a notification to its observers when its state changes.
- **ConcreteObserver**
  - maintains a reference to a ConcreteSubject object.
  - stores state that should stay consistent with the subject's.
  - implements the Observer updating interface to keep its state consistent with the subject's.

Description

- ConcreteSubject notifies its observers whenever a change occurs that could make its observers' state inconsistent with its own.
- After being informed of a change in the concrete subject, a ConcreteObserver object may query the subject for information. ConcreteObserver uses this information to reconcile its state with that of the subject.

B.4 Strategy

Intent

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

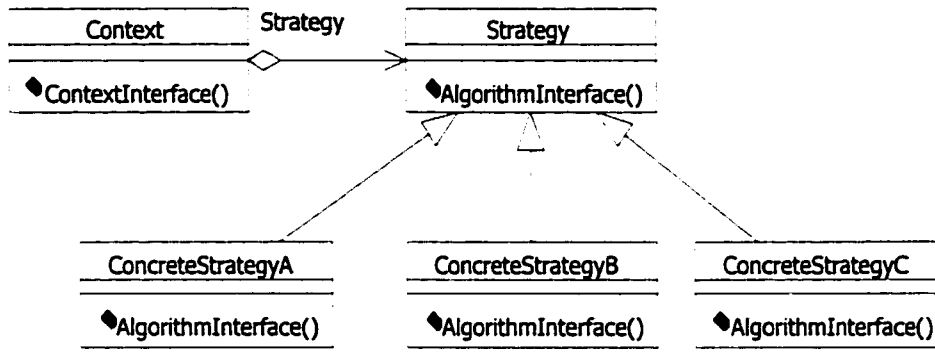
Applicability

The Strategy pattern should be used when:

- many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
- different variants of an algorithm are needed.

- an algorithm uses data that clients shouldn't know about. Use of the Strategy pattern will avoid exposing complex, algorithm-specific data structures.
- a class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

Structure



Participants

- **Strategy**
  - declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.
- **ConcreteStrategy**
  - implements the algorithm using the Strategy interface.
- **Context**
  - is configured with a ConcreteStrategy object.
  - maintains a reference to a Strategy object.
  - may define an interface that lets Strategy access its data.

Description:

- Strategy and context interact to implement the chosen algorithm. A context may pass all data required by the algorithm to the strategy when the algorithm is called. Alternatively, the context can pass itself as an argument to Strategy operations. That lets the strategy call back on the context as required.
- A context forwards requests from its clients to its strategy. Clients usually create and pass a ConcreteStrategy object to the context; therefore, clients interact with the context exclusively. There is often a family of ConcreteStrategy classes for a client to choose from.

Appendix B: Command Line Interface

java sim [-a] [-b] [-c] [-d] [-t] [-s] [-g] [-n] [-f] [-p] [-i] [-v] [-m] [-u] [-w]

-a: START ROW

-b: START COL

-c: END ROW

-d :END COL

-t: SIM DURATION

-s: CELL SIZE

-g:ALGORITHM TYPE

-n:NETWORK TYPE

-f: FILE NAME, name of playback file

-p:MOBILE DURATION

-i :CALL ARRIVAL

-e:MOBILE SPEED

-v: MOVE METHOD

-m :STEP MOVE

-u: MAX SPECTRUM

-w: File Name, name of the record file.

References

[1] A. Alsumait, A new scheme for prioritizing handoffs in cellular networks, MComp Science thesis, Concordia University, 2000.

[2] G. Calhoun, *Digital Cellular Radio*, Artech House, 1988.

[3] E. Gamma, R. Helm, R. Johnson, and J.Vlissides, *Design Patterns*, Addision-Wesley 1995.

[4] W. Hale, Frequency assignment: theory and applications, *Proceedings of the IEEE*, vol. 68, pp. 1497-1514, 1980.

[5] I. Katzela and S. Naghshineh, Channel assignment schèmes for cellular mobile telecommunication systems: a comprehensive survey, *IEEE Personal Communications*, pp. 10-31, June 1996.

[6] L. Narayanan and S. Shende, Static frequency assignment in cellular networks, *Algorithmica*, vol. 29, pp. 369-409, 2001.

[7] M. Oliver and J. Borras, Performance evaluation of variable reservation policies for handoff prioritization in mobile networks, *IEEE INFOCOM* '99, vol. 9, pp. 1187-1194, 1999.

[8] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W.Lorensen, *Object–Oriented Modeling and Design*, Prentice Hall, 1991.

[9] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, Addison Wesley, 1999.

[10] http://www.dcs.ed.ac.uk/home/hase/simjava/

[11] http://www.it.kth.se/labs/sim/projects/

[12] http://www.dbce.csiro.au/ind-serv/brochures/cellsim/cellsim.htm

[13] http://www.cmpe.boun.edu.tr/~emre/research/msthesis/node41.html

MQ

64080

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

INCREMENTAL VALIDATION OF POLICY-BASED SYSTEMS

ANGUS GRAHAM

A Thesis

in

The Department

Of

Computer Science

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

May 2001

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Angus Graham**

Entitled: **Incremental Validation of Policy-Based Systems**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair

_____ Examiner

_____ Examiner

_____ Supervisor

_____ Supervisor

Approved by _____
Chair of Department or Graduate Program Director

JUL 05 2001 _____
Dr. Nabil Esmail, Dean
Faculty of Engineering and Computer Science

ABSTRACT

Incremental Validation of Policy-Based Systems

Angus Graham

Policy-based systems are gaining popularity as a way to manage applications with dynamic behaviour. These systems have policies specifying the desired behaviour, entered into the system by either end-users or system administrators. In order to assure that the policies don't violate any stipulated properties of the system or conflict with one another, the policies must be validated. This validation process can take a very large amount of time as the system's policy base grows.

This thesis suggests an incremental validation method, whereby a system which has been determined to be consistent can be validated when a new rule is added to the system. "Trigger chaining" is a concept introduced in this thesis that examines which policies are triggered by the firing of a particular policy. This concept leads to new kinds of conflicts. An algorithm is suggested for incremental detection of such conflicts and is shown to operate in linear time, as opposed to complete revalidation which has quadratic complexity. Trigger chaining also leads to the detection of cyclic conflicts which are briefly discussed.

Decision tables are suggested as a suitable format for the internal representation of policies. This format provides a method of checking a policy set for completeness and could help in checking for conflicts. Also decision tables are shown to be a natural format for storing policies. It is also known how to convert decision tables into executable rules, making the transition from decision table-based policies to rule engine policies a simple one.

To my parents

Acknowledgements

The researching, preparing, and writing of my thesis has been a long, challenging, and rewarding experience. There are many people who have helped me along my journey. Some who have helped me labour over an idea, some who have graced me with their wisdom and experience, and some who have simply made my life more enjoyable. Without these people I would be no farther along than when I started. In particular, I would like to give thanks to the following people:

Dr. Thiruvengadam Radhakrishnan, for his constant flow of ideas and his inspiring words. Thanks for teaching me the art of patience and diplomacy. Thanks for giving me the freedom to explore ideas on my own and guiding me in the right direction with your wisdom.

Dr. Clifford Grossner, for helping me appreciate the skill of organization and for helping me strive for perfection. Thanks for the knowledge and insights you have given me both for the academic world and the business world.

Nortel Networks for their financial support during the researching of this thesis. Thanks to Richard Brunet for taking me under his wing while at Nortel.

Yota Karvelas for being the perfect lab partner. I have bounced many ideas off her head only to have them come back clearer and more solid. Thanks for all the laughs, and always encouraging me to turn up the music, no matter how weird it was.

My parents, Marilyn and Ray, for all the love, moral, and financial support they have given me and all the sacrifices they have made for me over the years.

Emily Bradshaw for all her love and moral support. Thanks for listening to my problems whenever I needed an ear, no matter how big, how small, or how computer science-related they may have been.

Josie McSoriley for helping me through a summer of setbacks, and for being supportive during a difficult time in my thesis work.

My cats, Daisy and Alice, for reminding me on a daily basis that above all the most important things in life are eating, sleeping, bathing, and a good chase around the backyard.

Table Of Contents

List of Figures

1. Introduction

1.1. Policy Based Systems and Policy Validation

Policy based systems offer the capabilities to dynamically change the behaviour of software. Such systems are gaining wide popularity in the industry today. Applications for these systems range from event notification software [1],[2],[3] to network management [4],[5],[17],[19] to electronic commerce [12],[11].

Policies can be entered into a system to instruct the system what actions should be taken when certain events occur, who is permitted to perform particular actions, and who is not. The system can either allow all end-users to enter their own policies, or to have one or multiple policy administrators to be in charge of this task. Policies can be entered before the execution of the system begins, but many systems allow the entry of policies to occur during system execution as well. The form in which the policies are entered varies from system to system. Some systems require that policies are entered in a strict code-like format, whereas others allow natural language input.

When policies are entered into the system, they must be checked to see if their syntax is correct. This consists of making sure the policy uses language understood by the system, and in such a way that the system understands what is meant by the policy. If the policy is not syntactically correct, the policy will not be understood by the system and there will be no way to execute it. This however, is not the only requirement a policy must meet in order to be accepted by the system.

After syntax checking the policies must be validated. When a new policy is entered into the system, it is not necessarily consistent with the rest of the system. A policy could order a combination of actions which are illegal in the application, or actions that conflict with actions specified by another policy. To detect any such anomalies, a validation process is needed. Validation can be performed at specification time, before

any of the policies have been executed, or at runtime, catching conflicts as they are triggered but before they are executed.

Detecting conflicts between policies is an important concern, and solutions have been provided to tackle the problems of both specification time and run time policy validation. Validating at specification time ensures that conflicts will be detected before the execution of the system. Eliminating these conflicts before the execution of the system means that less time must be spent resolving the conflicts at runtime. Validating at runtime allows the system to catch conflicts which could not be predicted before the execution of the system. These two types of validation complement each other and therefore both forms are needed.

As policy based systems get larger, the problem of policy validation becomes more complex. Systems may have a very large number of policies, which would mean the validation process would take a long time if all policies are considered. However if a small change is made, the number of policies affected by this change could be relatively small. If the entire set of policies is validated every time a small change is made, then the system is spending a great deal of time performing validation which may not be practical, or necessary. Similarly in order to validate some systems, the policies may have to be brought offline in order to perform the validation. If the policy set is changed often, this would pose a serious problem when providing users with continuous service. Clearly this delay makes it unacceptable to revalidate the entire set of policies every time a change is made if the set of policies is large.

In this thesis, we will suggest incremental policy validation as a solution to validating large policy sets. More specifically, we will examine how to determine if a set of policies is consistent after a small change to the set has been made. This will be done by finding only those policies which are affected by the changed policies, and then validating that small subset. We will look at conflicts being detected at specification time as opposed to at run time. We will also provide a method of incremental validation that

performs well as the number of policies grow. This method will be analysed to show its time complexity and compared to a non-incremental validation solution.

We will introduce the notion of storing policies in decision tables in the system. It will be shown how this format can help check a set of policies for completeness. Also we indicate that the method used for checking decision tables for consistency could possibly be used in future to detect conflicts in policies. In order to accommodate all of the policy information, the decision table format was modified. These modifications are explained, as well as the methods to enter policy in this format and to create executable rules out of this format.

The concept of "trigger chaining" has been introduced in this thesis in order to see which policies will be triggered by the firing of a particular policy. This new concept introduces a new kind of conflict, in that a policy firing immediately after another may undo the actions of the first. This new type of conflict is explained and a method of incremental validation is developed in order to detect these conflicts. The method is analysed and compared to performing an exhaustive revalidation of the entire system. The concept of trigger chaining also introduces another kind of conflict, a cyclic conflict, which is introduced but not discussed in detail in this thesis.

Policy scope is something we have introduced into our policy model. Scope is what nodes of the system a particular policy should affect. Some policies should affect the whole system, while other policies may only be applicable to certain areas of the system. Scope is an important factor when looking at policy conflicts. Two policies which might otherwise conflict will not if their scopes don't overlap.

1.2. Organization of this Thesis

In Chapter 2 policy and its terminology are introduced, and a couple of policy frameworks are presented as examples. We also present the policy model used in this thesis. Chapter 3 introduces the benefits of storing policies in a modified decision table

format. In Chapter 4 we present a solution to the problem of incremental policy validation. We also introduce the concept of trigger chaining and provide an incremental validation solution for trigger chaining conflicts. An analysis of the algorithm is performed in this chapter to demonstrate that the algorithm performs well as the number of policies gets to be very large. In Chapter 5 we present an existing policy framework and demonstrate how our incremental validation technique would be added to this system. Chapter 6 examines the contributions of this thesis, while Chapter 7 discusses directions for possible future work.

2. Policy Based Systems

Policy-based systems have been studied by many different researchers in recent years. This chapter gives a summary of their research. We will start by giving a quick introduction to policies and the common terminology used when discussing policies. Next we will discuss why there is so much interest in policy-based systems and how policies can benefit software. We will next look at various policy models, in particular looking at how they specify policies, detect and resolve policy conflicts, and perform the execution of the policies. Finally, we will discuss our own policy model which we have developed using earlier models as a base. Our model will also introduce a new concept: policy scope, which is not present in any earlier models.

2.1. Introduction to Policies

The definition of **a policy** given by Lupu and Sloman [4] is "information which can be used to modify the behaviour of a system". A policy is often made up of a set of rules related to a particular aspect of an entity. Use of policies is not restricted to software. They can be used to describe how situations are to be handled in many real-life scenarios. An example would be a professor's policy governing the submission of a term paper. In this case the policy is made up of rules which the professor follows to determine what to do when a paper is handed in late, etc.

A **rule** is defined as a set of actions, which are either to be performed on, or prevented from executing on a particular entity when certain criteria are met. The rules have an attribute specifying which events trigger the rule. However, there is also a rule attribute called a constraint. The constraint is a second-level condition that must be matched in order for the rule to fire. Rules following this format are said to follow the Event Condition Action (ECA) rule paradigm [12]. Several systems use the ECA rule paradigm [17],[4],[12]. The object which is to be manipulated by the actions of the

policy is called the **policy target** whereas the object which is interpreting the policy is the **policy subject** [6].

Because there are multiple policies in a given system, it is important to examine how one policy will affect another. If there are two policies which are triggered to fire at the same time, and they contradict one another, these policies are said to **conflict.** By contradict, we mean that one policy states to perform an action a, and another policy states to perform the opposite action $\bar{a}$ or an incompatible action b. A pair of incompatible actions would be two actions which cannot be performed by the same subject at the same time, for example proposing a budget and approving a budget. Cholvy and Cuppens [7] say a rule is **inconsistent** if there exists a world such that it leads to a conflict. We can therefore say that a rule which does not lead to a conflict is **consistent.**

Intuitively a conflict refers to a **real conflict.** This means should the system execute, the conflict will definitely occur and interrupt the system's execution if no attempt at resolving the conflict is taken. The process of checking policies to see if they conflict is called **policy validation.** The term validation in this context has a different meaning than the one associated with software engineering. The term validation in software engineering is the process of checking if the implemented software meets the expectations of the user [8]. Although policies could be checked to see if they represent the expectations of the user, the term validation will not be used to this end.

Often times, conflicts can be detected when the policies are entered into the system [9]. It is very important to deal with conflicts that are detected at specification time, because when the system executes, the conflict will definitely occur, resulting in an execution problem. This could be due to a specification error, and so it is important to detect any possible conflicts at specification time and notify the administrator [6].

Conflicts can also be detected at run-time. Although it is often advantageous to detect potential conflicts at specification time, this may not always be possible. Some

conditions are based on states of the system which are unknown at the time of policy specification [6],[9]. In this case, the only way to detect the conflict is while the system is executing [4]. Although the notion of run-time conflicts is an important area of conflict detection in policy systems, this thesis will focus on the problem of specification time conflict detection.

A **potential conflict** is present in a set of policies when there are two policies which may or may not result in a conflict at runtime. The conflict is dependent on some variable values or the state of the system. The difference between a conflict and a potential conflict is that a conflict will definitely occur at runtime whereas a potential conflict may or may not happen at runtime. For example, if one rule fires when $x = 5$ and another rule with opposing actions fires when $x = y$, it is impossible to determine whether the two rules will be triggered at the same time before the system is running and the value of y is known. If a system has a potential conflict, and at runtime the state necessary to trigger the potential conflict is reached, then the potential conflict becomes a real conflict [10]. In the example given above, if at runtime we see that the value of y is 5, then the two rules will trigger at the same time and we get a real conflict.

It is important to look at Cholvy and Cuppens' definition of an inconsistent system again [7]. They state that if there exists a world such that there is a conflict in that world, then the system is inconsistent. The world they refer to would be the state of the system, including the state of all attributes in the system, which events are occurring, etc. Therefore even if there is a potential conflict, the system is considered inconsistent according to Cholvy and Cuppens. If a system has a real conflict in it, it too then would be considered inconsistent, but to indicate the difference, we will say that the system is in a state of conflict.

Definition 2.1.1:

A system is said to be inconsistent if and only if there is a potential conflict or a real conflict between two or more of its policies.

When a system is in a state of conflict, it is important to somehow resolve the conflict. In order to resolve the conflict, one or more actions are cancelled, but this is not an obvious solution. Actions cannot be cancelled arbitrarily, and the goal is to remove the least number of actions. There are two philosophies to canceling actions. If all actions are assumed to be atomic then canceling one action will not affect the others. The other approach is to cancel all actions that are triggered by the same event. In this approach either all the actions associated with a particular event will succeed together or fail together. This, although it cancels more actions, sometimes makes more sense. For example if a rule states to perform three actions which are all related to each other, such as *package item*, *update inventory*, and *ship item*, and the action *package item* is canceled, then the entire process will no longer make sense. It would make more sense to cancel all three actions in this case [12].

2.2. Why Policies are Needed

In today's world, software users are no longer content with software that performs a fixed set of functions in a preset manner. Users have various individual needs, and they want software to meet those needs. In recent years the software industry has been moving towards building software which can be customized by the user so that it can meet the individual's needs. Policies are one way in which this customizability can be delivered. By developing a common policy framework, software can easily be built around this framework, without having to design a new method of making the software customizable for each software system.

Policies also separate the behaviour aspect of the software from the main functions. This allows either the main functionality of the software, or the user's custom behaviour to be changed without affecting the other aspects of the software. This is very important, since one would not want a user to have to redefine all of their rules if one of the non-customizable components of the system was upgraded. Similarly one would not want to have to change the code of the main software system if the policy framework is upgraded. For this reason, policies are interpreted as opposed to being compiled [4].

Policies also offer a level of abstraction from code. If rules had to be defined in code, then customizing the software would be a very tedious process. An easy to use front end interface is a solution to this, but in the end, all custom behaviour needs to be defined in a low level form which can be understood by the software. If, say, the user interface lets users enter their behaviour in natural language, then there may be vague terms or double meanings which may prevent analysis of the policies at that level. Similarly analyzing it at a low-level such as code level is not always ideal either, since at that level abstract concepts like what the entire series of given commands is trying to achieve is lost. There should be some middle layer which is not as abstract as natural language and does not have any of its ambiguities, but at the same time still carries the overall concepts and goals which the low-level code loses. Policies offer this middle layer. Users can enter policies directly using some formal policy definition language, or can enter it at a higher level and have it translated to the definition language automatically. A policy can be analyzed for correctness and consistency before it is translated to the final code to be executed [17].

2.3. Policy Models

In this section we present various policy models created by other researchers. We discuss the features of each model and indicate both their strengths and weaknesses. The main criteria being examined are how the policies are entered into the system and how conflicts are detected and resolved, and how policies are executed. Finally two complete policy frameworks are presented.

2.3.1. Specifying Policy

In order for policy based systems to get their policies, there needs to be some way a policy administrator can enter policies into the system. The system may have only one policy administrator, or it may allow all of its end users to enter policies. The system could even have multiple levels of policy administrators. For example, a manager may

be permitted to set policy for all of his subordinates, while the subordinates can only set policies for themselves.

Sometimes policy is specified at a fine level using mathematical formalisms such as first order logic [13],[14], sometimes it is stated at a very high-level such as natural language [17], or a GUI based interface [15]. More commonly, however, a **policy definition language** is used to specify the policy. Even in the case that policy is specified at a higher level, the policies are often converted into a policy definition language form, which can then be used to analyze them [17],[14].

The role of the policy definition language is to represent the conditions that trigger the policy and the actions to be carried out when it is triggered. It must be formal enough to eliminate any possible ambiguities found in natural language. It should also be high-level enough to be able to represent the concept of the policy's behaviour as opposed to a series of low-level instructions [13]. This will allow the policy to be analyzed for such things as correctness of behaviour before it is translated to low-level instructions. Not only this, but being at a higher level than code will make it easier for the administrator to enter the policies, and understand them when re-reading them in the case that he needs to make further changes to the policies.

Jajodia, Samarati, and Subrahmanian [16] proposed the Authorization Specification Language (ASL) in order to specify access control policies. It provides **authorization** policies to specify whether an action is authorized or denied for a particular combination of [<object>, <user>, <role>] sets. Since the language is meant for access control purposes, this single type of policies is sufficient. One problem with this model, however, is that all authorizations are given the same level of importance.

Bertino, Jajodia, and Samarati [21] proposed a model that is also meant for authorization policies only. They do, however, divide the notion of authorizations into two levels: weak authorizations and strong authorizations. If there is a strong authorization for one action and a weak authorization for the opposite action, the strong

authorization always overrides the weak one. When there is a strong authorization for one action and another strong authorization for the opposite action an inconsistency is said to have occurred. This model was used for access control policies for databases, so having only authorization policies was sufficient. This may not be flexible enough, however, for other types of applications. There are other languages, which provide further types of policies, such as a policy enforcing that an action must be done.

Koch, Kress, and Krämer [17] proposed a language where policy can have three possible modality values: obligation (having to do something), permission (permitted to do something), and prohibition (not permitted to do something). These three modality values take their base from standard deontic logic [27]. This seems to be a common approach that is used, with some variation, in other models [1][18][19]. The authorization policies used by Jajodia, Samarati, and Subrahmanian are similar to policies in Koch, Kress, and Krämer's language using the permission and prohibition modality values.

Another language proposed is that of Lupu and Sloman [4]. In their language they use only two modalities: authorization and obligation. They also allow positive and negative modifiers to be used with these modalities, so that rules can have positive or negative authorization (can or cannot do), or positive or negative obligation (must or must not do).

Lupu and Sloman state that their notion of authorization is independent of their concept of obligation, and so their approach is not based on deontic logic as Koch, Krell, and Krämer's is. According to Lupu and Sloman, whether it is permissible to do something or not has no bearing on whether or not that action must be done. With Lupu and Sloman's representation, both of these states can be represented without affecting the other. E.g., a person may have permission from the file system to delete all his or her files, but that person's boss may state that they must not do it. In this case the person has positive authorization to delete his files but also has negative obligation for the same action.

One of the main differences lies in where the modality is interpreted. For obligations the modality is interpreted at the subject, whereas authorizations tend to be interpreted at the target. This is because authorizations are designed to protect the target, so subjects cannot be trusted to obey the modality in this case. On the other hand, if there is an obligation such as "The subject must not disclose any information to the target until after the final submission date", the target may in fact want to receive the information contrary to the rule. Therefore in this case it is the subject which must interpret the modality in order to ensure that it is taken into account [4].

Cuppens and Saurel's language [20] provides each rule with a more powerful conditional argument. Instead of only specifying a requirement that must be met in order for the rule to fire, their language also provides the keywords *before*, *during*, and *after* for use in the conditional argument. This means that policy designers can specify that the policy should fire before, during, or after a certain condition has been met.

2.3.2. Detecting and Resolving Conflicts

Lupu and Sloman define two types of conflicts that can occur between policies given the above model: **modality conflicts** and **application specific conflicts.** According to their definition, a modality conflict arises when two policies with opposite modality refer to the same subject, actions, and targets. This can happen in three ways:

- The subjects are both obligated-to and obligated-not-to perform actions on the targets.
- The subjects are both authorized and forbidden to perform actions on the targets.
- The subjects are obligated but forbidden to perform actions on the targets.

Note that since Lupu and Sloman do not imply authorization with obligation, the third type of conlfict is possible [6].

Application specific conflicts occur when two rules contradict each other due to the context of the application. Examples of this are:

- conflict of priorities for resources
- conflict of duties
- conflict of interest
- actions that perform opposing tasks

For example, assume that one rule instructs user X to submit a budget to the organization he works for, and say that another rule instructs the user to approve all budgets in the system. From looking at the policies without any knowledge of the world there is no apparent conflict, but with common sense, we know that in most organizations the person who approves budgets cannot approve his own budget. This of course may or may not be relevant to a particular application. Therefore, these restrictions are unique to the application and the rules cannot be considered to conflict without some sort of extra application specific knowledge. This extra knowledge is entered in the form of meta-policies. The meta-policies describe assumptions the system must make and explains how the system's "world" works. A set of policies may be consistent in one application and inconsistent in another due to the difference in the applications' meta-policy or world rules [6].

Lupu and Sloman provide a method of checking policies as they are specified for conflicts. Although they acknowledge the need for run-time conflict checking, they do not discuss any methods which would be useful in performing this type of checking. Similarly, their method of static (specification time) policy validation involves examining the entire set of policies, or at very least all the policies found in one domain. They do not provide any method of incrementally checking a very small number of policies when a new policy is added or modified, to determine if the new set of policies is consistent [6].

Fraser and Badger [23] proposed a way to detect conflicts for new rules that are introduced at run-time. Their approach examines the new rules to see first if the new policy is well formed. If so the next test is to see if it conflicts with the rest of the rule

base, and if so, the new rule is rejected. The technique proposed, however, was for use with the Domain and Type Enforcement prototype kernel, and so was specific to DTE.

Howard, Lutfiyya, Katchabaw, and Bauer [22] provide a generic policy based architecture. Their approach is designed to allow dynamic modification of policies at run-time. They do not, however, discuss the notion of policies conflicting with each other in any way, and therefore do not provide any mechanisms to check for conflicts. They do provide a policy validation service in their architecture, but its purpose is to check if any managed objects violate any policy rules.

Chomicki, Lobo, and Naqvi [12] propose a way to resolve conflicts. Instead of detecting conflicts at specification time, they perform detection and resolution at run time. There are monitors in the system which detect actions that cannot occur together, and then decide whether to delete one of them, or delete one of them plus the other actions that were meant to execute along with it. They suggest obtaining priorities for each policy from the user in order to decide which actions should be canceled when conflicts arise. Since the detection and resolution is performed at run time, predicting which rules will conflict is not a problem. However, there may be some rules which by examining them at specification time, we know will always conflict. If these conflicts were detected at specification time, the policy administrator could modify the rules so that this conflict is avoided at run time. Their approach offers no specification time conflict detection, which could eliminate some of the automatic conflict resolution needed.

Lupu and Sloman also propose a few ways to resolve conflicts once they are detected. The first is to always give priority to negative policies. For example if one policy gives positive authorization for an action and another policy gives negative authorization for the same action, the policy giving negative authorization overrides the positive one. The second method of resolution they suggest is to give each policy an explicit priority value. This can then be used to give rules precedence in the event of a conflict. The third method they suggest is to examine the distance in the class hierarchy

of the object being managed from the policy managing it. For example, let us say one policy is from the person class and another conflicting policy is from the human resources employee class which is a subclass of the employee class, which is a subclass of the person class. In this case the latter rule takes precedence, since the managed object, the human resources employee, is closer to that subclass than the general person class in the class hierarchy. The final method suggested is similar to the above method, but it is based on the domain structure, as opposed to the class hierarchy. A policy applying to a subdomain takes precedence over a policy applying to an ancestor domain because it is more specialized and targets fewer objects [6].

2.3.3. Policy Execution

Moore, Ellesson, Strassner, and Westerinen [26] state that a declarative approach to policy systems is better suited than a procedural one. They do not rule out the possibility that a policy framework can be successfully implemented using a procedural approach, but they think it would not be as natural an adaptation. This allows various implementations to use different algorithms for the testing of conditions and the sequence of action execution. The declarative rules state only the relationships between attributes to be tested and the actions to be executed. The opinion that a declarative approach is more suited to the problem of policy systems appears to be a common one, as many other researchers propose policy frameworks in the same manner [4][17][12].

2.3.4. Policy Implementations

2.3.4.1. IETF's Policy Framework

The Internet Engineering Task Force (IETF) has been developing a policy framework that is vendor independent, interoperable, and scalable. Although they deal mainly with the application of network management, they acknowledge that their policy framework could easily be adapted for other uses as well.

There are three main requirements according to the IETF that a policy framework must meet. First there must be a repository in order to store and retrieve policies. By storing all the policies in one location, the policies can be re-used by several devices. This also helps maintain consistency throughout the policies [24].

Second, there must be a common format to enter the policies. With a common policy format that is vendor independent, the same policies can be compatible with many different applications. For example, a management application may need to manage several similar devices, such as printers, that come from different vendors. If there is a standard policy format being used, then the same policy can be used to manage all the devices. This prevents the policies for each device from having inconsistencies as they might if the same policy was entered for each device in a slightly different format [24].

Finally, a policy framework needs a way to communicate the policy beyond the repository. This is necessary for scalability [24]. If there are a large number of nodes accessing the repository for policies, a bottleneck will occur. There needs to be some way to distribute the policies so that the repository is not overwhelmed with requests for policy information.

In addition to these three main requirements, they have identified a few other issues which should be examined: security, timely delivery of policies, and dealing with mobile devices [24]. We will not dwell on these as they are beyond the scope of this thesis.

IETF's approach differs from management tools such as SNMP [25] in that they propose some degree of automation of management. When the system already contains information that is needed to manage the system, the policy can retrieve it automatically without needing an administrator to re-enter it. This helps achieve the goal of managing the network with the minimum amount of human intervention [24].

In addition to the requirements of the framework itself, the IETF have identified certain requirements that must be met by a policy specification language (PSL):

- a way to state which actions should be taken by the policy-managed entity
- a way to state which conditions must be met in order to take the above actions
- a way to state what the policy is to control (policy subject)
- getting status information about the policy subject
- describing properties of the policy subject
- describing the relationship of multiple policy actions (e.g., one is dependent on the completion another, etc.)
- security information

In IETF's model, policy is represented by a set of policy rules. Each rule is made up of conditions and actions. The rules can be grouped into policy groups. These groups can be nested, and therefore give a way to represent policy hierarchy. A policy group is limited to an aggregation of policy rules or other policy groups. A policy group cannot be both. Policies do not need to be contained in a policy group. A one-rule policy that is not part of a group is called a stand-alone policy [26].

A rule's set of conditions can either consist of an ANDed set of conditions (Disjunctive Normal Form) or an ORed set of conditions (Conjunctive Normal Form). When a rule's set of conditions evaluate to TRUE, the actions associated with that rule are executed. The rule can specify to execute the actions in a particular order, state that the order is mandatory or merely recommended, or state that there is no recommended order of execution at all [26].

IETF also allows their rules to have a priority value. Two policies may apply at the same time but may state different things. This is where priority values come in handy. For example, one policy may state that everyone in the department gets 100 pages of printer quota, but another policy may state that Dr. Jones gets 500 pages of printer quota. Since both rules apply, there needs to be some way to distinguish which one should take priority. In this case the rule stating that Dr. Jones gets 500 pages of

printer quota should have a higher priority to indicate that it takes precedence over the other rule [26].

IETF's policy model was designed with a declarative, non-procedural approach. In declarative languages, rules and functions are declared and then executed according to an execution algorithm. The rules are not specified to execute in a particular order. This contrasts to a procedural approach where the order of execution of various functions is strictly stated. Although their model can be implemented using a procedural language, it is better suited to declarative languages, since the condition testing sequence and action execution sequence are not specified in the policy repository [26].

IETF's framework uses roles which policies are associated with, instead of assigning policies to specific resources. The policies are associated with the roles and the resources are also associated with various roles. If the role of a resource changes, there is no need to change its policies. All that needs to be done is to reassign the resource to a different role which has a different set of policies associated with it. Also if the policies governing a certain role are changed, then all resources associated with that role will take on the new behaviour. Compare this to having to change the policies for each of the individual resources, possibly introducing inconsistencies among the various policies due to human error [26].

Roles also help in avoiding certain conflicts. By specifying certain policies for one role and other policies for another role, all the policies can coexist even if some of the policies conflict. As long as all the rules in each rule are consistent, and the two roles cannot be active at the same time, there will not be a conflict. This can make specifying policies easier, since not all policies need to be consistent with each other when there is no chance they will be used at the same time [26].

2.3.4.2. Lupu and Sloman's Policy Framework

Lupu and Sloman proposed a management framework that included a policy service. Their framework consists of:

- A domain service to group target objects.
- A policy service to allow policies to be specified, stored, and distributed to agents which will interpret them.
- A monitoring service to monitor managed objects for occurring events and to notify interested agents [4].

A domain is used to group objects based on certain attributes, such as geographical location, object type, functionality, etc. or simply even for the user's convenience. In Lupu and Sloman's framework, domains contain multiple objects and can also contain other domains, called subdomains. Domains can also overlap with each other. This means that an object X can be a member of two or more domains. Path names are used to identify domains. For example, domain C which is a subdomain of B which is a subdomain of A can be referred to as /A/B/C [4].

The policy service is the component of the system that allows a user to specify the behaviour of their policy. The service provides tools to the user to define policies. All the policies are also stored in this component as objects. Policies can be members of domains. This allows the policies to have other authorization policies governing who can access and modify them [4].

A policy editor is used by an administrator to enter, modify, delete, enable, and disable policies in the policy system. The administrator checks for any conflicts and then changes the policies accordingly. Authorization policies are then distributed to target agents, and obligation policies are sent to subject agents [4]. Once the policies have been distributed to the appropriate targets and subjects, they may be enabled or disabled by the administrator without having to redistribute the policies [28]. There are manager agents which register themselves with the monitoring service. This allows the manager agents to receive any relevant events coming from the managed objects. If a manager agent

receives an event which triggers an obligation policy, the agent queries the domain service to determine which target objects are affected, and then performs the actions stated in the policy on the targets [4].

The policies in the system are represented by rules. The rules have an attribute specifying which events trigger the rule. However, there is also a rule attribute called a constraint. The constraint is a second-level condition that must be matched in order for the rule to fire. For example a rule may fire upon receiving event A but only between the hours of 9:00am and 5:00pm. This time period condition would be expressed in the constraint part of the rule. As with most rules there is also an action attribute stating which actions should be taken upon the rule firing [28].

Roles associate behaviour with a position as opposed to a particular individual. For example if Eric Jones is a manager, he has certain responsibilities based on that manager position. If he were to be promoted to senior manager, then his responsibilities would change. If we were to modify his policies to reflect his new position, we may introduce errors, or the responsibilities tied to his position may not be consistent with someone else's policies who is in the same position. It is much simpler to associate those responsibilities with the roles of manager, and senior manager themselves. Then the individual can be associated with the role. If the role changes, the individual can simply be associated with a new role and disassociated from the old one [6].

Lupu and Sloman also suggest that it is useful for each role to have its own context in which to operate. This prevents a user from playing multiple roles at once, and performing actions in one role which he is not permitted to but is permitted by another role. For example, if Role A permits an action and Role B prohibits it, then the behaviour would be determined by whichever role is active at the time [6]. A good example of this is a police officer. While on duty a police officer has certain privileges such as to drive faster than the speed limit, and certain restrictions such as not being able to consume alcohol. Off duty the police officer does not have all of the same privileges or

restrictions. His role of being a police officer is no longer active and so the privileges and restrictions associated with that role no longer affect him.

Roles are implemented using domains. The positions of the organization to be modeled can be represented by domains. A role is then considered to be the set of policies with a particular domain as the subject. A person, represented as an object, can be added to or removed from the domain, without changing the policies associated directly with that person, or the role the person takes on [4].

Roles often have relationships, which are ties between two roles. For example, a senior manager has a relationship with each of its managers. Lupu and Sloman also provide a way to associate these relationships to the roles using policies. This allows policies to express the relationship between roles in terms of rights and duties of both parties towards each other [6].

This framework could easily be generalized to serve as a general purpose policy application framework. Domains can still be used to group objects together, and roles can still be used to show relationships and responsibilities between objects.

2.4. Our Policy Model

In this thesis, we are attempting solve the problem of incremental policy validation regardless of what application the policy is being used for. We have created our own policy model, which takes some of its elements from work by other researchers. In particular policy specification and policy execution are taken straight from other work. We have based our conflict detection on other work, but have extended it to be more efficient and more complete. Also, we have introduced a new concept into our model: policy scope.

2.4.1. Policy Specification

Out of all the specification languages we examined, Lupu and Sloman's language appears to be the most flexible. It offers both positive and negative modifications of authorization and obligation policies, and allows policies to have priority values. We find that it meets all the needs for policy specification. Therefore we have chosen to use their policy specification language for our policy model. Although this thesis does not focus on policy entry, we assume that all policies entered into the system are done so in the form of Lupu and Sloman's policy specification language.

2.4.2. Detecting and Resolving Conflicts

Since we are using Lupu and Sloman's policy specification language, our policies can have the same type of modality and application specific conflicts indicated by Lupu and Sloman. Therefore our conflict detection methods will aim to find these types of conflicts. Lupu and Sloman provided ways to detect these conflicts by examining the policies at specification time. In this thesis we will provide a way to determine if a set of policies is consistent when adding a new rule to a pool of consistent rules. We also include some new types of conflicts. If we know that one policy firing will cause another policy to fire, we introduce new conflicts that we can detect. We call this type of conflict a trigger chaining conflict and we will discuss it further in Chapter 4.

2.4.3. Policy Execution

Our model will store policies in an internal policy format in order to perform conflict checking and other analysis on the policies. In order to execute the policies, they must be transformed from this internal representation to a format which can be executed. Although we are not concerned with how this is done, for the purposes of this thesis we assume that after conflict checking and analysis, the policies are transformed into rules that can be entered in a rule engine such as an expert system. There has already been a lot of research done in the field of rule engines and many rule engines are available on the market today.

2.4.4. Policy Scope

In our model we introduce the concept of policy scope. Up to this point it has been implicitly assumed that all policies apply to the entire system. This, however, may not be desirable. It may be beneficial to have some policies, which, rather than affect the system as a whole, only affect a well-defined subset of the system. In this case we call policies which do affect the whole system **global policies**, and policies which are local to a particular subsection of the system **local policies**. Take for example an organizational hierarchy. There may be an administrator who is authorized to set policy for the entire organization. In this case his policies would be global. On the other hand, there may be an individual user who wants to set a policy for how his e-mail should be delivered. In this case he should not be able to set the policy for everyone, only for himself. In this case it would be a local policy. Note that in some cases the policy could be global or local depending on the scope of the hierarchy being examined. For example if a manager is able to set a policy for all his subordinates, the policy would be considered global to the department, but it does not affect the entire organization. Therefore, when looking at the entire organization, the policy would be considered local.

An example of this would be a university department that has policies governing printers and how they are used. The department owns the printers so every member of the department uses the same policy for all the printers, therefore it can be considered a global policy. If a professor, however, purchases his own printer for use by his students, he may have his own custom policy he wants to enforce over his printer. In this case the printer policy for the private printer is not applicable to the whole system and is a local policy. Note that because global policies reach to the same nodes that are governed by local policies, local policies must have a higher priority value. Without this higher priority the local policy will never be taken into account since the global policy will be used all the time.

Similarly, the concept of consistency can be divided into two subdivisions: **globally consistent** and **locally consistent**. If a system is completely consistent, that is to say that every single one of its policies coexist with all other policies in the system

without causing conflicts, then we can say that the system is globally consistent. However, we may be able to predict that there may be little chance of a particular policy ever interacting with another policy. In this case it may be unnecessary to strive for global consistency. Instead it may be necessary to determine only that certain subsets of policies are consistent within themselves. If a subset's policies can coexist with all other policies in the subset without causing conflict, but may cause conflicts with policies not in the subset, we say that the subset is locally consistent and globally inconsistent. A policy that is globally consistent implies that it is locally consistent. Similarly a policy that is locally inconsistent implies that it is globally inconsistent.

Theorem 1:

Let S be the entire set of policies, and let there be a set of policies X such that $X \subseteq S$. Then we can say:

1.1. GloballyConsistent(S) $\Rightarrow$ LocallyConsistent(X)

1.2. LocallyConsistent(X) $\not\Rightarrow$ GloballyConsistent(S)

1.3. LocallyInconsistent(S) $\Rightarrow$ GloballyInconsistent(X) [Dual of 1.1]

1.4. GloballyInconsistent(X) $\not\Rightarrow$ LocallyInconsistent(S) [Dual of 1.2]

Proof:

1.1. If S is globally consistent then every policy in S can coexist with every other policy in S without causing any conflicts. Since all policies in X are policies in S, then every policy in X can coexist with every other policy in X without conflict. Therefore X is locally consistent.

1.2. Say we have a set of policies X such that every policy in X can coexist with every other policy in X without causing conflicts. Then X is consistent. Let us say that one policy in X says "Close door A at 1:00pm". Now say that S is made up of all the policies in X plus one more policy which says "Open door A at 1:00pm". Since this new policy conflicts with a policy in X, we must in this case

say that S in inconsistent. Therefore X being consistent does not imply that S is consistent.

1.3. If X is inconsistent then there are two or more policies in X that conflict with each other. Since S contains all the policies in X, it also contains the policies which conflict. Therefore S is also inconsistent.

1.4. Say that the set of policies S contains two policies: "Open door A at 1:00pm" and "Close door A at 1:00pm". These policies conflict and so S can be said to be inconsistent. Now say that we have X which is made up of only one of these policies. Since X contains only one policy, it cannot be inconsistent with any other policies. Therefore global inconsistency of S does not imply local inconsistency of S.

A local policy is not compared at every node when checking for global consistency, since the scope of local policies does not reach the entire set of nodes. Instead it is only examined for local consistency at the nodes it affects. If a local policy is inconsistent at any node it affects, then it is impossible for the system to be globally consistent. One possible way of resolving a global conflict is for the policy administrator to make the conflicting global policies each local to a particular node or set of nodes. If only one of the policies is made local then one of the policies is still global and there will still be a conflict in those selected nodes that are locally affected by the first policy. For example say that global policies A and B conflict. Now say that in order to fix the conflict, policy A is made local to node N1. Now there will be consistency on every node, except for N1, because B being global, it too affects N1. Therefore N1 is still locally inconsistent. Similarly, if policies A and B were both made to be local to N1, then N1 would of course be locally inconsistent, thereby not changing the status of the system from being globally inconsistent.

Since we know that a system containing nodes that are locally inconsistent is globally inconsistent, we can use this information when checking for the consistency of the system. If one node is found to be locally inconsistent, we can halt validation and

declare that the entire system is globally inconsistent, without having to check the remaining nodes.

3. Policy as Decision Tables

Decision tables are a formal way to specify the behaviour of a software system. It groups conditions with actions that will be performed upon the conditions being met. These groups of conditions and actions are called rules and stored as a column in the table. Decision tables offer an easy way to analyze whether a set of rules is complete and consistent. This will benefit us by allowing policy rules to be checked for completeness, as discussed in section 3.2.1.

3.1. Introduction to Decision Tables

A decision table is made up of four parts. The top left quadrant of a decision table is called the condition stub. The condition stub lists all the possible conditions that can occur in the system. It can be listed as asking a binary question (yes/no, do/don't do, etc.) or something more general such as colour or value. The bottom left quadrant is called the action stub. The action stub lists all the possible actions that can be performed by the system. As with the condition stub, the items in the action stub can be written to expect a binary answer (yes/no for example), or to expect a value such as "Sell car for", which expects a dollar value [29].

Michael Montalbano defines the condition stub as a vector of conditions and an action stub as a vector of actions. He then defines a condition as a set of condition values and an action as a set of action values [29]. I have extended these last two definitions in order to make them more complete. According to Montalbano, a condition is a set of condition values. Therefore a condition called *Cost* whose value set is the set of integers and another condition called *Sold for* whose value set is also the set of integers would be identical, since their values are both from the set of integers. I find that in order for the condition to have any meaning, it must be given some sort of context. The clue for that context is often contained in the name of the condition. Therefore when I refer to the term *condition* I will be referring to a set of condition values and the condition name

associated with that set. Similarly, when I refer to an *action* I will be referring to a set of action values and the action name associated with that set. In Figure 1 we can see that there is a condition name called "All Reports Submitted" which is associated with the values *yes* and *no*. We consider all of this information to make up a condition called "All Reports Submitted".

The top right quadrant of a decision table is made up of condition entries. The condition entries hold the answers to the questions asked in the condition stub. The bottom right quadrant is made up of action entries. The action entries hold the answers to the action questions asked in the action stub. Each column of answers in the table is called a rule [30]. The set of condition values that make up a rule is called a **condition set.** The set of action values that make up a rule is called an **action set.** Each rule indicates which conditions must be met (condition set) in order for the actions indicated in the rule (action set) to be performed [29]. When the events and states in the system (called a transaction), match the conditions specified in a rule, the actions in that rule are executed [30].

One thing to note is that a rule can have an explicit answer to a condition listed in the stub, such as *yes*, *no*, *5*, *after 6:00pm*, etc., or can be left blank for that value. A blank value is referred to as a don't-care-entry. Don't-care-entries imply the entire set of possible values for the corresponding condition and so match any value when tested. This is important when we discuss testing the consistency of the table later on [29].

Note that both condition sets and action sets, although given the name sets, are in fact represented as vectors. The order of the values in a condition set, for example, is important, and will have an entirely different meaning if changed around. For example, suppose we have a condition stub containing two questions: *number of students*, and *professors available to teach.* Now say we have a condition set of 1000, 20. This means that there are 1000 students and 20 professors available to teach. This gives us a student to teacher ratio of 50:1. If we mix up the condition set, however, we now incorrectly see that we have 20 students and 1000 available professors. The values by themselves are

useless. We need to know the order in which they appear in order to abstract some meaning from the values. Therefore, our condition sets and action sets are in fact, vectors. Note that to some, the strict definition of a vector is an ordered set of scalars. Montalbano has decided for simplicity to overlook this restriction and use vector as meaning an ordered set of some type of element, be it rules, conditions, actions, etc. All statements in this thesis referring to vectors use Montalbano's wider definition of vectors [29].

There are two types of decision tables. Tables with entry sections expecting to have binary answers to the questions in the stubs such as *yes/no*, *do/don't do*, etc. are called limited-entry decision tables. On the other hand if the answers are expected to be a little more general such as *red*, *5*, *after 6:00pm*, etc., then the table is called an extended-entry decision table. Tables which contain both types of answers are called mixed-entry tables [29]. Note that wild-card or don't-care entries represent two or more values implicitly, therefore tables which contain wild-card entries are considered extended-entry tables, even if all other values are only yes or no [30].

An example decision table giving rules on how to set up a thesis defence is shown below in Figure 1.

	Rule1	**Rule2**	**Rule3**	**Rule4**
All Reports Submitted	YES	NO	YES	NO
Room Booked	YES	NO	YES	
Chairman Appointed	YES	NO	NO	YES
Book a Room		√		√
Announce the Exam	√	√	√	√
Appoint Chair		√	√	
Find Examiners		√		√

Figure 1 - An Example Decision Table

Notice that there are no rules with duplicate condition sets. Theoretically there could be, but there is no need for their separate existence because they could be merged. Suppose we were to have two rules with identical conditions indicated, but different actions specified. Then we could simply merge the two columns such that there is one column with the union of the actions listed in the two old columns. To the contrary, there could be two columns with identical actions but different conditions. This is because it is more difficult to merge columns based on differing conditions, although it is possible.

There are two types of rules in a decision table: simple rules and composite rules. Simple rules have at most one simple value listed in the condition entry. An example of this would be a rule with one condition, *Time of departure* with a value of *5:00pm.* A composite rule (sometimes called complex rule) has one or more entries with two or more values. An example of this would be a rule with the same one condition, *Time of departure*, with a value of *(5:00pm ∨ 9:00am)* [29]. Note that since a don't-care-entry represents two or more values, a rule containing a don't-care-entry is considered composite [31].

3.2. Proposal for Using Decision Tables as an Internal Policy Represenation

There are two things that decision tables make easy: detecting conflicts between rules in the decision table, and checking to see if the table is complete. Since it is important to check our policies for conflicts and may be important to check them for completeness, the format of decision tables to store policies seems appropriate. We will examine both completeness checking and conflict checking in decision tables and discuss how they can be used with policies. In their standard format, decision tables are not sufficient to store policies. Modifications will have to be made in order to store all the policy information. Furthermore, we will discuss the problem of converting policies from the internal decision table representation to that of executable code.

Note that in this section I will use example condition sets of rules to demonstrate certain ideas. The notation I use is the following:

RuleName: {(value1a, value1b, value1c), ¬(value2a), (*)}

Where:

RuleName is the name of the rule.

The curly braces { and } show the end of the rules values.

Values for a particular attribute are grouped in parentheses and separated by commas. Each set of parentheses are also separated by commas.

The value *(*)* denotes the don't-care-value.

The value *¬(value2a)* denotes *NOT (value2a)* i.e., *(*) – (value2a).*

Since the actions were not important for our present discussion, I have omitted them in this notation. This notation only shows the condition part of the rule.

3.2.1. Checking for Completeness

When writing policies, it is often important to know whether or not a policy set will cover any possible situation the system will ever come across. A policy set which does this is said to be **complete**. Completeness is something that is often difficult to check when a policy is written at a high level, especially when there are many possible conditions to be met. Decision tables prove themselves very useful in this area.

In order to check if a policy set is complete, we must first know how many simple rules are theoretically possible within our table. Recall that a simple rule has exactly one condition value corresponding to each condition in the stub. Therefore the number of possible simple rules in our table is equal to the number of possible values for the first condition multiplied by the number of possible values for the second condition multiplied by the number of possible values for the third condition, etc. [29] For example, let us say that I have policy describing costs of used cars. I have three conditions in my table: *Colour*, *Year*, and *Make*. Let us say that *Colour* can be one of four values: *black*, *green*, *red*, *blue*; *Year* can be one of twenty values: *1981* through *2000*; and *Make* can be one of

three values: *Honda, Volvo, Toyota.* The number of simple rules possible in this table is equal to the number of possible conditions that can occur in the system, which is equal to 4 (number of possible colours) * 20 (number of possible years) * 3 (number of possible makes) = 240.

Once it is known how many simple rules the table can represent, the next step is to find out how many simple rules the table actually represents. If the table only contains simple rules, this is simply a matter of counting how many rules are in the table. However, often times tables will contain composite rules. Composite rules represent multiple simple rules, and so it is necessary to figure out how many simple rules each composite rule represents. In order to do this, we multiply the number of values entered for each condition in the rule [29]. For example, if I have a rule with the condition values

{(red, black), (1995, 1996, 1997, 1998), (*)}

then the number of simple rules this composite rule represents is 2 (number of values for first condition) * 4 (number of values for second condition) * 3 (number of values for third condition) = 24. Note that since the last value was a don't-care value, it in effect represents all possible values for that condition.

This is not completely correct, however. Let us say that we have two rules with condition sets:

{(red, black) (1995, 1996, 1997, 1998), (*)} and

{(*), ¬(1997, 1998), (*)}

According to the above discussion, the first rule represents *2 * 4 * 3* = 24 simple rules, and the second rule represents *4 * 18 * 3* = 216 simple rules. Since our table can represent *4 * 20 * 3* = 240 rules and we are representing *24 + 216* = 240 rules, we might be tempted to say that our policy set is complete. This is not correct, however. Note that the condition set *{(red), (1995), (Honda)}* can be represented by both rules. Note also that the condition set *{(blue), (1997), (Honda)}* is not represented by our policy set. Our policy set contains overlapping conditions, and therefore some of them have been counted twice. Two rules which do not overlap are said to be **disjoint.** In order to count

the number of simple rules represented by our policy, all of our rules must be disjoint. We must then create disjoint rules from the rules in our policy set [29].

First, a set of overlap-rules are created. Overlap-rules are rules generated to express the overlap of rules in the table.

Definition 3.2.1.1:

An overlap-rule R_O is created from two rules R_1 and R_2, such that if R_1 has condition set $X:\{x_1, x_2, \ldots, x_n\}$ and R_2 has condition set $Y:\{y_1, y_2, \ldots, y_m\}$, R_O will have condition set $Z:\{z_1, z_2, \ldots, z_p\}$ where $Z = X \cap Y$.

For example, using the two rules above, the overlap-rule would be *{(red, black), (1995, 1996), (*)}*. Using our technique of counting how many simple rules a rule represents, we can see that our overlap rule represents twelve simple rules.

What we need to do is replace one of the two rules that overlap with one or more rules that represent the old rule without the overlap. To do this we suggest the following procedure:

1) Find the rule representing the overlap in the two rules. Call this rule R_O.

   For each slot in R_O:

   2) Call the position of that slot P. If the slot contains only one value, is the null value, or is the don't-care value, do nothing. Otherwise create a rule. The new rule's values are R_O.
   3) The value in slot P of the new rule is then replaced by:
      the intersection of the value in slot P of R_2 and the complement of the value in slot P of R_O.

4) Eliminate the original R_2.

Example:

We have the two rules:

R_1: {(red, black), (1995, 1996, 1997, 1998), (*)} and

R_2: *{(*), ¬(1997, 1998), (*)}*

First we find the overlap rule R_O, which in this case is:

R_O: *{(red, black), (1995, 1996), (*)}*

We look at the first slot in R_O which is (red, black). It is more than one value so we create a new rule R_{2a} which takes its values from R_O.

{(red, black), (1995, 1996), (*)}

Now we replace the value in the first slot with the intersection of the first slot from R_2 and the complement of the first slot in R_O.

$$R_2^1 \cap \neg R_O^1 = (*) \cap \neg(red, black)$$
$$= (*) \cap (blue, green)$$
$$= (blue, green)$$

Now we have:

R_{2a}: *{(blue, green), (1995, 1996), (*)}*

Next we examine the second slot of R_O. It is *(1995, 1996)*, which again contains more than one value so we create another rule identical to R_O.

{(red, black), (1995, 1996), (*)}

This time we replace the second slot's value of the new rule with the intersection of the second slot in R_2 and the complement of the second slot in R_O.

$$R_2^2 \cap \neg R_O^2 = \neg(1997, 1998) \cap \neg(1995, 1996)$$
$$= \neg(1995, 1996, 1997, 1998)$$

Now we have:

R_{2b}: *{(red, black), ¬(1995, 1996, 1997, 1998), (*)}*

Next we look at the third slot of R_O. It is (*), which is the don't-care value so we do nothing. Finally we eliminate the original R_2.

Now as our rules we have:

R_1: *{(red, black), (1995, 1996, 1997, 1998), (*)} and*

R_{2a}: *{(blue, green), (1995, 1996), (*)}*

R_{2b}: *{(red, black), ¬(1995, 1996, 1997, 1998), (*)}*

From our earlier calculations we determined that originally R_1 had a simple rule count of 24, R_2 had a simple rule count of 216, and that there were 12 rules which overlapped. Therefore we were representing 228 rules.

Our new count is:

R_1) *2 * 4 * 3 = 24*

R_{2a}) *2 * 18 * 3 =108*

R_{2b}) *2 * 16 * 3 = 96*

Total = 24 + 108 + 96 = 228

Therefore our new rules do represent the same number of rules as the old one. If we now find the overlap rules we get:

$R_{O1,2a}$: *{∅, (1995, 1996), (*)}*

$R_{O1,2b}$: *{(red, black), ∅, (*)}*

$R_{O2a,2b}$: *{∅, ¬(1997, 1998), (*)}*

Notice that every single rule has a null value somewhere, meaning that none of our new rules overlap with each other. In this example, with only two rules, it was not really necessary to remove the redundancy between the rules, since we could calculate the overlap simply by seeing how many simple rules the overlap rule represented. In a larger system, however, it may be necessary. Let us say we had three rules A, B, and C. A represents 100 rules, B represents 100 rules, and C represents 100 rules. Let's say the entire system can represent 250 rules. We may know that the overlap-rule for A ∩ B represents 50 rules, the one for B ∩ C represents 50 rules, and the one for A ∩ C

represents 50 rules. But it is complicated to determine which of these overlap-rules also overlap. For example we may have over-counted the number of simple rules that are represented by the overlap rules if $A \cap B \cap C$ is not empty. It is much easier to transform the policy set into one that does not overlap at all and then count the equivalent simple rules.

It is not always desirable to have a rule set that covers every single possible combination of conditions. In that case, checking for completeness will always return a negative result. If there is a standard completeness checking mechanism in the system that checks every single table for completeness, an else rule with a null action set can be added to cover any rules not covered by the rest of the table, thereby making the table artificially complete.

3.2.2. Consistency Checking

Because of the structure of decision tables, it is easy to see particular combinations of conditions and actions when looking at one. [29] This would certainly make it easier for humans to analyze the rules and determine if they are consistent. If a new rule is added to the decision table or an existing rule is modified, first find all conditions that are indicated by the rule in question. Next go along the row for each condition and find all columns that also indicate that condition. (By indicated we mean they have some value other than the don't-care-value.) Each column that does is a rule that could potentially be triggered at the same time as the updated rule. Out of those rules, any one that has an action that conflicts with an action in the original rule can be said to potentially conflict with the original rule.

However, for our purposes, humans will not be used to check for consistency of the rules. What interests us is whether this structure eases consistency checking in some way the computer doing the checking can use. There has been research on how computer programs can best perform consistency checking using computer algorithms, however, it

may first be useful to see if this type of conflict checking will solve the problem of detecting policy conflicts.

As mentioned in the previous chapter, policy is concerned with two overall types of inconsistency: modality conflicts and application specific conflicts. In classic decision table theory, the concern is with two other types of conflicts: intercolumn inconsistency and intracolumn inconsistency.

Intercolumn inconsistency occurs when two rules in the decision table have overlapping conditions. What happens is that when the conditions for the two rules are matched at run time, the system does not know which rule it should act upon [29]. This is also referred to as having an ambiguous table. When the two rules with overlapping conditions have the same actions, the ambiguity is said to be apparent. If the rules specify different actions to be performed, the ambiguity is said to be real [32]. In decision table terminology, having a real ambiguity means the table is inconsistent [35].

It has been said that ambiguities must be detected and removed from decision tables [31]. However, King and Johnson have argued that this is an unnecessary restriction to those applications which require only one action for a set of conditions. They argue that some applications may require that two or more actions be performed when one set of conditions is matched, and that this is perfectly acceptable. Rules that permit this behaviour are called multi-action rules [32]. Lew also used these kinds of rules by providing a vector action set as opposed to a single action for each rule. He also allowed all actions to execute in the case that two rules with the same conditions specified different actions [35].

For our purposes, it is perfectly acceptable to have more than one action executed upon a set of conditions being matched. Not only can this be done if we have one rule specifying multiple actions, but also if two separate rules happen to be triggered at the same time. Therefore, detecting the presence or lack of ambiguities in our table will not

lead to concluding whether or not our set of policies is consistent, since our notion of consistency is different from that associated with single-action-rule decision tables.

Intracolumn inconsistency refers to a rule having conflict within itself [29]. For example, say one condition asks what year their driver's license was obtained and another asks for the year of birth. Based on driving laws, we know that people under a certain age cannot have driver's licenses. In that case if there is a rule which has a value for year of birth that does not allow a license and there is a non-null value for the year the driver's license issuance, then we have an inconsistency. For decision tables this often seems to be a conflict between two conditions, but it seems logical that this can be extended to cover an action combination that does not make sense according to the "world" the system operates in. For example, if one action says to delete a file, and another action says to modify the file, we would have an inconsistency. Intracolumn inconsistency checking, therefore, seems to map quite well to the notion of application specific conflicts in policy. How this can be done is not discussed in this thesis and is left for future work.

3.2.3. Storing Policies as Decision Tables

Now that we have seen the benefits of representing policy as decision tables, it is important to know how to transform policies represented in another form to the decision table representation.

For the purpose of storing policy rules in decision tables, we will consider an event trigger to be a specialized form of condition. In this way, we will not need to modify our decision tables to be able to specify which events will trigger particular rules. We can simply make a condition indicating that if the event occurs that the condition has been met. This solves the problem of representing both events and conditions in the table which is only meant to represent conditions. For the remainder of this section the word condition in a decision table will refer to our expanded notion of condition which is either

a condition from the rule or an event from the rule. For example in Figure 2, *Room Booked* could refer to either a condition or event.

Since all policy rules may at one time fit into one table, the table must have a row for each possible condition statement that appears in the rules. This does not act as a restriction on the number of condition statements the system can represent. Let us say that we are adding a new rule which has a condition statement never before seen in the system. All that needs to be done is to add a row to the table with the new condition statement. Then the new column can be built for the rule in the normal fashion. Since the other rules didn't care about this condition, we need not modify the other rules. They will already have a blank entry for that condition indicating a don't-care-value.

Similarly the table must have a row for each possible action that rules can perform. As with the condition statements, if a new action appears in a rule which the system has not listed, a new row can be built on demand for that action. Notice that in Figure 2, that if Rule 5 were added to the system and the action *Defend* did not appear in the table, that it could be added to the table. The other rules would get a don't-care-entry in that row, which would not affect them in any way.

Since actions and condition statements can be added to the table as needed, there is no need to examine the rules, and construct the rows all at once before entering the rules. The table can be built incrementally, adding rows as needed. For example, if our system can produce 10 events but we are unaware of how many events are possible at the time of building the table, there will be no problem. The table is created using the first rule which let us say indicates two events, so the table now contains two events. Now we take the second rule which let us suppose indicates two different events. These new events are added to the table as the second rule is added to the table and the process is continued, adding new conditions and actions as the rules containing them are processed.

Our table is missing two important attributes that are represented in the PSL we are using, however: the notion of modality, and the order of execution. In Sloman's PSL,

each rule has a single modality value. Therefore, we extend the decision table structure such that each column in the entries section has a value representing the modality. The question is where should this value be located. Modality is not something that affects whether or not a rule should be triggered, so it really shouldn't appear in the condition part of a rule. However, it is not really an action either, but a modifier on the actions. It determines whether the actions should be executed, permitted to execute, prohibited to execute, or must not execute. In that case, we should really have another horizontal divider in our table. This third section will hold exactly one row representing the modality values for all the rules. To fit with the rest of the table, the modality section of the table should have a modality stub. Since there is only one row, the item in the stub will never change, and will in fact have no real purpose except to be consistent with the rest of the table. We'll call this stub item ***Modality***. The value in the entries section corresponding to this stub can be exactly one of the values used in Sloman's PSL for each rule: A+, A-, O+, or O-.

If two rules are triggered at the same time, we may wish there to be priority values associated with each rule as suggested by Sloman, in order to determine which rule should be fired first. Therefore, we need another modification to the decision table in order to indicate this priority value. Again, this value is neither a condition upon which we can determine if the rule should fire, nor an action to perform. Therefore we should create another section in our decision table for this purpose. As with the *Modality* row, we will create another row with the stub called ***Priority***. The entries in the *Priority* can be varied depending on the sort of priority used in the policy specification language. It could be numeric values, strings such as HIGH, MEDIUM, LOW, or some other form of priority indicator. The table, being only an intermediate form for the policy to be stored and analyzed, does not need to restrict the values it can accept.

The order of appearance of actions in a decision table rule does not normally affect the order of execution of these actions. If it did, then a particular action X would always be performed before another action Y because actions in the same table always appear in the same order. This may be contrary to what is desired for the behaviour of

the rules. Therefore, we modify the action entry of the decision table such that it gives us a way to represent the desired sequence of execution of the actions. The table should allow each action to have a corresponding value representing when the action is to be executed compared to the other actions. There are two ways this can be done: absolute ordering and priority ordering. Using absolute ordering we can specify that a particular action will be the first to execute, another action will be the second to execute, etc. Using relative ordering, we can specify execution priorities for each action. That way, some actions could have the same priority, when it is not important which is executed first. Then, if there are multiple actions of the same priority, there is no need to wait for one if it cannot execute right away. The other actions of equal priority can be executed in the mean time, and waiting will resume if the remaining action still cannot execute.

	Rule1	Rule2	Rule3	Rule4	Rule5
Modality	A+	A+	A+	A+	A-
Priority	MEDIUM	MEDIUM	MEDIUM	HIGH	MEDIUM
All Reports Submitted	YES	NO	YES	NO	NO
Room Booked	YES	NO	NO	YES	
Chairman Appointed	YES	NO	YES		
Book a Room		3	2	1	
Announce the Exam	√	1		2	
Appoint Chair		4			
Find Examiners		2	1	3	
Defend					√

Figure 2 - An Example of our Modified Decision Table Format

Figure 2 shows an example of the modified decision table we have just described, with the new and modified areas shaded. Notice that in Rule5 there is a priority value associated with the rule, but no priority value given in the action entry. If there is no

action priority value given, the system can determine in which order to perform the actions.

Our next concern is how the policies should be grouped. A policy rule is stored in a decision table as a rule in the table. How to store entire policies or policy sets is not as obvious a choice. A single decision table could be used to store the rules of one single policy, all the policies in a particular policy set, or all the policies in the entire system. There is really no restriction in this respect. However, large decision tables could become unwieldy in terms of memory usage, thereby increasing the time needed to update the table or analyze it for consistency [33]. Therefore, it may be beneficial to store each policy as a separate decision table. This is also advantageous because in order to check for local consistency, often only one policy will need to be compared with itself, so no unnecessary policies will have to be examined if all the rules related to the policy are contained within that decision table. On the other hand, storing entire sets of policies (such as all the policies of a university department) in a single decision table can help to organize policies by keeping related policies together.

In the case where multiple policies are needed in order to check for consistency, two or more tables can be temporarily merged into one larger table for the purpose of analysis. Research has been done on how to best merge decision tables into larger tables in order to optimize performance times, such as the work of Shwayder [33]. In our case, merging the tables would be a similar process, but for a different reason.

Now that our table contains all the necessary attributes needed to represent our policies, we must figure out how to convert them from their policy definition language, into our table.

3.2.4. Conversion of Decision Tables to Code

We now have our information in a table where it can be analyzed for completeness, correctness, and consistency. Once this has been done, the decision tables

must be transformed into some form that can be executed. There have been several algorithms devised on how to translate the information given in decision tables into an executable code form [34],[35],[36],[37]. Generally these algorithms translate the contents of the decision table into a flowchart or execution tree.

However, with declarative languages such as CLIPS [38], JESS [39], and PROLOG [40], we could simply form rules in the desired language directly from the rules in the decision table. This would also take care of the problem of deciding exactly how the rules should be executed. Rule engines have been developed over many years with algorithms designed for the execution of declarative rules. In that case, all we need is a translation from the policy terms in our decision table rules to actual executable terms in the declarative language.

Vanthienen and Wets [41] have discussed how easy it is to convert decision tables into rules that are executable by an expert system shell. For example we could use column based translation. This involves creating one rule in the expert system shell for every column or rule that we have in the decision table. This is the most obvious translation method, although they also discuss two other translation methods into expert system rules: entry based translation and row based translation. Entry based translation is where each entry in the table is turned into a rule in the expert system shell. Row based translation creates a rule in the expert system shell for every action indicated in the decision table. Therefore the problem of converting decision tables to executable rules has already been solved, and we can benefit from these research results.

3.3. Summary

The power of decision tables is the ability to check its rules for completeness and to check for consistency. Since rules from decision tables can easily be adapted from policy rules, the decision table format seems like a natural method of storing policies in a declarative manner. Although the consistency checking normally performed on decision tables does not map perfectly to the problem of consistency checking between policies,

they do serve as a helpful tool in performing completeness checking. With rule engines available, all that is required to make policies executable is to take the decision table rules and translate them to the expert system shell language.

4. Incremental Validation

4.1. Why is Incremental Validation Needed

Clearly stated, the problem of incremental policy validation is this: Assume that there is a pool of rules R. Let us say for now there are a thousand of them. Assume that these rules have been validated somehow to check for conflicts, and R has been determined to be consistent – i.e., there are no conflicts within R. Suppose a change, ΔR, is made to R, such that the number of rules that have changes are much smaller than the total number of rules in R. We want to see if the set of rules R is still consistent without having to revalidate the entire set. We are aiming to revalidate R with significantly less effort than is needed to revalidate the entire set of rules. In particular we will show that this can be done in linear time.

4.2. Simple Incremental Conflict Detection

Let us assume that our system does not have all the properties of a distributed system, in particular the property that events multiple events can occur at the same time. We assume that events cannot happen concurrently. One event must occur before or after another, even if they are separated by only an extremely small time interval.

Then, two rules can be triggered at the same time if and only if they contain the same triggering event in their conditional statements. Furthermore, the common event between them must be the final event to occur out of all the events listed in the conditional statements for both rules. This is because we assume that two events cannot occur simultaneously. For example: if R_1 fires upon events A, B, and C occurring, and R_2 fires upon events A, D, and E occurring, then A must be the event that occurs last if these two rules are to conflict. If events A, B, and D are fired, then if event C or E occurs, then one of the rules will be triggered. Since both events cannot occur at the

same time, only one of the rules will be triggered at a time. The only way the two rules can be triggered at the same time is if events B, C, D, and E were fired, and then event A is fired last. Note that in the condition section of our decision table we are including both the event triggers and the conditions obtained from the policy specification language. Therefore for the remainder of this chapter, when we refer to a condition in the decision table, we are referring to either a policy's condition or a policy's event. There will be no difference between the two in our conflict detection algorithm.

Theorem 1:

Two rules can be triggered at the same time if and only if the condition entries for both rules contain at least one common condition.

Corollary 1:

Out of all the conditions in the condition entries for both rules, one of the common conditions must occur last in order for both rules to be triggered at the same time.

Theorem 2:

If a set of rules S is the union of two sets of rules S_1 and S_2, where S_1 is consistent, and S_2 is inconsistent, then S will be inconsistent.

Proof:

If S_2 is inconsistent, that means that S_2 contains two or more rules such that they conflict with each other. Since S is the union of S_1 and S_2, S will also contain this set of conflicting rules. Therefore S is inconsistent.

Corollary 2:

Any set of rules containing a rule which is inconsistent with itself, is an inconsistent set.

Example: if rule R_1 states that it should lock door A and open door A, this is inconsistent. Therefore the entire set of rules S in which R_1 is contained is inconsistent.

Theorem 3:

If the set of rules S is the union of the sets of rules S_1 and S_2, where S_1 is consistent, and S_2 is also consistent, then there is no conclusion on whether S is consistent or not. It could be either consistent or inconsistent.

Proof:

Say S_1 contains one rule R_1: on event X open window A.

Say S_2 contains one rule R_2: on event X close window A.

It is quite obvious that S_1 is consistent, since there are no other rules in S_1 and R_1 does not conflict with itself. The same can be said of S_2. Now if we take $S = S_1 \cup S_2$, we can see that R_1 and R_2 will be triggered at the same time, and that they have conflicting actions. This means that if event X occurs, that a conflict will occur between R_1 and R_2. Therefore S is non-consistent. Thus, the union of a consistent set of rules with another consistent set of rules does not necessarily yield a consistent set of rules.

Theorem 2 and Theorem 3 lead us to two important observations. From Theorem 2 we can see that if we find a rule which is inconsistent with itself, then we can stop the validation process and declare that the set of rules is inconsistent. From Theorem 3 we can determine that if we add a new rule to a set of rules previously said to be consistent, we cannot say that the new set is consistent without re-validating the set, even if the rule is consistent with itself.

Therefore our incremental validation algorithm can begin with the following step:

See if a rule is consistent with itself. If it is not, then we can determine that the system is inconsistent. If the rule is consistent with itself, then move onto the next validation step.

There are two ways a rule can be inconsistent with itself: if it contains mutually exclusive conditions, or if it contains opposing actions.

Mutually Exclusive Conditions

If a rule depends on two conditions that can never occur together, then the rule is said to be inconsistent. An example of this would be if a rule had a condition "`(x = '6') ^ (x = '9')`". These two statements can of course never be true at the same time, so we know that the rule will never be triggered. Although this will not result in a runtime conflict, it means that the rule is useless as it will never be fired, and so the user should be warned. Therefore we consider this rule inconsistent.

Opposing Actions

If a rule contains two actions that oppose each other, then the rule is said to be inconsistent. An example of this would be if a rule had an action that said "Close Door A" and another action that said "Open Door A". The end result of these actions is completely dependent on the order that these two actions are performed. Therefore we consider this rule to be inconsistent.

The next step in our algorithm would be to check the new rule to make sure it is not inconsistent with any other rules.

Algorithm V_0:

1. See if the new rule R_N contains two conditions that can never occur together. If so, halt, and return that the new rule is inconsistent.
2. See if the new rule R_N contains two actions which oppose each other. If so, halt, and return that the new rule is inconsistent.
3. For each rule in the policy system, compare it with R_N to see if they have a common condition and an opposing action. If so, halt, and return that the two rules conflict.

Given that there are *n* rules in the system, this process would take O(n), since the new rule needs to be compared with every rule in the system. Compare this with complete revalidation which compares every rule with every other rule, taking $O(n^2)$. Algorithm V_0 is effective, but it compares the new rule with rules which have no chance of conflicting. There must be a more effective way to validate our system of rules.

4.3. A Simple Improvement to Validation

In the algorithm V_0, every single rule in the system was compared with the rule that was changed. We could improve our algorithm by comparing the new rule only with rules that have a possibility of conflicting. We must find a way to determine which rules should be examined.

We know that in order for two rules to conflict with each other, that they must be triggered at the same time. From Theorem 1 we know that in order for two rules to be triggered at the same time, they must both contain at least one common condition. Therefore if two rules do not have any conditions in common, we know they can never conflict, and thus we do not need to compare these two rules when validating.

This makes the set of rules that we need to compare smaller, but can we shrink the set even further. In order for two rules to potentially conflict, two conditions must be met. As mentioned above, the first condition is that both rules contain at least one common condition to trigger them. This means the rules can both be triggered at the same time. The second condition is that each rule must contain an opposing action with the same modality, or the same action with opposing modality. This means if the rules are triggered at the same time, each rule will attempt to do the opposite of the other rule. Depending on which rule we choose to perform first, the outcome will be completely different. Therefore we can compare the new rule to two subsets of rules:

(a) those rules which have an action opposite to one in the new rule but the same modality.

(b) those rules which have an identical action to one in the new rule but has the opposite modality.

Any rule which is in both of these subsets does not even need to be compared with the new rule, since it automatically meets the definition of potentially conflicting

with the new rule. Therefore, it is enough to know which rules fall in both of these subsets of S. Then we can find all the rules that potentially conflict with the new rule.

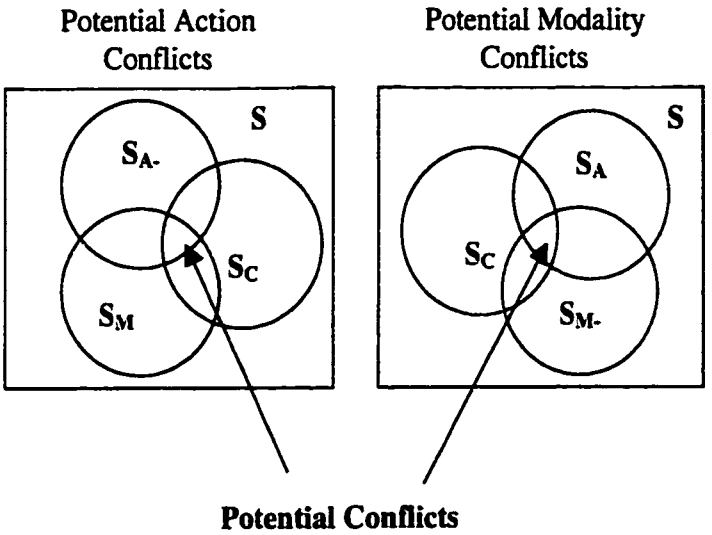


Figure 3 – Potentially Conflicting Rules with R_N

The following explains the symbols in Figure 3:

S : The entire set of rules in the system

S_C: The set of rules that contain a common condition with the new rule, R_N

S_A: The set of rules that contain an action found in R_N

S_{A-}: The set of rules that contain an action opposite to R_N

S_M: The set of rules with the same modality value as R_N

S_{M-}: The set of rules with an opposing modality value to R_N

$C_A = S_C \cap S_M \cap S_{A-}$: The set of rules with a potential action conflict with R_N

$C_M = S_C \cap S_{M-} \cap S_A$: The set of rules with a potential modality conflict with R_N

4.3.1. Development of the Concepts for Algorithm V_1

Let us examine a policy set in decision table form, and see how we can determine which rules potentially conflict. Let us say that initially the table was validated as consistent, and we added a new rule, R_1. We must now find out if R_1 potentially conflicts

with any rules in the table. Let us begin by constructing an exact copy of our decision table, which we can manipulate to eliminate rules we know don't match conflict criteria. (See Figure 4.)

	R_N	R2	R3	R4	R5	R6
M	A+	A-	A+	A-	A+	A-
C1	√			√		
C2			√	√		
C3	√		√			√
C4		√		√	√	
C5					√	√
A1			√			√
A2	√	√	√			
A3	√			√		
A4				√	√	√

Figure 4 – Decision Table Before Validation Process

	R_N	R2	R3	R4	R5	R6
M	A+	A-	A+	A-	A+	A-
C1	√			√		
C3	√		√			√
A1			√			√
A2	√	√	√			
A3	√			√		
A4				√	√	√

Figure 5 – All Irrelevant Rows are Eliminated

The first thing we know is that in order for a rule to conflict with R_I it must contain at least one common condition with R_I. Therefore we can eliminate all conditions from the table that R_I does not have. (See Figure 5.)

Next, we can eliminate the rules whose condition entries are all empty. (See Figure 6.)

	R_N	R3	R4	R6
M	A+	A+	A-	A-
C1	√		√	
C3	√	√		√
A1		√		√
A2	√	√		
A3	√		√	
A4			√	√

Figure 6 – All Empty Condition Entry Rows are Eliminated

	R_N	R3
M	A+	A+
C1	√	
C3	√	√
A1		√
A2	√	√
A3	√	
A4		

	R_N	R4	R6
M	A+	A-	A-
C1	√	√	
C3	√		√
A1			√
A2	√		
A3	√	√	
A4		√	√

Figure 7 – Table is Split by Modality

The next thing we know is that in order for two rules to have a modality conflict they must have opposite modality values. In order for two rules to have an action conflict they must have the same modality values. Therefore, we must examine all rules that have the opposite and the same modality values to R_N, and can ignore the rest. Let us create a new table containing all rules with the opposite modality value as R_N. Let us create another table containing all rules with the same modality value as R_N. (See Figure 7.) We are creating two tables instead of one because the actions that follow on each of the tables will not be the same. Note that if our new rule was of modality O+, then we would split our table into three tables: one for O+, one for O-, and one for A-, since we can have O+/O- and O+/A- modality conflicts.

In the table that contains rules with the opposite modality value as R_N, we are looking for modality conflicts. In order for two rules to have a modality conflict, they must contain at least one common action. Therefore let us eliminate all actions in the table that are not found in R_N. Similarly in the other table, the one with the same modality value as R_N, we are looking for action conflicts. In order for two rules to have an action conflict, one rule must contain an action that is an opposing action to an action in the second rule. Therefore let us eliminate all actions in this table that are not opposing actions to an action found in R_N. (See Figure 8.)

	R_N	R3
M	A+	A+
C1	√	
C3	√	√

	R_N	R4	R6
M	A+	A-	A-
C1	√	√	
C3	√		√
A2	√		
A3	√	√	

Figure 8 – All Non-Opposing Actions are Eliminated

Once this is done, we can again eliminate some rules from each of the tables. This time we remove all rules whose action entries are all empty. (See Figure 9.)

M	
C1	
C3	

	R_N	R4
M	A+	A-
C1	√	√
C3	√	
A2	√	
A3	√	√

Figure 9 – All Empty Action Entry Rules are Eliminated

Finally, we remove R_N from the table with the same modality value as R_N, and the other table if it still exists. Now any rules that are left in either of the two tables are rules that potentially conflict with R_N. In Figure 10 we can see that R_4 has potential modality conflict with R_N.

	R4
M	A-
C1	√
C3	
A2	
A3	√

Figure 10 – Only the Conflicting Rules Remain

Let us examine what we know about the table with the same modality value as R_N. We will call this table T_A. We know that any rule found in T_A has at least one condition in common with R_N. In fact, we know that the conditions it has in common are the conditions that are indicated in T_A after performing the steps above. We also know

that any rule in T_A has at least one action which is an opposing action to one found in R_N. Again we also know exactly which actions they are: they are the ones indicated in T_A. Finally we know, of course, that any rule in the table has the same modality value as R_N. These rules then meet all the requirements necessary to have an action conflict with R_N, therefore we know that all these rules have a potential action conflict with R_N. All the actions and conditions that would be responsible for each rule's conflict are the non-empty entry values in each rule in T_A.

Similarly, in the table with the opposite modality value as R_N, which we will call T_M, we know that any rule found in this table has at least one condition in common with R_N. As with T_A, the common conditions are indicated in the table. We also know that any rule in T_M has at least one action in common with R_N, and again, these actions are indicated for each rule in the table. Finally, of course, we know that each rule in T_M has the modality value opposite to R_N. These rules meet all the necessary requirements to have a modality conflict with R_N, therefore we can say that these rules have a potential modality conflict with R_N. Again, all the actions and conditions responsible for each rule's conflict are the non-empty entry values in each rule in T_M.

4.3.2. Algorithm V_1

R_N: the new rule

D: the input decision table

Algorithm $V_1(R_N,D)$:

1. Create a duplicate of the policy decision table D. Call this table D'.
2. Eliminate from D' any conditions not found in the rule being examined, R_N.
3. Eliminate from D' any rules whose condition entries are all empty.
4. Split D'. Make a new table T_M, containing all rules with the opposite modality to R_N, and a new table T_A, containing all rules with the same modality as R_N. Eliminate all other rules from D'.
5. In T_A, eliminate any actions that are not opposing actions in R_N. In T_M, eliminate any actions that are not found in R_N.

6. In T_A and T_M, eliminate any rules whose action entries are all empty.
7. Eliminate R_N from T_A and T_M.
8. The rules that remain potentially conflict with R_N, and the non-empty entries correspond to the conditions and actions that overlap with R_N. Table T_A contains rules that have action conflicts with R_N. Table T_M contains rules that have modality conflicts with R_N.

4.3.3. Analysis of V_1

We should see if we are doing each step in the right order to obtain the most efficient solution. First we will examine whether the table should be divided before or after removing the conditions. If we divided the table into two smaller tables before we had removed any non-relevant actions, we would have to remove the same conditions for each table. The way we do it, we are only removing each condition once. Therefore, by removing the conditions before dividing the table, we are being twice as efficient as compared to removing them after dividing.

We next examine whether the table should be divided before or after removing the actions. If we divided the table into two smaller tables after we had removed any non-relevant actions, we would have had a lot more work. Since in each table, different kinds of rules were removed, we would not have been able to remove rules and then divide the table. When we pick which rules to remove, we are deciding what kind of conflicts we are looking for. By removing, say, all rules that don't have identical actions to those in R_N, we are choosing to look for modality conflicts. Therefore, once this step is done, the table can no longer be used to find action conflicts. In order to find action conflicts, we would have to start the entire process from D again. Therefore, by dividing before removing actions, we save doing the entire procedure twice. By similar arguments, it is easy to see that removing irrelevant conditions before removing actions is also the more efficient approach.

Say we had a table with C conditions, A actions, and n rules. Our approach starts by examining each condition once and eliminating it if necessary. This takes C operations. Next each rule is examined to see if its condition entries are all empty, and the rule is removed if so. This takes n operations. Let us assume as the worst case that no rules were removed in the previous step leaving us with n rules. The table is then split into two smaller tables, by examining each rule and putting all rules of opposite modality into one table and all rules of the same modality into another table. Let us assume the worst case, that all rules in the original table had a modality value either the same as or opposite to R_N's modality value. Then this takes n operations and all the rules are still present. Say that the first new table contains n_1 rules and the second contains n_2 rules. Next each action is examined to see if it is a possible conflicting action. This is done for each action in each of the two tables so this takes $2 * A$ operations. Finally each rule is examined in each of the tables, to check if all its action entries are empty, and the rule is removed if so. This takes n_1 operations for the first table and n_2 operations for the second table. Note that $n_1 + n_2 = n$, therefore this step takes n operations. Then we can say that our entire process takes $C + n + n + 2 * A + n = C + 2A + 3n$ operations in the worst case.

Let us say we had done the validation by comparing R_N with each other rule in the system (Algorithm V_0). For each rule R_I compared with R_N, we must examine each condition of R_I which takes c_I comparisons, and each action of R_I which takes a_I comparisons. Assume c_I has an average value of c_{avg} and a_I has an average value of a_{avg}. Therefore each rule compared with R_N takes $a_{avg} + c_{avg}$ operations. Assuming we have n rules in the system, there are $n - 1$ rules to compare with R_N, therefore $(a_{avg} + c_{avg}) * (n - 1)$ operations to perform. This method takes $a_{avg}n + c_{avg}n - a_{avg} - c_{avg}$ operations, while our algorithm using decision tables takes $C + 2A + 3n$ operations. Keep in mind that the number of conditions and actions that are possible in a system generally stay constant, therefore as more rules are added to the system, Algorithm V_1 takes only 2 more operations per rule. Compare this to Algorithm V_0. In the absolute best case each rule has one single condition and one single action, therefore the $a_{avg}n + c_{avg}n$ becomes $2n$, and so as n increases, the number of operations needed is less than Algorithm V_1. If

however, there is more than 1 action or condition per rule, then the number of operations rise at a rate greater than or equal to Algorithm V_1. Therefore except under the rare condition where each rule only has one condition and one action, Algorithm V_1 will perform better for large n values.

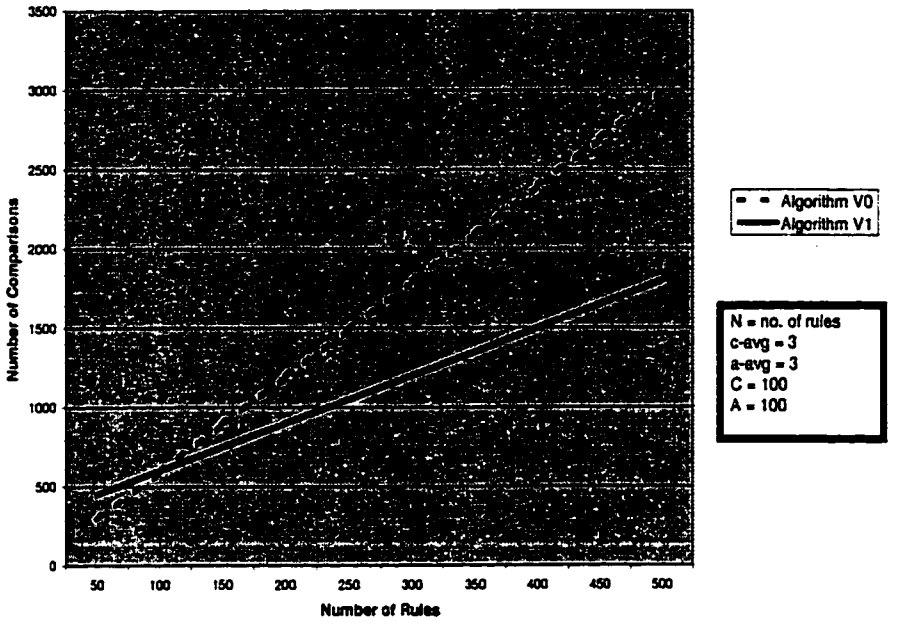


Figure 11 – Graphical Comparison of V_0 and V_1

4.4. *Trigger Chaining*

Suppose we know not only which conditions and actions a rule contains, but also which rules a particular rule may trigger. For example if the execution of R_1 results in an event[1] that is in the condition of R_2, then we can say that R_1 potentially triggers R_2.

Definition 4.4.1:

A rule R_i triggers R_j if and only if there exists an event E, such that E is caused to occur upon the execution of R_i, and E causes R_j to fire.

---

1 "The execution of R1 results in an event" is abbreviated as "R1 throws an event" throughout the rest of the thesis

Definition 4.4.2:

If there is a rule R_1 which upon firing creates an event E_1, and another rule R_2 which has E_1 as a condition, then we say that R_1 potentially triggers R_2.

Therefore even though a new rule R_N may not contain a condition that is in R_2, if it contains a condition in R_1, then there is a chance that it will conflict with R_2 as well. This conflict does not meet the strict definition of conflict we mentioned earlier. In this case we consider it a conflict because the action in R_2 could undo an action performed in R_N, therefore the action in R_N has no lasting effect. A method to detect these conflicts is needed in addition to conflicts that meet our earlier definition.

Definition 4.4.3:

If there is a rule R_1 which throws k events E_1, E_2, ... E_k, and another rule R_2's conditions consist of a subset of the events E_1, E_2, ... E_k, then we say that R_1 triggers R_2. Since R_2's conditions are a subset of all the events thrown by R_1, we know that all R_2's conditions will be met when R_1 fires. Therefore R_1's firing will certainly cause R_2 to be triggered.

We can visualize this by drawing a graph with each vertex representing a rule in the system. Each edge from one vertex to another vertex represents a rule potentially triggering another rule. In Figure 12 we see that R_1 potentially triggers R_2 and R_3, R_2 potentially triggers R_4 and R_5, etc.

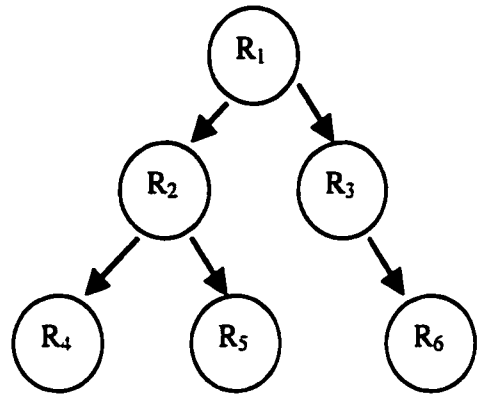


Figure 12 – Trigger Graph for a Set of Rules

Definition 4.4.4:

Suppose a graph is drawn where each vertex represents a rule in a system, and an edge from one vertex to another represents a rule potentially triggering another rule. If there is a path from one vertex V_1 to another vertex V_2 of length *n*, then we can say that R_1 potentially triggers R_2 by degree *n*.

We can now say that if R_1 potentially triggers R_2 by some degree n, and R_2 has an opposing action to an action in R_1, that R_1 and R_2 potentially conflict.

4.4.1. Algorithm V_2

To detect the above kind of conflict, first we start by adding R_N to our graph of the system. We do this by adding a node to represent R_N. Then we draw an arrow from R_N to any rule that contains an event in its condition that is found in R_N's action entry. This is, however, only half done. This connects R_N to the nodes that it potentially triggers, but it does not include the rules that potentially trigger R_N. Suppose we had R_A which had opposite modality to R_N and actions in common with R_N but did not have any conditions in common. Then looking at these two rules alone, they will not be triggered at the same time, and therefore do not potentially conflict. Let us say, though, that there

is a rule R_B which causes events 1 and 2 to occur. Lets us also say that R_N gets triggered upon event 1 occurring, and R_A gets triggered upon event 2 occurring. Therefore, when R_B fires, R_A and R_N potentially will be triggered and could have a conflict. Then it is not only important to look at the tree from R_N down, but to examine the whole tree that R_N is in. So we must make the tree complete.

One way to do this is to keep a list of all rules that cause each event to occur, sorted by event. Then when we add R_N to the graph, we see what events cause it to fire. In this case Event 1. Then we look up Event 1 in our list and see that rules R_B and R_Z cause Event 1 to occur. Then we draw arrows from R_B to R_N and from R_Z to R_N. Then the graph is complete. We must keep track of the roots of the trees we have just connected R_N to. These trees will be used to find any conflicts.

Let us first examine the situation to see which nodes in our graph must be validated. Suppose we have the graph in Figure 13.

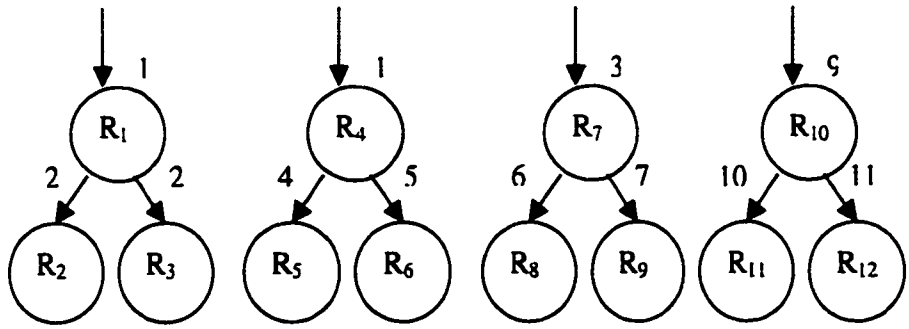


Figure 13 – A Consistent Trigger Graph

In Figure 13 we see that rule R_1 is triggered upon receiving event 1, and upon being triggered, causes event 2 to occur. This triggers R_2 and R_3 to occur, and so on. Suppose that we have previously validated these rules and so no rule in our graph potentially conflicts with any other rule in our graph. This is a consistent set of rules.

Now let us add a new rule, R_N, to the system, resulting in the system shown in Figure 14.

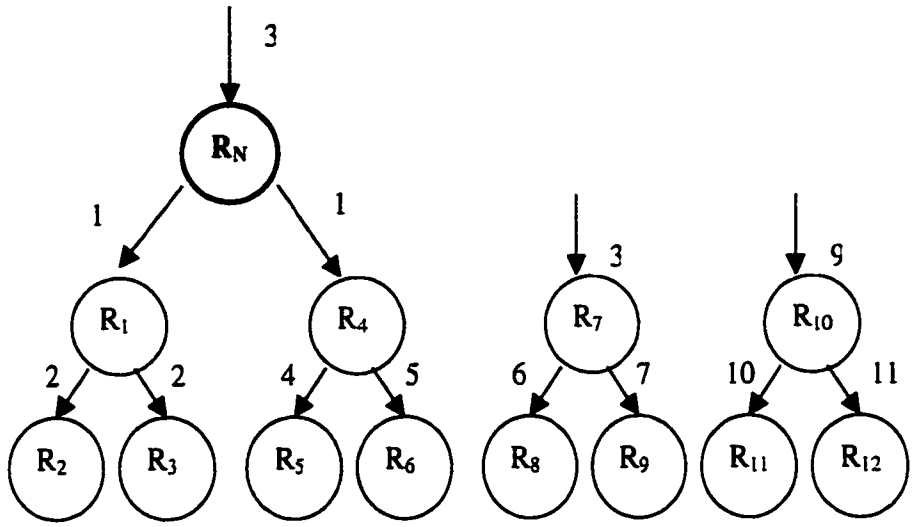


Figure 14 – New Rule Added Above Two Trees with Common Events

As we can see in Figure 14, event 3 causes R_N to fire, which in turn triggers R_1 and R_4. All the rules in the tree whose root is R_N are triggered when R_N is triggered, in this case: R_N, R_1, R_4, R_2, R_3, R_5, and R_6. This means there is the possibility of R_N conflicting with any of these rules. Therefore we must compare R_N with every rule in the tree containing R_N to see if there is a conflict. Not only this, but we see that event 3 causes not only R_N to be triggered, but also R_7. Since none of the rules in R_N's tree were previously triggered upon event 3 occurring, all these rules must be checked to see if they conflict with any of the rules in R_7's tree. In other words, one of the rules in the set $\{R_N, R_1, R_4, R_2, R_3, R_5, R_6\}$ could conflict with one of the rules in the set $\{R_7, R_8, R_9\}$. Since R_1 and R_4 are triggered by the same event, they would have been triggered at the same time, before we added R_N. We already knew that the system of rules was consistent before R_N was added, with R_1 and R_4 being triggered by the same event, there is no change, now that they are both triggered if event 3 occurs as well. Therefore no conflicts will arise between rules in R_1's subtree and R_4's subtree.

Now let us assume we add R_N such that we get the situation depicted in Figure 15.

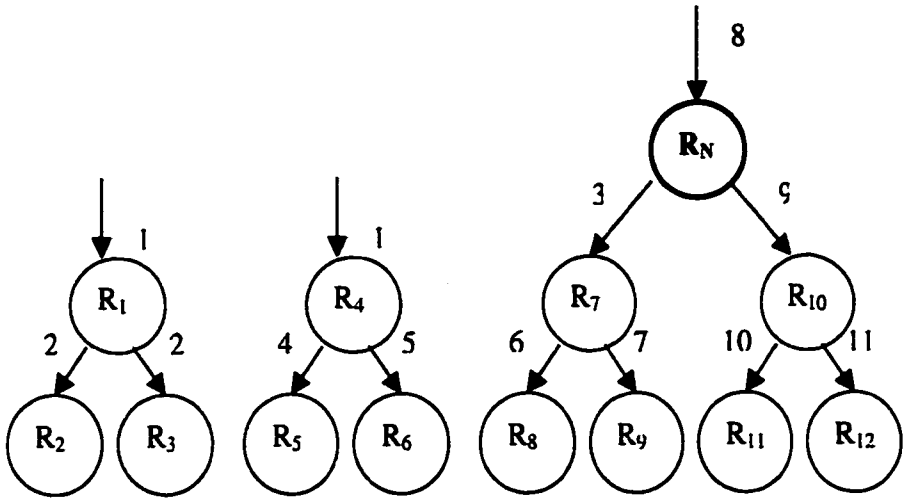


Figure 15 – New Rule Added Above Two Trees with Different Events

In Figure 15 we see that event 8 triggers R_N. R_N causes events 3 and 9 to occur, thereby triggering rules R_7 and R_{10} respectively and so on. As in the previous case, the rules in R_7's subtree and the rules in R_{10}'s subtree must be checked to see if they conflict with the new rule, R_N. This time, however, the two subtrees are not triggered by the same event as in the last case. Although the system was consistent before R_N was added, that conclusion was based on R_7 and R_{10} being triggered by different events, and therefore they could never be fired at the same time. Now R_N firing could cause both R_7 and R_{10} to fire at the same time. This means that the rules in R_7's subtree could possibly conflict with the rules in R_{10}'s subtree. This must be examined. In summary, for this case, it is important to check if R_N conflicts with any of the rules in its tree. Also we must make sure that the rules in every child tree of R_N are consistent with the rules in every other child tree of R_N that is triggered by a different event.

Now suppose we add R_N such that we get the graph shown in Figure 16.

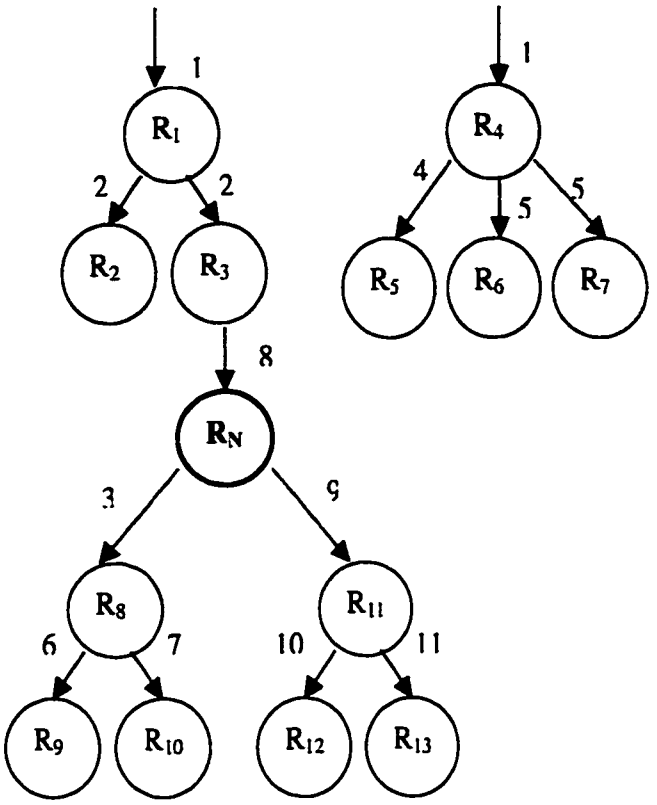


Figure 16 – New Rule Added Above Two Trees and Below Another

In Figure 16 we have added R_N such that it is triggered by event 8, thrown by R_3, and throws events 3 and 9 itself, thereby triggering R_7 and R_{10} respectively. This has added R_N to a tree, where R_N is not the root. As always we must check to see if R_N will conflict with any of the rules in its tree. This time, since R_N is not the root, note that it is not just R_N children we must check against R_N. In this case when the rules in the original tree that R_N was added to, the rules in R_N's subtree are also triggered, which was not the case before. Therefore we must check the rules in the original tree to see if they conflict with R_N's subtree.

Since R_N triggers two rules with different events, as in the last case, we must check that each child tree does not conflict with every other child tree of R_N.

Notice that the root of the tree R_N is contained in R_1. This means that when R_1 is triggered, so are all the other rules of the tree. Therefore, since there are new additions to

this tree, we must see if this tree conflicts with any other tree triggered by the same events. In this case R_4 is triggered by event 1 which triggers R_1. Therefore there could be conflicts between rules in R_1's tree and R_4's tree. Notice, though, that in R_1's tree, all the rules not found in R_N's subtree were originally in that tree. In this case those rules are R_1, R_2, and R_3. Since all these rules were triggered by R_1 in the original system, and the original system was said to be consistent, we know that none of these rules will conflict with the rules in R_4's tree. Therefore the only rules that need to be compared against R_4's tree are those found in R_N's subtree. In summary then, for this case, R_N must be checked to see if it conflicts with any of the rules in its tree. Also any child trees of R_N must be checked to see if they conflict with any child trees of R_N that are triggered by a different event. The tree that contains R_N has events that trigger the root. R_N's subtree must be compared against every tree that is triggered by one or more of these events.

There are three things we can conclude from all this. First of all, we know that no matter where R_N is added, R_N must be checked to see if it potentially conflicts with any of the rules in its tree. Second, if the tree containing R_N is triggered by events $X:\{X_1, \ldots X_n\}$, then all the rules in the subtree with root R_N must be compared with all the rules in any tree triggered by one or more events $X_k \in X$. Third, if multiple trees are merged into one large tree, then the rules of each merged tree must be compared with the rules of each other merged tree. With this information we can build an algorithm to check for the consistency of the rules with the minimum number of comparisons.

In order to find all conflicts, we must perform three steps:

1. Find any conflict between a rule in the original tree R_N was added to, and R_N's new subtree.
2. Find any conflict between rules in R_N's subtree and rules in a tree whose root is triggered by an event that also triggers the root of R_N's tree.
3. Find any conflicts that occur between children trees of R_N.

4.4.2. Step 1

Let us start by creating a decision table that contains all the rules in R_N's subtree, including R_N itself. Let's call this table DT_N. We want to compare these rules with the rules in the original tree that R_N was added to. We can do this by adding one rule from the original tree to DT_N at a time, and using Algorithm V_I to determine if there are any conflicts after each addition. This means, though, that Algorithm V_I will need to be executed once for every rule in the original tree. There may be many rules in the original tree and this could end up being a time consuming process. We should try to improve on this approach. Consider the following situation:

4.4.2.1. Improvement to Step 1

1. Make an empty rule for each modality that can exist in the system. In our case we have a rule for A+, one for A-, one for O+, and one for O-.
2. Next traverse every rule in R_N's subtree. When a rule is traversed, merge all of its actions with the rule along with the corresponding modality. At the end of this process, we will have four rules, each containing all the actions found in R_N's subtree for each modality in the system. We will call the rules R_{NA+}, R_{NA-}, R_{NO+}, and R_{NO-}.
3. Repeat steps 1 and 2 for the original tree to which R_N was added. This time we get the rules R_{OA+}, R_{OA-}, R_{OO+}, and R_{OO-}.
4. Now we can compare pairs of these new compilation rules, one from each tree, to check for conflicts. To check for modality conflicts the following pairs of rules should be examined to see if they contain any common actions:

   R_{NA-} and R_{OA+}

   R_{NA+} and R_{OA-}

   R_{NO-} and R_{OO+}

   R_{NO+} and R_{OO-}

   R_{NO+} and R_{OA-}

   R_{NA-} and R_{OO+}

To check for action conflicts the following pairs of rules should be examined to see if they contain any opposing actions:

R_{NA-} and R_{OA-}

R_{NA+} and R_{OA+}

R_{NO-} and R_{OO-}

R_{NO+} and R_{OO+}

5. If a rule R_{NX} is found to conflict with the rule it was compared to we know that there is a conflict between one of the rules in R_O's tree and one of the rules in R_N's tree, and we continue on to the next step. Otherwise, we know there are no conflicts between those two sets of rules and we can stop.
6. Take R_{NX} and use Algorithm 1 using the rules in R_O's tree, to see which rule in R_O's tree conflicts with it. Call this rule R_{OC}.
7. Once R_{OC} has been found, use Algorithm 1 using the rules in R_N's subtree to find out which rule R_{OC} conflicts with. Call this rule R_{NC}. We now know which two rules conflict with each other in R_O's tree and R_N's tree.

If two rules are detected as potentially conflicting, then we can stop our conflict detection, and report the two rules that conflict. Otherwise, we must continue to Step 2.

4.4.2.2. Explanation of Improvement

Let us examine how Step 1 works. By creating the four rules, we are compiling all the actions that will occur based on R_N being triggered and matching them with the appropriate modality. Since we want to examine the worst case, when all rules in the tree will be triggered, we can ignore the conditions of the rules. In this way we can think of R_N's tree as being all one rule, with all the possible modality values. We are simply dividing this virtual rule into four real rules, each having one single modality value. In order for a conflict to occur between R_N's tree and a tree that gets triggered at the same time as R_N, there must be two actions in each of the trees that conflict. That is to say that either both trees contain a similar action with opposing modalities, or that both trees contain an opposing action with identical modalities.

We then compile the four summary rules for the original tree that R_N was added to. Call the rule at the root of this tree R_O. Now we have four rules for each tree. R_{OO+} contains all actions that will be performed for modality O+ when R_O is triggered (prior to R_N being added to the system). Similarly we have R_{OO-}, R_{OA+}, and R_{OA-}, for actions that will be performed when R_O is triggered for modality O-, A+, and A- respectively.

We now have 8 rules: R_{NO+}, R_{NO-}, R_{NA+}, R_{NA-}, R_{OO+}, R_{OO-}, R_{OA+}, and R_{OA-}. We can now compare R_{NO+} and R_{OO+} to see if there are any actions in R_{NO+} that are opposite to an action in R_{OO+}. If so, we know we have an action conflict between the two trees, since both rules would be triggered by the same events, and are of the same modality. Similarly we can compare R_{NO+} with R_{OO-} to see if there are any identical actions in both rules. If so, we know that there is a modality conflict between the two trees, because we know the two rules are triggered by the same events and are of opposite modality. This can be performed for the other six rules as well. We compare R_{NA+} with R_{OA+}, R_{NO-} with R_{OO-}, and R_{NA-} with R_{OA-} looking for opposite actions, which result in potential action conflicts. We compare R_{NA+} with R_{OA-}, R_{NO-} with R_{OO+}, R_{NA-} with R_{OA+}, R_{NO+} with R_{OA-}, and R_{NA-} with R_{OO+} looking for identical actions, which result in potential modality conflicts.

By performing these ten comparisons, we can determine if there are any potential action or modality conflicts between the two trees. It does not tell us, however, which two rules are the ones that conflict with each other. Therefore the previous actions are performed in order to see whether or not further conflict analysis needs to be done. It is a simple test to determine if there are potential conflicts present. Provided we have found the presence of potential conflicts, it would be necessary to examine the rules further. We should examine whether it is necessary to compare every rule in R_N's subtree with every rule in R_O's original tree. In fact, it is not necessary.

We know which of the summary rules conflicted, and we can use that as a lead. Say that R_{NO+} conflicted with R_{OO-}. Then we know that there is a rule in R_N's tree with

modality O+ that conflicts with a rule in R_O's tree with modality O-. We can take R_{NO+} and compare it with each rule in R_O's tree, until we find a rule that conflicts with it. Let us call this rule R_{OC}. This is one of the two conflicting rules that we are looking for in our system. Then we need to find the corresponding conflicting rule in R_N. We can compare R_{OC} with every rule in R_N's tree until we find a rule that conflicts with it. Let us call that rule R_{NC}.

4.4.2.3. Analysis of Improvement

Let us assume that R_N's subtree contained x rules and R_O contained y rules. If we were to perform comparisons with each rule in R_O with every rule in R_N's subtree, in the worst case there would be no conflict detected, and every rule in one tree would have to be compared with every rule in the other tree. This would take $x * y$ comparisons. Assuming x and y are proportional to n, the total number of rules in the system, this process is $O(n^2)$.

Now let us examine our newly proposed method of detecting conflicts in these trees. The first step is to go through both trees and create summary rules for each tree. Since all rules in both trees need to be traversed, this will take $x + y$ operations. Next 10 comparisons are made between the summary rules to determine if there are any conflicts. If there are no conflicts, we stop after the 10 comparisons. This means the process only took $x + y + 10$ operations. If there are conflicts detected, we move onto the next step.

In the next step, one of the summary rules for R_N's subtree is compared with the rules in R_O. Let us assume the worst case, that the rule in R_O that is causing the conflict will be the last rule traversed in the tree during these comparisons. In that case, finding this rule will take y comparisons. Next, we take this conflicting rule and compare it with the rules in R_N's subtree. Again in the worst case, the conflicting rule in R_N's subtree will be the last rule examined. This will take x comparisons to find. This gives us both of the conflicting rules, with a worst case of taking $x + y + x + y = 2x + 2y$ operations. Assuming x and y are proportional to n, the total number of rules in the system, this

process is O(n). Obviously this is a significant improvement over comparing every rule in R_N's subtree with every rule in R_O.

4.4.3. Step 2

Next, we must compare the rules in R_N's subtree with rules found in trees with root R_T, such that R_T and R_O have at least one triggering condition in common. We use the same technique that was described in Step 1 to do Step 2. First we create summary rules for the tree with root R_T. We already have summary rules for R_N from Step 1. We then compare these rules to determine whether there is a conflict present, and if so further analysis is performed to find the two conflicting rules.

In fact, we know that any rule in a tree with root R_T, such that R_O and R_T have one triggering condition in common, cannot conflict with any rule in R_O's tree. This is because R_O's tree and R_T's tree would have been triggered by the same event before R_N was added to the system. Since we assumed that all rules prior to R_N being added were consistent with one another, we know that no rule in the original R_O tree will conflict with any rule in the R_T tree. Therefore we could do step 1 and step 2 at the same time, by compiling all of the rules in R_O's original tree with all the rules in all the R_T's trees, into the same decision table. Then when the process to solve step 1 is executed, it will not only determine whether or not a rule in R_N's subtree conflicts with a rule in the original R_O tree, but also if they conflict with anything in any of the R_T trees. All this is useless, however, if it does not save any time.

Consider if we have R_N such that its subtree has y rules in it. Now say that R_O's original tree had x rules in it. Also let us assume there are k-1 trees R_{Ti} that have at least one triggering event in common with R_O. Let's also assume each of these trees has x rules. If we compare R_N's subtree to R_O's original tree, and then compare R_N's subtree with each R_{Ti}, we must make k comparisons in all. In order to start we must find the summary rules for R_N. This takes y operations. Next we must find the summary rules of each R_{Ti}. This takes kx operations. For each pair of trees being examined, we must do

the following: First we must do 10 comparisons of the summary rules. Next we use Algorithm 1 on R_{Ti}'s tree, taking $C + 2A + 3x$ operations. Finally we use Algorithm 1 on R_N's subtree, taking

$C + 2A + 3y$ operations.

So, for each pair of trees being compared, we take

$10 + 2C + 4A + 3x + 3y$ operations.

So for k pairs of trees being examined, we use

$k * (10 + 2C + 4A + 3x + 3y)$ operations.

If we were to first merge R_O's original tree with all of the R_{Ti}s and then do our comparison we would have the following. Creating the summary rules for R_N's subtree still takes y operations. Creating the summary rules for our new merged tree would take kx operations (note that the new tree has kx rules). Now we need to do 10 comparisons of the summary rules. Next we use Algorithm 1 on our new merged tree, taking

$C + 2A + 3kx$ operations.

Finally we use Algorithm 1 on R_N's subtree taking a maximum of

$C + 2A + 3y$ operations.

So in all we need

$10 + 2C + 4A + 3kx + 3y$ operations.

Therefore it is advantageous to do Step 1 and Step 2 concurrently.

If two rules are found in Step 2 to be conflicting, then we can stop our conflict analysis and report the conflicting rules. Otherwise we must continue on to Step 3.

4.4.4. Step 3

Finally, we must determine if any of the rules in one of R_N's child trees conflicts with another rule in a different child tree of R_N. Since we already have R_{NA+}, R_{NA-}, R_{NO+}, and R_{NO-}, we can compare these with each other to see if anything in R_N's subtree conflicts with anything else in R_N's subtree. To detect modality conflicts we compare:

R_{NA+} and R_{NA-}

R_{NO+} and R_{NO-}

R_{NO+} and R_{NA-}

and to detect action conflicts we compare:

R_{NA+} and R_{NA+}

R_{NA-} and R_{NA-}

R_{NO+} and R_{NO+}

R_{NO-} and R_{NO-}

If no conflicts are detected with the above comparisons, we can say that R_N has not caused the system to become inconsistent. If conflicts have been detected however, then we must find the exact rules causing the conflict. As in Step 1, we take one of the conflicting R_{NX} and use Algorithm 1 to compare it to the rules in R_N's subtree. We will find one of the conflicting rules. Let us call this rule R_{NC1}. Next we use Algorithm 1 to compare R_{NC1} to the rules in R_N's subtree, and we will find R_{NC2}, the rule that conflicts with R_{NC1}.

Note that this algorithm may be performing some unnecessary comparisons. Conflicts can only occur between child trees of R_N if both of the children are triggered by different events. Our algorithm compares all descendants of R_N with all descendants of R_N. We should examine if this is necessary.

Let us compare only the subtrees that need to be examined. In the best case, there will be only two subtrees that need to be examined. In this case only one comparison needs to be made. This comparison involves creating summary rules for each subtree. Assuming each subtree contains x rules, this takes $2x$ operations. Next 10 comparisons of the summary rules are made. Finally, Algorithm 1 is used twice to find the two conflicting rules, which takes $2 * (C + 2 (A + x)) = 2C + 4A + 6x$ operations. Therefore the entire process takes $2C + 4A + 6x + 10$ operations.

In the worst case there are k subtrees of R_N and all of them have different triggering events, meaning they all need to be compared with each other. Assuming each

subtree has the same number of rules, and assuming R_N's subtree has n rules, we can say each subtree has n/k rules. In order to compare all these rules with each other we must compare the first tree with all the others, which is $k-1$ comparisons. The second tree is compared with all the trees except the first tree, which is $k-2$ comparisons, etc., until the second to last tree is compared to the last tree. This takes

$k-1 + k-2 + \ldots + 2 + 1$ comparisons,

which is equal to

$(k-1)*k/2$ comparisons.

From above we know that comparing two trees of size x takes

$2C + 4A + 6x + 10$ operations.

In this case we are doing this $(k-1)*k/2$ times, each time with size n/k. Therefore we have

$$(k-1)*k/2 * (2C + 4A + 6n/k + 10)$$
$$= (k^2/2 - k/2) * (2C + 4A + 6n/k + 10)$$
$$= Ck^2 - Ck + 2Ak^2 - 2Ak + 3nk - 3n + 5k^2 - 5k$$
$$= (k^2 - k) * (C + 2A + 5) + (k - 1) * 3n$$

Now consider doing one comparison of R_N's entire subtree with itself. Worst case and best case would essentially be the same. Either way would take one comparison. This comparison could at worst take

$2C + 4A + 6n + 10$ operations.

Therefore, comparing it to the worst case of comparing every single subtree, for $k = 3$ we see that the previous case takes

$6C + 12A + 6n + 30$,

which is more than the worst case for comparing the entire R_N subtree with itself. Notice that as k increases, the complete revalidation approach takes more and more operations, whereas our method takes a constant number of operations each time.

4.4.5. Analysis

4.4.5.1. Complete Re-validation

If we were to perform a complete re-validation of the system given that it contains X rules, in the worst case we would have to compare the first rule with all other $X-1$ rules, the second rule with the remaining $X-2$ rules, etc. Therefore this would take

$X-1 + X-2 + \ldots + 2 + 1$ rule comparisons.

Assume each rule has c_{avg} conditions and a_{avg} actions. Then each rule comparison would require c_{avg} comparisons of conditions between the two rules, and a_{avg} comparisons of actions between the two rules. This is equal to

$(X) * (X/2) = X^2/2$ rule comparisons $* c_{avg} + a_{avg}$ operations per rule comparison

$= (c_{avg} + a_{avg}) * X^2/2$ operations in total.

Therefore complete re-validation is $O(X^2)$ in the worst case. Note that since we are performing re-validation of the entire system, the best case is the same as the worst case. Therefore the worst case also takes $(c_{avg} + a_{avg}) * X^2/2$ operations, and thus is $O(X^2)$.

In the best case, the first two rules that are compared would conflict and conflict detection would end. This would take 1 operation, and thus is O(1).

4.4.5.2. Algorithm V_2

Let us say that we had the following:

R_N = a rule that is added to a system.

R_O = the root of the tree R_N was added to.

k = the number of other trees R_{Ti} that each have at least one triggering condition in common with R_O.

t = the average number of rules in each R_{Ti}.

y = the number of trees R_N took as its children when it was added.

s = the size of the R_N new child trees.

n = the number of rules in R_N's tree.

m = the number of rules in R_O's original tree.

In section 4.4.3 we showed that merging R_O's original tree with all of the R_{Ti}s to compare with R_N's subtree takes

$10 + 2C + 4A + 3(kt+m) + 3n$ operations in the worst case.

In 4.4.4 we showed that Step 3 takes

$2C + 4A + 6n + 10$ operations in the worst case.

Therefore our entire process in the worst case takes

$10 + 2C + 4A + 3kt + 3m + 3n + 2C + 4A + 6n + 10$

$= 20 + 4C + 8A + 3kt + 3m + 9n$ operations.

Assuming m, n, and t are proportional to X, we can say that in the worst case our algorithm is O(X).

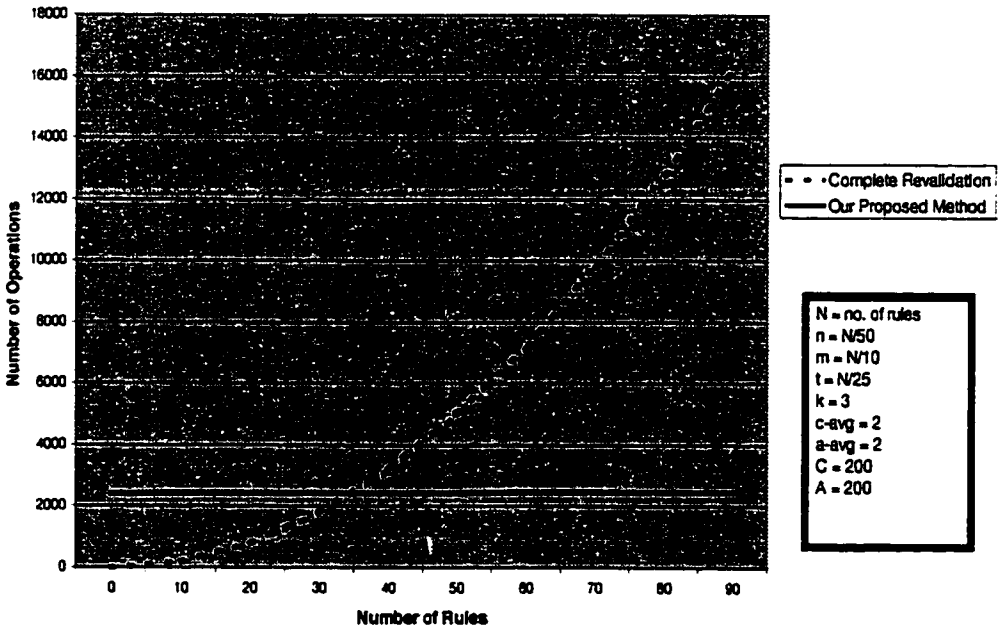


Figure 17 - A Graphical Comparison of the Two Algorithms

In the best case Step 1 and 2 are completed without detecting a conflict. It takes $kt + m$ operations to compile the summary rules for the merged tree. Next it takes n operations to compile the summary rules for R_N's subtree. Finally 10 comparisons are made and it is determined there are no conflicts. Similarly in Step 3 we do 10 comparisons of R_N's summary rules, and find no conflict. In this case, the entire process

takes $20 + kt + m$ operations. If once again we assume that m and t are proportional to X, we can say that in the best case our algorithm is O(X). Figure 17 shows a graphical comparison of the complete revalidation method with our algorithm. Note that because of the large difference in performance the scale of the graph was chosen to illustrate that complete revalidation performs in quadratic time. This scale gives the illusion that our method performs in constant time, when in fact it performs in linear time with a small slope.

4.5. Cyclic Conflict Detection

Another possibility for a conflict is if a rule R_I triggers a rule R_J which triggers a rule R_T etc., until finally a rule R_T is triggered which triggers R_I again. This causes a cycle of triggers. This means that rules are constantly being triggered, and may never stop executing. Not only this but depending on the rules being triggered, if a job is being passed onto the next rule, then the job will never be completed because no rule ever tackles the problem. For example say R_1 owned by Bill states that upon receiving a work-related e-mail, the e-mail should be forwarded to John, and R_2 owned by John states that upon receiving a work-related e-mail, the e-mail should be forwarded to Bill. In this case if either Bill or John ever receives a work-related e-mail one rule will trigger the other which triggers the first rule, and this cycle continues on and on and on. The e-mail never gets read, and many CPU cycles are wasted by from this endless passing the buck. This behaviour needs to be detected in the rules and reported.

Let us make a graph showing each rule as a vertex in this graph. Now let us connect the vertices using directed edges, each edge drawn from one vertex to another representing that the first rule triggers the second. Now if we examine the graph and detect a cycle, we know that one rule may, after a trigger chain reaction, effectively trigger itself.

In Figure 18 we see that R_1 triggers R_3 which triggers R_6 which triggers R_1. Since this forms a cycle in our directed graph, we know that there is a potential cyclic conflict involving R_1.

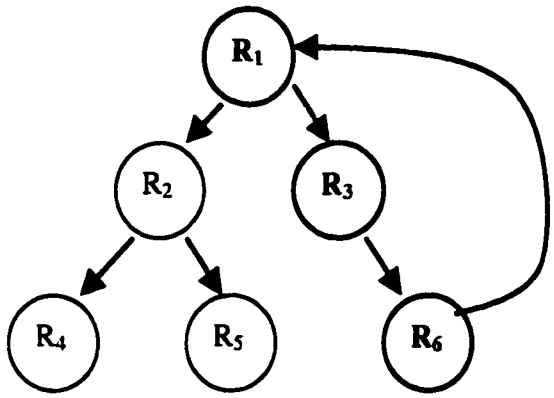


Figure 18 – A Cyclic Conflict

Note that if we were to draw a similar graph, this time having the edges representing if a rule potentially triggers another rule, and performed the same search for cycles, we could find potential conflicts of this sort. Since, however, the chance of all the conditions in one of these cycles being met over and over, without a rule to throw all the events needed, is not particularly high in most cases, this is less likely to result in a conflict. Still, because of the potential conflict, it is a pattern that should be detected so that at very least the user can be warned, and in the best case the policy will be redesigned.

4.6. Summary

In summary, although the best case of the complete revalidation approach performs better than our algorithm in the best case, our algorithm performs much better in the worst case. Also our best case (having no conflicts) seems much more likely to occur than having the two first rules in the system to conflict, as is needed for the best

case of the complete revalidation approach. Regardless of this, we have shown considerable improvement over the complete revalidation approach with our algorithm, and thus have demonstrated the benefits of incremental validation.

5. Architecture for Incremental Validation

We have shown how incremental policy validation can improve the performance of policy-based systems. It is now important to show how policy-based systems can be built that can use incremental validation. To do this, we will take an existing policy architecture and show what modifications will be necessary in order to take advantage of incremental validation.

5.1. An Existing Architecture

In his thesis, Kanthimathinathan [42] described an architecture that supports many popular policy models. The main modules necessary for policy input are a PSL Editor, an EA Model Handler, a Policy Handler, a Conflict Handler, and a Policy Engine, as illustrated in Figure 19. A policy is entered into the system through the PSL Editor. From there it is sent to the EA Model Handler where it is examined to see if the specifier of the policy has the permissions necessary to act upon the targets and other program entities indicated in the policy. If this level of verification succeeds, the policy is then sent to the Conflict Handler module. Kanthimathinathan does not mention the explicit purpose of the Conflict Handler but presumably it is to check for such things as modality conflicts and application specific conflicts, since after this stage the policy is sent to the Policy Service where it begins its transformation into executable rules. These executable rules are then sent to the Policy Engine which enforces them.

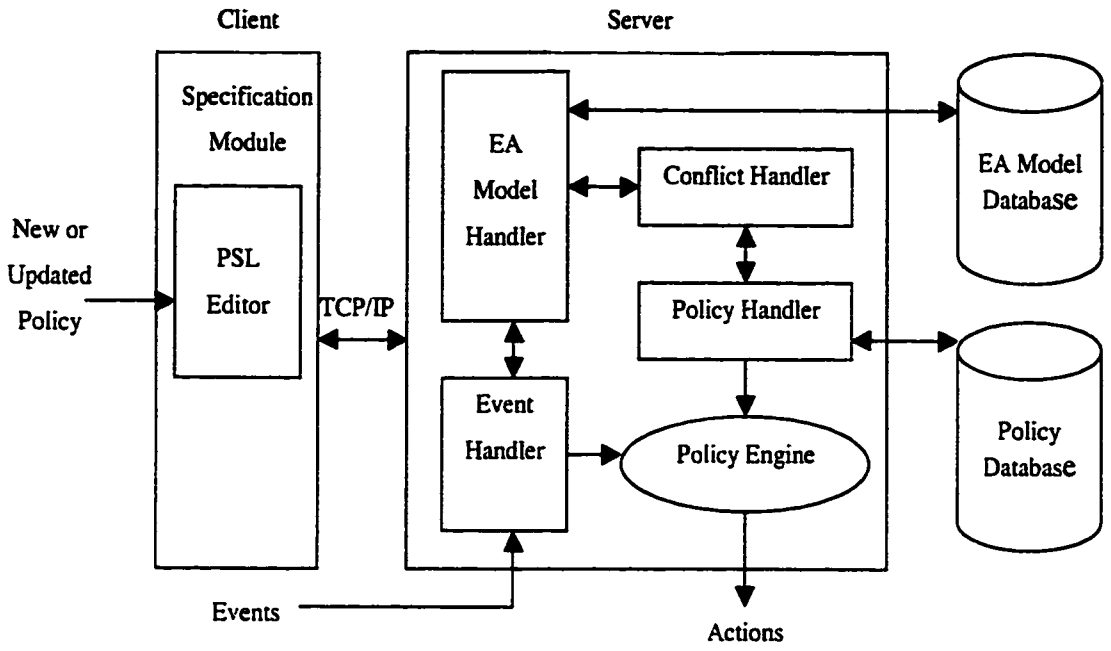


Figure 19 – Kanthimathinathan's Policy Architecture

5.2. What is Needed in a Conflict Handler

We will now define what should be present in this Conflict Handler module. Since we can add policies to the system one at a time, we can perform incremental validation as each policy is added. Then we may think it is possible to always validate the policies using the incremental validation technique. However, there are cases where a validation of the entire set of policies is necessary. Take for example the case where an event is removed from the system. There could be two rules that have identical condition sets. Since these rules had overlapping condition sets to begin with, they already would have potentially conflicted, so we do not need to perform a full policy set validation due to this. However, these rules could now become identical. This is not really a problem, but having multiple identical rules takes up unnecessary space and processing time during validation, so these duplicate rules should be removed.

We could also have rules which have empty condition sets. In this case, these rules should be removed from the system. Therefore every time an event is removed from the system, all the rules in the system should be examined and all rules with empty condition sets should be removed. The same is true if an object that is tested in condition statements is removed from the system. Although the entire policy set must be examined, this scenario does not introduce any conflicts into the system, therefore it is not necessary to perform any conflict detection between policies.

If a target or a function that can be performed on a target is removed from the system, then we have a similar case to the case above. We could have two or more rules that now have identical action sets. This does not pose a problem since we are allowing multiple rules to have identical actions sets. However, again, if these rules are identical then they should be removed to reduce the amount of resources needed. Just like removing an event could result in rules with empty condition sets, removing a target or target function can result in rules with empty action sets. Therefore every time a target or target function is removed from the system, all the rules in the system should be examined and all rules with empty action sets should be removed. Again in this scenario no conflicts between policies are introduced into the system and hence conflict detection between policies is not necessary in this case.

If an event, target, or function that can be performed on a target is added to the system, there is almost no change in the system. The decision tables in the system will have to have another row added to them, which will result in a new entry for each rule. By default this entry will be blank which represents a don't-care-entry, accepting all values for the new attribute. We can see then, that the set of policies will not change in terms of what events and conditions trigger them, and what actions they perform. Therefore, when a new event, target, or function that can be performed on a target is added to the system, no re-validation of the set of policies is necessary.

When a new policy is added to the policy set, or an existing policy is modified, we have the possibility of introducing conflicts into the system. As we have shown in

Chapter 4, these conflicts can be detected efficiently using an incremental validation technique. As long as incremental validation is performed after every policy addition or modification, no method of validating the entire set of policies will be needed for such modifications to the system.

Therefore, from the above case analysis, we can see that our Conflict Handler will need two types of policy validation. The first is the incremental conflict detection explained in Chapter 4. The second is the ability to examine all the policies for duplicate or empty rules and remove them. If we include trigger chaining as a property we can check for, then cyclic conflict detection should also be performed.

In order to perform incremental validation and cyclic conflict detection our Conflict Handler can use a tree representation of the set of policies.

5.3. Scenarios

This section provides various scenarios for changing the policy set in the system. We discuss what the flow of information when a policy is added, modified, or deleted from the system, and what happens when conditions, events, and actions are added or removed from the system.

5.3.1. Adding a policy

Upon adding a policy to the system, the new policy will move from the PSL Editor to the EA Handler. From here it will be sent to the Conflict Handler. The Conflict Handler will then create a decision table of all policies that are in the same scope as the new policy including the new policy itself. The policy will also be added to the appropriate place in the policy tree. The tree will then be used to perform incremental validation on the policy. If validation fails, the policy will be removed from the policy tree. If validation succeeds, then the policy will be added to the appropriate decision

table, and the policy will be sent to the Policy Service where it can be translated into executable rules and sent to a Policy Engine.

5.3.2. Modifying an Existing Policy

The scenario for modifying an existing policy is nearly identical to adding a new policy. In this case, when performing validation, one proceeds as indicated in 5.3.1, however, this time the policy that the new policy is replacing should be excluded from the decision table and the policy tree. If validation succeeds, the new policy will replace the old policy both in the decision table and policy tree. The new policy is sent to the Policy Service which translates the policy into executable rules which are sent to the Policy Engine. The Policy Service also requests the old policy be removed from the Policy Engine.

5.3.3. Deleting a Policy

In the case of deleting a policy, there is no conflict detection to be done, provided that no policies are dependent on other policies. In this case, the Conflict Handler can be completely bypassed and the Policy Service will request the indicated policy be removed from the Policy Engine.

5.3.4. Adding an Event, Condition, or Action

If an event, condition, or action is added to the system, then all decision tables in the system must be modified to accommodate this change. This is done by adding a row with the appropriate stub to all decision tables in the system. This will not result in any conflicts, so the Conflict Handler is not needed in this case.

5.3.5. Deleting an Event, Condition, or Action

When an event, condition, or action is deleted from the system, the row containing that attribute must be deleted from every decision table in the system. This

could result in duplicate rules, or rules that have completely empty condition entries or action entries. In the case of duplicate rules, this does not lead to a conflict, however, duplicate rules are unnecessary and take up an extra amount of memory and processing time. This is especially true when validating rules, which we have shown to be dependent on the number of rules in the system. Therefore the rules should be examined and any duplicate rules should be removed. Similarly, if there are rules with empty condition entries or action entries, these rules are not valid, and should be removed. Therefore, in this case, the Conflict Handler should modify all decision tables in the system and remove any duplicate or empty rules. The policy tree does not need to be modified in this case. The Conflict Handler should then notify the Policy Handler which rules should be updated or deleted from the Policy Engine.

5.4. Transition to Enforcing Updated Policies

When new policies are added to the system, the objects using the system must be updated to use the new policies. An important question, however, is exactly how the transition will be made to the new policies. We suggest two ways to do this: the stop and reload method, and the static generation method.

5.4.1. Stop and Reload Method

The idea behind the stop and reload method is to provide all the objects in the system with the very latest updates to the policies. In order to do this, the execution of the system should be stopped, so that all objects can update their policies from the database. The advantage of this method, of course, is that all objects will have the very latest version of their policies. This also means that all objects in the system will have the same version of a particular policy and so all objects referring to a particular policy will behave in the same manner. The disadvantage to using this technique is that the system must be halted temporarily while the objects refresh their policies. Depending on the number of objects and policies in the system, this may take a noticeable amount of time and may disrupt the appearance of continuous service to the users of the system.

5.4.2. Static Object-Policies Method

The stop and reload method of policy transition required that the system halts execution while the objects refresh their policies from the database. The static object-policies method avoids this. With the static object-policies method, each object fetches its policies from the database as the object is created and never updates its policies again. If a change is made to the policy database, the change will affect the new generation of objects, but the objects created before the policy change will retain the old version of the policy. This solves the problem of having to halt the system every time a change is made to the policy database. However, it does have disadvantages.

One disadvantage is that there are various objects in the system which have differing versions of the same policy. This means that otherwise identical objects may behave differently because of the varying policy versions. The other disadvantage is that the policies for each generation of objects is static, and so although the policy-based system may be dynamic, i.e., it allows policies to be changed during the execution of the system, the objects will never inherit these new behaviours. The objects will only take on the new behaviours if the objects are deleted and created again.

5.5. Summary

Building a system which implements incremental policy validation is not a simple task. However, extending the architecture of a current policy-based system makes this task somewhat easier. We have shown the components that are required to make an existing policy-based system able to validate its policies incrementally. We have also discussed the problem of policy transition when policies are changed in the system, and offered two possible solutions to this problem.

6. Conclusion

The main contribution of this thesis was exploring the problem of incremental policy validation, and providing a method of solving this problem. Incremental policy validation is an area in which no prior work has been done so far. With the increase in number of systems that use policies, and the large size of policy systems, incremental policy validation will become an important problem. By validating only those rules which have a chance of conflicting, we can substantially reduce the execution time of the validation process of policy systems.

As a way of solving this problem, we have developed an algorithm. We have developed the concepts used in the algorithm, in a step by step manner, and refined those concepts to arrive at the final algorithm. Then, we analyzed the algorithm for its performance and compared it to the complete revalidation approach of revalidating all policies in the system. This analysis showed that incremental policy validation does offer a significant advantage over complete revalidation.

Trigger chaining was also introduced in this thesis. Trigger chaining consists of looking at policies to see which chain of policies they trigger. This allows us to find more conflicts, and so an incremental validation algorithm was suggested to find these conflicts. This algorithm was analysed and compared to a complete revalidation method. Our incremental validation method was shown to operate in linear time, as opposed to the complete revalidation method which was found to have quadratic complexity.

Trigger chaining also introduced a second new type of conflict. Cyclic conflicts were introduced as a policy which triggers a chain of policies to fire, resulting eventually in its own execution. This new type of conflict was not examined in much detail in this thesis. However, it was suggested that directed graphs could be used to detect these conflicts.

We have also introduced the idea of representing policies in decision tables. There has been much work done in the past related to decision tables which we can use with respect to policies. The problem of decision table completeness and decision table consistency have all been studied in the past. They helped us in refining the concepts in our current approach to policy validation, and they can also help in some future work. The problem of converting decision tables into executable programs has also been examined at length by other researchers. This research can be used to solve the problem of converting policies to some form of executable code, which needs further exploration.

Decision tables have also given us a benefit in the form of conceptual simplicity. By representing policies in this simple representation, it is easy to see the differences between policies, what happens when new policies are added or old policies are removed, etc. This view makes it simpler to develop algorithms related to policies. In fact, this representation helped in the task of developing an incremental validation algorithm, by providing a simple way to eliminate conditions, actions, and entire rules from the search space. We presented policies in the form of decision tables and then showed the limitations of this format. We then extended the decision table format to meet the requirements of representing policies.

Developing a system using incremental policy validation is not a small task. We have taken an existing policy framework from the literature and have shown what modifications are necessary in order to achieve a system capable of incremental validation.

7. Future Work

In this thesis we have shown that the concept of incremental policy validation is a valid one, and is useful. We have introduced an algorithm which solves this problem and performs better than the complete revalidation approach. Future work may include developing and analyzing an algorithm which performs the same task but does it with optimum performance.

Further work could also include implementing the incremental algorithm given in this thesis to show that it performs well in practice as well as in theory. This would ideally be done with a large system consisting of many rules, since as we have shown, the difference between the two algorithms becomes very apparent when there are a large number of rules in the system.

Although we did not explicitly restrict our algorithm to simple decision tables, whereby each entry has only yes or no values, we did not go into detail how tables with allowing entries with multiple values would affect incremental policy validation. Similarly we did not examine the case of two conditions that overlap. For example, in our work we have considered conditions such as "(x = 6)". In our system a rule with condition "(x = 6)" would not conflict with a rule with condition "(x > 3)" since the two conditions are not the same. Our table would have a row in the decision table for each condition and our validation algorithm only checks if two rules are triggered by the same row in the decision table. If we were to allow conditions such as "x" where the values could be " = 6" or " > 3", then we could check for conflicts more completely.

Trigger chaining was introduced as a concept in our thesis, but it was only used to examine incremental static policy validation. Trigger chaining may or may not have an effect on run-time policy validation, which remains to be explored. Similarly, cyclic conflict detection was also introduced, but was not examined in very much detail. It would be useful to see if cyclic conflict can occur at run-time provided an incremental

static policy validation mechanism is in place. If so, it would be worth while to develop an algorithm to solve this problem.

We introduced the notion of representing policies as decision tables in this thesis. This is one area that could possibly be expanded upon in the future. Benefits taken from past work in the area of decision tables could be examined to see if it can be used to improve policy systems using this representation. For example, the notion of intracolumn inconsistency may aid in creating a more efficient validation algorithm. Also, the extended decision table format we introduced could possibly be expanded upon to add functionality which our format does not currently provide. For example, the decision table itself does not currently indicate who is able to change the policies or where the policies sit in the organizational hierarchy.

8. References

[1] Gruber, R. E., Krishnamurthy, B., Panagos, E., "High-Level Constructs in the READY Event Notification System", *Eighth ACM SIGOPS European Workshop*, Sinatra, Portugal, Sept. 1998.

[2] Krishnamurthy, B., Rosenblum, D. S., "Yeast: A General Purpose Event-Action System", *IEEE Transactions on Software Engineering*, vol. 21, no. 10, Oct. 1995.

[3] Rosenblum, D. S., Wolf, A. L., "A Design Framework for Internet-Scale Event Observation and Notification", *Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT symposium on Software engineering*, September 22 - 25, 1997, Zurich Switzerland, pp. 344-360

[4] Lupu, E., Sloman, E., "Conflicts in Policy-Based Distributed Systems Management", *IEEE Transactions on Software Engineering* vol. 25, no.6. Nov./Dec. 1999.

[5] Omari, S., Boutaba, R., "Policy-Based Control Agents for Boundary Routers in Differentiated Services IP", *First International Workshop on Mobile Agents for Telecommunications Applications*, pp. 477-490, Oct. 1999.

[6] Lupu, E., Sloman, E., "Conflict Analysis for Management Policies", *Proceedings of the 5th International Symposium on Integrated Network Management IM'97*, 1997.

[7] Cholvy, L., Cuppens, F., "Analyzing Consistency of Security Policies", *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pp.103-112, 1997.

[8] Sommerville, I., Software Engineering, Fifth Ed., Addison-Wesley, USA, 1996.

[9] Michael, J., Sibley, E., Littman, D., "Integration of Formal and Heuristic Reasoning as a Basis for Testing and Debugging Computer Security Policy", *Proceedings of the New Security Paradigms Workshop*, Los Alamitos, California, pp. 69-75, 1993.

[10] Moffett, J. D., Sloman, M. S., "Policy Hierarchies for Distributed Systems Management", *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 9, pp. 1404-1414, Dec. 1993.

[11] Grosof, B. N., Labrou, Y., Chan, H. Y., "A Declarative Approach to Business Rules in Contracts: Courteous Logic Programs in XML.", *Proceedings of the First ACM Conference on Electronic Commerce*, Nov. 1999.

[12] Chomicki, J., Lobo, J., Naqvi, S., "Conflict Resolution in Policy Management", www.cs.buffalo.edu/~chomicki/papers-tkde01.ps, Submitted June 12, 2000

[13] Smith, I. A., Cohen, P. R., Bradshaw, J. M., Greaves, M., Holmback, H., "Designing Conversation Policies Using Joint Intention Theory", *Proceedings of the International Conference on Multi Agent Systems*, pp. 269-276, 1998.

[14] Grossner, C., He, X., Kurusetty, B., Mahoney, G., Radhakrishnan, T., "The Use of a Restricted Natural Language and an Intermediate Representation in Policy Based Systems", Work in Progress

[15] Falchuk, B., Karmouch, A., "Visual Modeling for Agent-Based Applications", *IEEE Computer*, Dec. 1998, pp. 31-38.

[16] Jajodia, S., Samarati, P., Subrahmanian, V. S., "A Logical Language for Expressing Authorizations", *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pp. 31-42, 1997.

[17] Koch, T., Krell, C., Krämer, B., "Policy Definition Language for Automated Management of Distributed Systems", *Proceedings of the Second IEEE International Workshop on Systems Management*, pp. 55-64, 1996.

[18] Cuppens, F., Cholvy, L., Saurel, C., Carrere, J., "Merging Security Policies: Analysis of a Practical Example", *Proceedings of the 11th IEEE Computer Security Foundations Workshop*, pp.123-136, 1998.

[19] Howard, S., Lutfiyya, H., Katchabaw, M., Bauer, M., "Supporting Dynamic Policy Change Using CORBA System Management Facilities", *Proceedings of the 5th IFIP/IEEE International Symposium on Integrated Network Management (IM '97)*, pp. 527-538, 1997.

[20] Cuppens, F., Saurel, C., "Specifying a Security Policy: A Case Study", *Proceedings of the 9th IEEE Computer Security Foundations Workshop*, pp.123-134, 1996.

[21] Bertino, E., Jajodia, S., Samarati, P., "Supporting Multiple Access Control Policies in Database Systems", *Proceedings 1996 IEEE Symposium on Security and Privacy*, pp.94-107, 1996.

[22] Howard, S., Lutfiyya, H., Katchabaw, M., Bauer, M., "Supporting Dynamic Policy Change Using CORBA System Management Facilities", *Proceedings of the Fifth IFIP/IEEE International Symposium on Integrated Network Management (IM'97)*, San Diego, California, May 12-16, 1997.

[23] Fraser, T., Badger, L., "Ensuring Continuity During Dynamic Security Policy Reconfiguration in DTE", *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, 1998.

[24] Mahon, H., Bernet, Y., Herzog, S., Schnizlein, J., "Requirements for a Policy Management System", IETF Internet Draft, www.ietf.org/internet-drafts/draft-ietf-policy-req-02.txt, Nov. 9, 2000.

[25] Shay, W. A., Understanding Data Communications and Networks, PWS Publishing Company, USA, 1995.

[26] Moore, B., Ellesson, E., Strassner, J., Westerinen, A., "Policy Core Information Model", IETF document RFC 3060, www.ietf.org/rfc/rfc3060.txt, Feb. 2001.

[27] Hilpinen, R. (ed.), Deontic Logic: Introductory and Systematic Readings, D. Reidel, Dordrecht-Holland, 1971.

[28] Marriott, D., Sloman, M., "Implementation of a Management Agent for Interpreting Obligation Policy", *IEEE/IFIP Workshop on Distributed Systems Operations and Management (DSOM '96)*, Laquila, Italy, Oct 1996.

[29] Montalbano, M., Decision Tables, Science Research Associates, USA, 1974.

[30] Welland, R., Decision Tables and Computer Programming, Hyden & Son, Great Britain, 1981.

[31] Ibramsha, M., Rajaraman, V., "Detection of Logical Errors in Decision Table Programs", Communications of the ACM, vol. 21, no. 12, pp. 1016-1025, Dec. 1978.

[32] King, P. J. H., Johnson, R. G., "Some Comments on the Use of Ambiguous Decision Tables and Their Conversion to Computer Programs", Communications of the ACM, vol. 16, no. 5, pp. 287-290, May 1973.

[33] Shwayder, K., "Combining Decision Rules in a Decision Table", Communications of the ACM, vol. 18, no. 8, pp. 476-480, Aug. 1975.

[34] Muthukrishnan, C. R., Rajaraman, V., "On the Conversion of Decision Tables to Computer Programs", Communications of the ACM, vol. 13, no. 6, pp. 347-351, June 1970.

[35] Lew, A., "Optimal Conversion of Extended-Entry Decision Tables with General Cost Criteria", Communications of the ACM, vol. 21, no. 4, pp. 269-279, April 1978.

[36] Shumacher, H., Sevcik, K.C., "The Synthetic Approach to Decision Table Conversion", Communications of the ACM, vol. 19, no. 6, pp. 343-351, June 1976.

[37] Dathe, G., "Conversion of Decision Tables by Rule Mask Method Without Rule Mask", Communications of the ACM, vol. 15, no. 10, pp. 906-909, Oct. 1972.

[38] Giarratano, J., Riley, G., Expert Systems Principles and Programming, PWS Publishing, USA, 1998.

[39] Jess the Java Expert System Shell, http://herzberg.ca.sandia.gov/jess; accessed March 12, 2001.

[40] Rowe, N. C., Artificial Intelligence Through Prolog, Prentice Hall, USA, 1998.

[41] Vanthienen, J., Wets, G., "From Decision Tables to Expert System Shells", Data & Knowledge Engineering 13(3), pp. 265-282, Oct. 1994.

[42] Kanthimathinathan, V., "An Enterprise Policy Specification Tool", M. Comp. Sci. Thesis, Concordia University, Feb. 2001.

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Conference Management System

Hui Guan

A Project

In

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

March 2001

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that major report prepared

By: Hui Guan

Entitled: Conference Management System

and submitted in partial fulfillment of the requirement for the degree of

Master of Computer Science

Compiled with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Lixin Tao
Examiner

Supervisor

Approved by
Chair of Department or Graduate Program Director

JUL 10 2001

Dr. Nabil Esmail, Dean
Faculty of Engineering and Computer Science

ABSTRACT

Conference Management System

By Hui Guan

Internet has been recognized and accepted by the public and organizations around the world as a new and exciting opportunity for expanding the management through WWW site and changing the way those companies, large and small, do business. Conference Management System (CMS) – a web-based information and database management application system is developed to provide the solution to a large academic information management system for international professional conference. CMS consists of a 3-tier client/server application framework. CMS object is not a monolithic piece of code; instead, it is more like a Lego of cooperating parts that we can break apart and then reassemble along the 3-tier client/server line. The front tier displays the visual aspects of the CMS. These visual objects typically live on the client. The middle tier works with the server objects that represent the persistent data and the CMS logic functions. The back tier is the database that stores all the system information, and it can also be administrated without influences from the front and middle tier. Middle-tier server objects interact with their clients (the view objects) and implement the logic function of the CMS. The main functions of Conference Management System (CMS) are to interactively collect conference papers and their corresponding information from the web site, putting them into database and then allocate these papers among the Program Committee members. Also it provides all the information to the General Chair and assists administrating the Conference Management System and database.

ACKNOWLEDGMENTS

The completion of this project is due to the direction and instruction of my supervisor, Dr. Bipin C. Desai who gave me many valuable suggestions and advices. First, I would like to express my heartfelt thanks to him. Without his instructions I wouldn't have finished this project. He has provided a supportive work environment and kept encouraging me to make progress in doing this project. Here, I'd like also to thanks my husband for all kinds of support he gave me. I also want to thank my mom and dad for giving me help in take caring of my daughter while I was in Montreal doing the project. Without their assistance, I would still be working on the project. Thanks to all.

Table of Content

List of Figure

List of Table

1 Introduction

Information System has become a powerful solution to the efficient organization management. Internet has been recognized by organizations around the world as a new and exciting opportunity for expanding the enterprise through Web sites. A web application can be as simple as a keyword search on a document archive or as complex as an electronic storefront. Web applications are being deployed on the Internet and on corporate intranet and extranet, where they have the potential to increase productivity and change the way that companies, large and small, do business. The project of Conference Management System gives a great inspiration to practically explore more opportunities in this field. This document is a record of techniques used in developing this useful high tech system.

Conference Management System (CMS) is a web based information and database application system using a 3-tier client/server application framework. CMS object is not a monolithic piece of code. Instead, it is more like a Lego of cooperating parts that we can break apart and then reassemble along the 3-tier client/server line. The first tier represents the visual aspects of the CMS. These visual objects typically live on the client. In the middle tier are server objects that represent the persistent data and the CMS logic functions. In the third tier are existing databases. Middle-tier server objects interact with their clients (the view objects) and implements the logic function of the CMS. They can extract their persistent state from the multiple data sources. For example, SQL databases

or HTML files. The server object provides an integrated model of the disparate data sources and back-end applications. Client never directly interacts with third-tier data sources. These sources must be totally encapsulated and abstracted by the middle-tier server objects.

The main functions of Conference Management System (CMS) are to collect papers and author information from the web site, put them into database and then allocate these papers among the Program Committee (PC) members. Also it provides all the information to the General Chair (GC) and assists him to administer the Conference Management System and database. Many of these functionalities are performed by the system. In analysis, design, implementation and testing, so many useful methods, techniques and methodologies can be learned. This project is aimed at getting a better understanding of the general aspects of the software development process and exploring some widely used techniques in the software development.

This report is divided into seven parts. Chapter one is the introduction of the project. Chapter two provides a whole picture and the features of the CMS system. Chapter three, four and five are system requirements, designs and database design. Chapter six is system testing. The last part, chapter seven, points out the possible problems of the implemented system and what can be improved in future.

2 System Description

2.1 The System Components

The Conference Management System provides a number of independent services from Web based user interface through Internet. They include the following:

1) Authors can submit their personal information and papers to the CMS system through the Internet.
2) Program committees (PC) members can register their personal information with the web based user interface.
3) The system can automatically allocate the papers among the PC members who will be responsible to review the papers.
4) PC members can download the paper files that they are to review and send their reviews to the system through the web.
5) The system can help the General Chair (GC) to automatically send reminder email to the PC member.
6) The system can generate summery reports and tables on the web page to assist the GC to maintain the information for the conference through the Internet. The system classifies user privilege levels for authors, PC member and GC. Also, object-oriented design principle is used for easy extension of the system to meet the need of the future changes of information and their applications.

2.2 Software Used

The CMS is a 3-tier client/server application. HTML is used as the front-end of the server. The middleware is Servlet server and the back-end is MySql, which is the database management system. Also the project uses the JDBC to access the MySql database. Client side only needs web browser to run the client program. After server is set up, client side can run client program by accessing the following URL: http://cindi.cs.concordia.ca:88/forms/Confsys.html in the web browser.

2.2.1 Servlet

In this project, Servlet is one of the key components of server-side Java. A Servlet is a small, pluggable extension to a server that enhances the server's functionality. Servlets are commonly used with web servers, where they can take the place of CGI scripts. Servlets are Java technology's answers to Common Gateway Interface (CGI) programming. By comparing with traditional CGI, this section explains the reasons to choose Servlet as web extension instead of CGI. In part five I will describe how servlet works. Here, Servlet's advantages are introduced as follows:

- **Efficiency**

With traditional CGI, a new process is started for each HTTP request. If there are N simultaneous requests to the same CGI program, the code for the CGI program is loaded into memory N times. With servlets, the Java Virtual Machine stays running and handles each request using a lightweight Java thread, not a heavyweight operating system process. If there are N simultaneous requests to the servlets, however, there would be N

threads, but only a single copy of the servlet class. Finally, when a CGI program finishes handling a request, the program terminates. This makes it difficult to cache computations, keep database connections open. However, servlets remain in memory after servlets are started, even after they complete a response, so it is straightforward to store arbitrarily complex data between requests.

- **Convenience**

Servlets have an extensive infrastructure for automatically parsing and decoding HTML form data, reading and setting HTTP headers, handling cookies, tracking sessions, etc.

- **Powerfulness**

Servlets support several capabilities that are difficult or impossible to accomplish using regular CGI. Servlets can talk directly to the web server, whereas regular CGI programs cannot, at lease not without using a server-specific API. Communicating with the web server makes it easier to translate relative URL into concrete path names, for instance. Multiple servlets can also share data, making it easy to implement database connection pooling and similar resource-sharing optimizations. Servlets can also maintain information from request to request, simplifying techniques like session tracking and caching of previous computations.

- **Portability**

Servlets are written in the Java programming language and follow a standard API. Consequently, servlets written for, say, Enterprise Server can run virtually unchanged on

Apache, Microsoft Internet Information Server (IIS), IBM WebSphere, or StarNine WebStar.

- **Security**

One of the main sources of vulnerabilities in traditional CGI programs stems from the fact that it is often executed by general-purpose operating system shells. So the CGI programmer has to be very careful to filter out characters such as back quotes and semicolons that are treated specially by the shell. This is harder than one might think. A second source of problems is the fact that some CGI programs are processed by languages that do not automatically check array or string bounds. So programmers who forget to do this check themselves open their system up to deliberate or accidental buffer overflow attacks. Servlets suffer from neither of these problems. Even if a servlet executes a remote system call to invoke a program on the local operating system, it does not use a shell to do so. And of course array bounds checking and other memory protection features are a central part of the Java programming language. A server can further protect itself from servlets through the use of a Java security manager.

- **Elegance**

Servlets code is clean, object-oriented, modular, and amazingly simple. One reason for this simplicity is the Servlets API itself, which includes methods and classes to handle many of the routine chores of servlets development. Even advanced operations, like cookie handling and session tracking, are abstracted into convenient classes.

2.2.2 JDBC

JDBC (Java Database Connectivity) is an Application Programming Interface (API) that is industry standard for database-independent connectivity between the java programming language and a wide range of databases. It consists of a set of classes and interfaces written in Java that allow the programmer to send SQL statement to a database server for execution and in case of Sql query to retrieve query results.

Why did I use Servlets and JDBC to develop the CMS web application? Because Servlets, with their enduring life cycle, and JDBC, a well-defined database-independent database connectivity API, are elegant and efficient solution for hooking web sites to back-end databases. So we started working with servlets specifically because of this efficiency and elegance. Writing database applications in Java using JDBC has at least two advantages: 1) portability across database servers and 2) portability across hardware architecture. The portability across database severs is a consequence of the JDBC API. The value of the JDBC API is that an application can access virtually any data source and run on any platform with a Java Virtual Machine. In other words, with the JDBC API it isn't necessary to write one program to access a Sybase database, another program to access an Oracle database, another program to access an IBM DB2 database, and so on. One can write a single program using the JDBC API, and the program will be able to send SQL or other statements to the appropriate data source. The JDBC drivers take care of the server dependencies, and the applications written in Java using JDBC are independent of the database server. The portability across hardware platforms is a result of the Java language; one doesn't have to worry about writing different applications to

run on different platforms. Hence, the combination of Java and JDBC to develop database applications is an ideal match, as it is possible for the applications to be written once and run anywhere.

The Java programming language, being robust, easy to use, easy to understand, and automatically downloadable on a network, is an excellent language basis for database application. What is needed is a way for Java application to talk to a variety of different data sources. JDBC is the mechanism for doing this.

2.3 Operation of CMS

Conference Management System allows users to interact with the system via Internet. Server program, running on web server before any client program starts, acts as middle layer between a request coming from a Web browser or other HTTP client and database. When a user types a URL on a address line, follows a link from a Web page, or submits an HTML form, a browser generates the GET or POST request for the web pages. The server will read any data sent by the user and look up any other information about the request that is embedded in the HTTP request. This information includes details about browser capabilities, the host name of the requesting client, and so forth. When server receives the data from the client, it will generate the results. This process may require talking to a database or computing the response directly and then format the results inside a document. In most case, this involves embedding the information inside an HTML page. Before sever sends the document back to the client the appropriate HTTP response parameters must be set. This means telling the browser what type of document is being

returned (e.g., HTML), setting caching parameters, and other such tasks. Finally server sends the document back to the client. In most cases, this document may be sent in text format (HTML).

In this project, Java language is chosen. Servlets and JDBC are all written in the Java programming language and follow a standard API. The cross-platform nature of Java is extremely useful for the CMS server running various flavors of the Unix and Windows operating systems. Java's modern, robust, secure, easy to use, object-oriented, memory-protected design allowed me to cut development cycles and increase system reliability. In addition, Java's built-in support for networking and enterprise APIs provides access to legacy data, easing the transition from older client/server systems. This made the CMS system easy to change. Also the combination of the Java platform and JDBC technology makes disseminating information easy and economical. CMS can continue to use their installed database and access information easily even if it is stored on different database management systems or other data sources. From the above introduction, we can see using Servlets, JDBC and Java makes CMS system more powerful, efficient, portable, safe and flexible.

3 System Requirement Definition and Specification

3.1 Product Requirement

Conference management, especially for large international academic conferences, is a quite complicated task and needs a lot of human resource input. The people involved in a conference generally include authors of papers, program committee, general chair etc. The conference management system needs to provide a variety of functions that manipulate input, output and modification of information in the system. Moreover, the system should be able to support multi-user concurrent access and has distinct interfaces for the different tasks.

The users of the system are divided into three groups according to their privilege levels:

- Author
- Program Committee (PC) member
- General Chair (GC)

3.2 Functional Requirements Definition and Specification

3.2.1 Paper Submission

Paper registration is for the paper's authors. Through it authors can submit papers and other information to the server. The following are the service provided:

- Submission

  Paper information ---paper title, paper file name, paper's abstract.

Authors information ----author name, author title, author's email.

- Uploading Paper File

Choose a paper file and upload it.

- Input

Authors input their name, title, email and paper 's title, abstract, and submit the file.

- Output

When an author clicks the Submit button, a new window will popup, which acknowledges the information the author has entered. This window also contains a form that allows the author to enter the co-author information (see figure 3-2). If the paper was written by only one author then the author can click the 'Finish' button to finish the input.

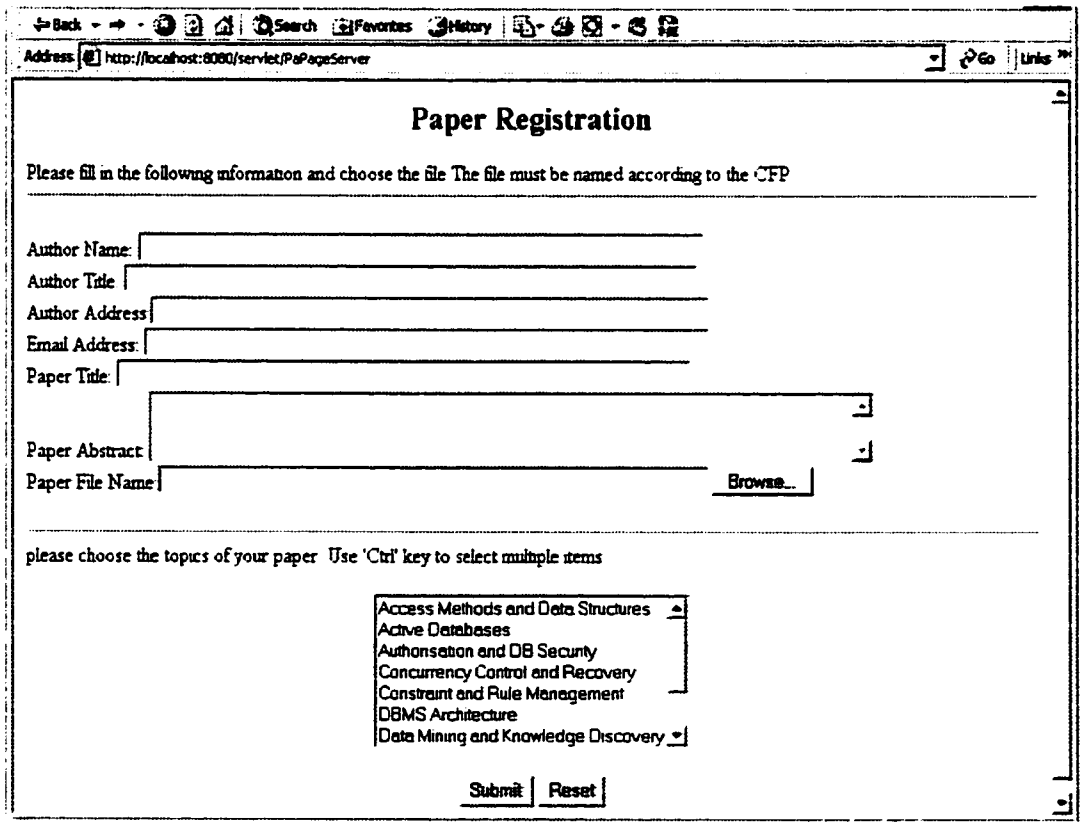


Figure 3-1Paper Registration

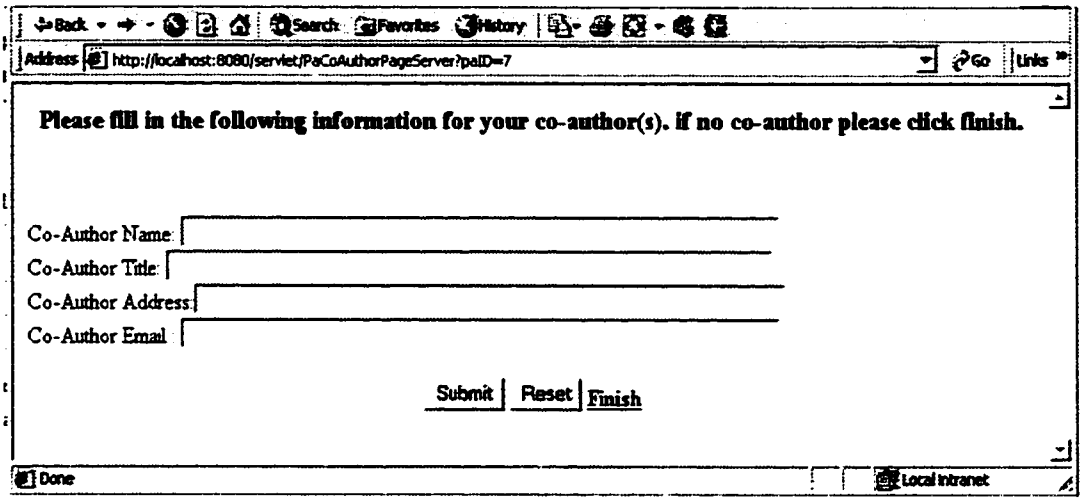


Figure 3-2 Co-Author Information

3.2.2 PC Workstation

This service is provided for the Program Committee (PC) member. A valid ID and a password are required to access the site. The page provides the following services:

- PC Registration
- PC Download File
- PC Review

• Input

• Output

When a PC member chooses an option, a new window will pop up, which contains the page linked by the option.

• Constraints

All these options can be accessed only by PC member and GC.

Each task for the PC member is explained as follows:

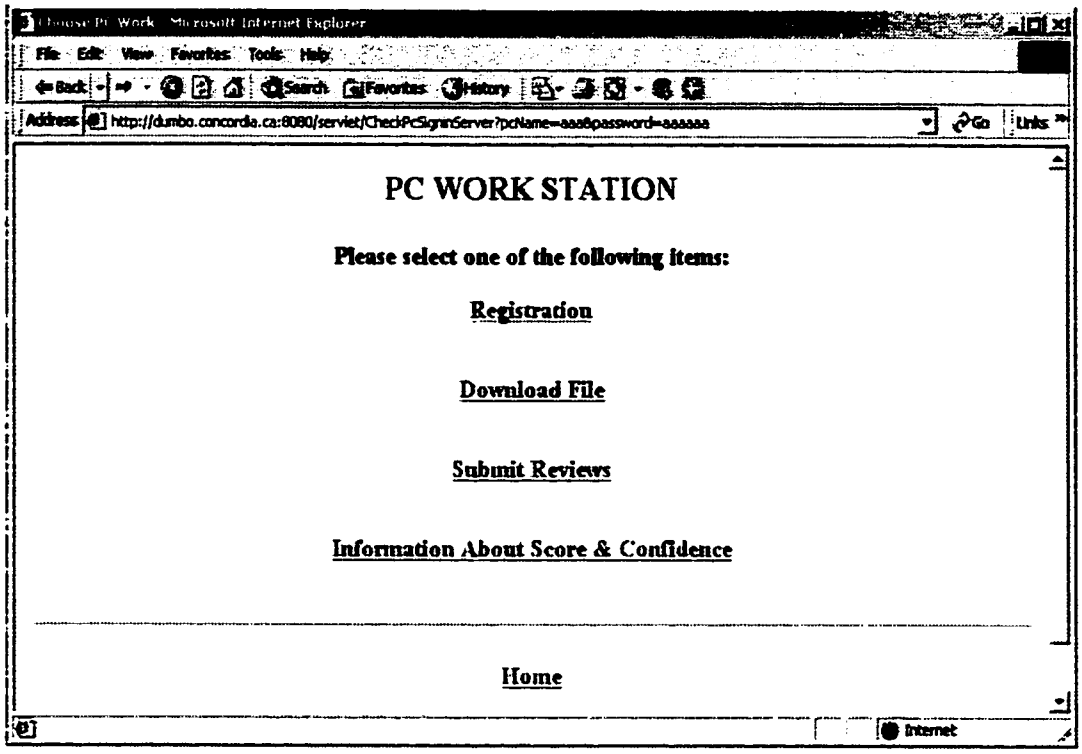


Figure 3-1 PC Workstation

Registration by PC member

The services provided to PC member are:

- Enter user's information for the system.
- Input

  Input user's ID number, name and organization into the form.

- Output

  When an user clicks the Submit PC member Information button, a new window will pop up, which acknowledges the information entered.

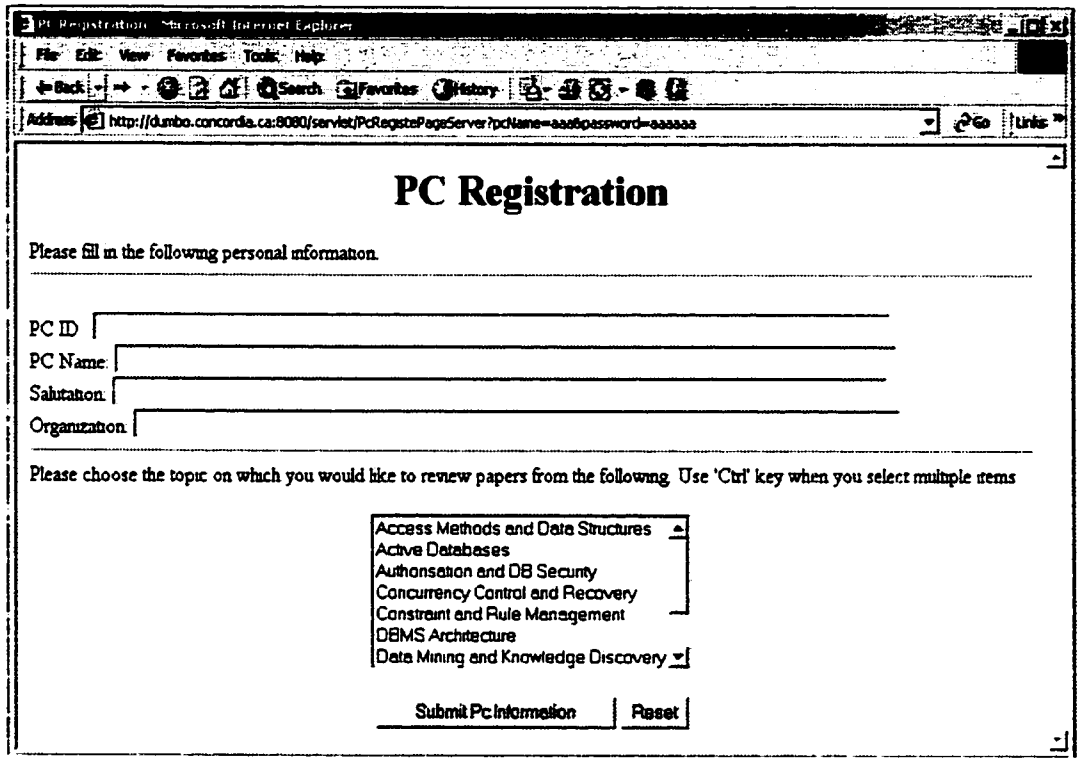


Figure 3-2 PC Registration

Download File by PC member

From this page PC member can download all the files to be reviewed. The services provided are:

- Show all the topics of interest to the PC member and all the files to be reviewed.
- Input

  User can click on one of the files to download it.

- Output

  Show the files that can be downloaded.

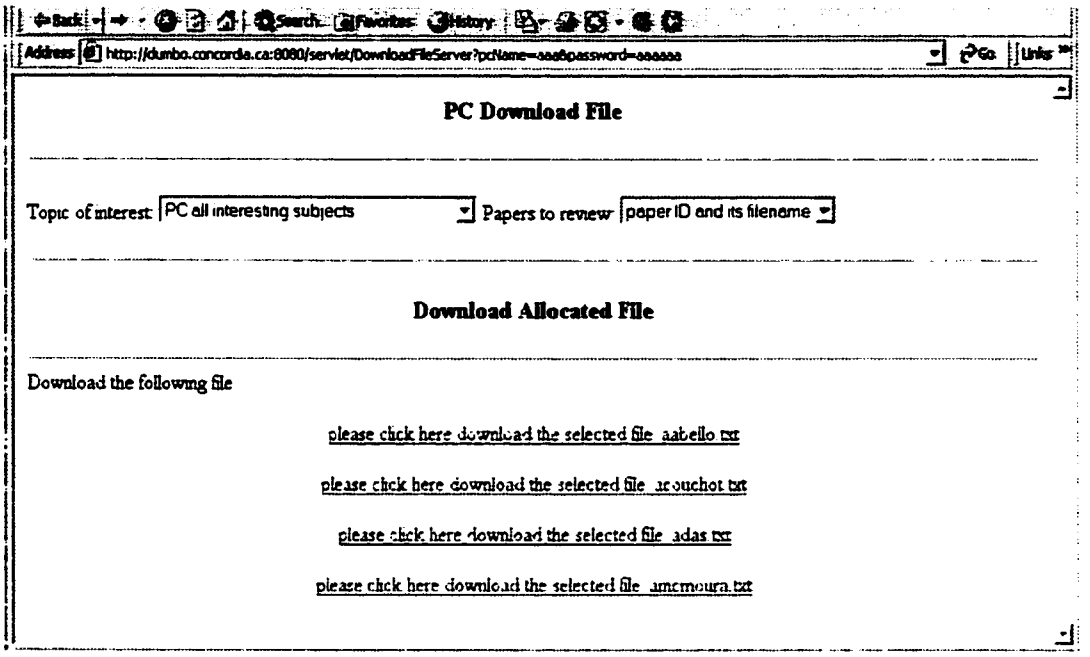


Figure 3-3 PC download file

Review by PC member

This web page provides the following:

- Input the information about each paper: the score, confidence and evaluation of the papers that is reviewed by the PC member.

• Input

PC member has to input paper's score, confidence and evaluation to the form.

Information about Score and Confidence

The service provided PC member information about how to give the score and confidence of the reviewed papers.

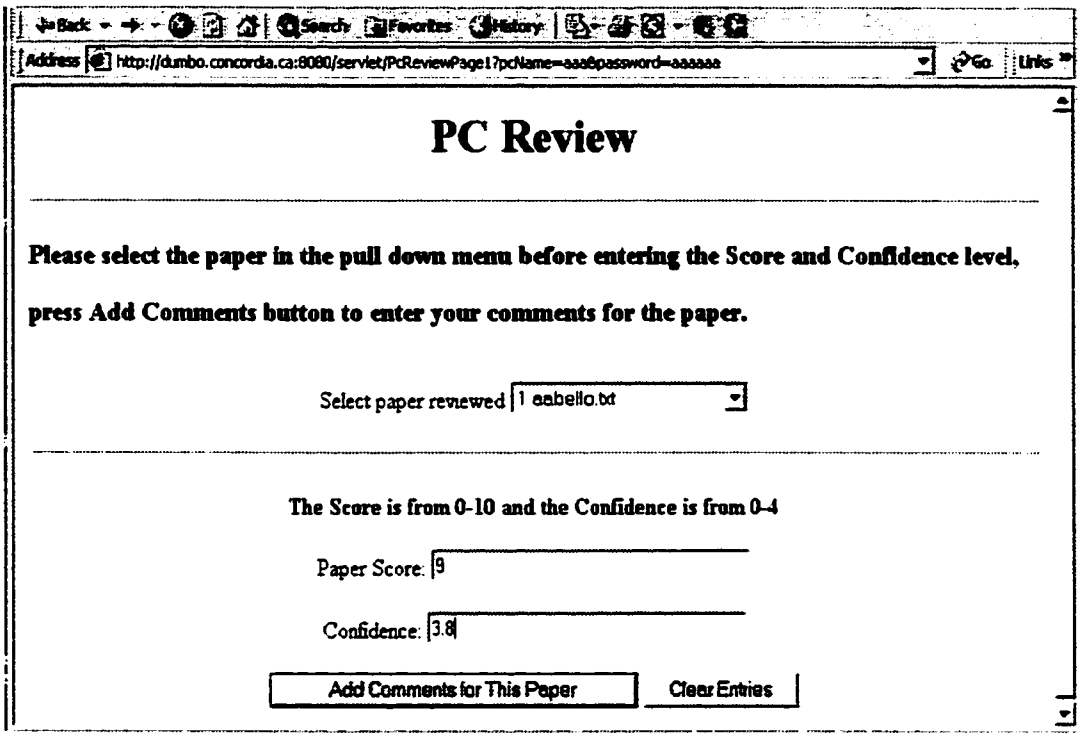


Figure 3-4 PC Review page 1

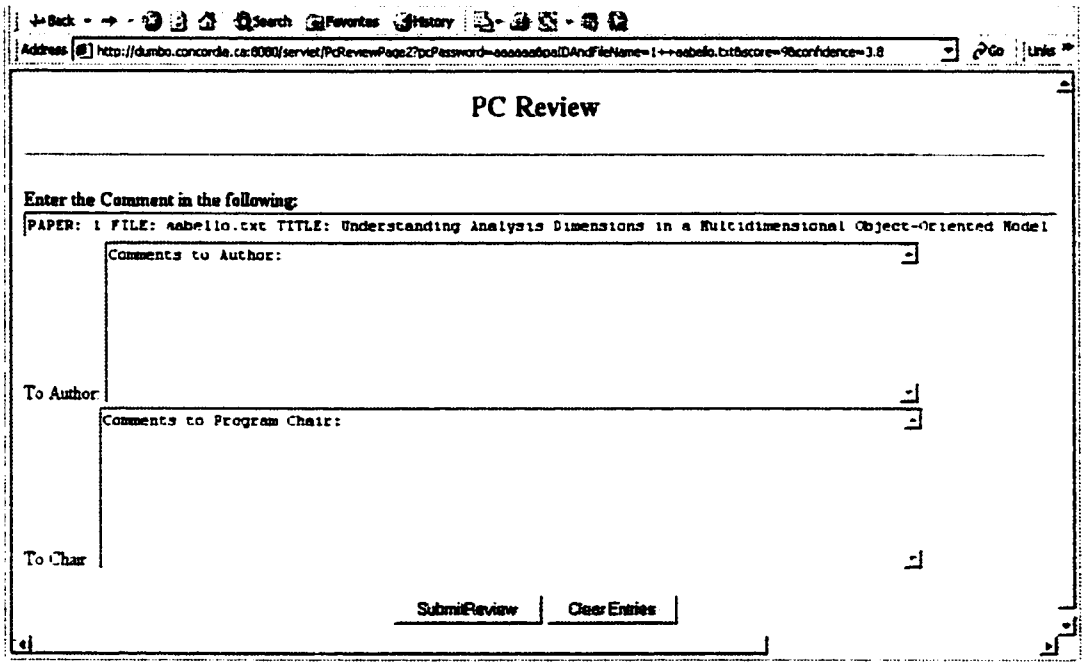


Figure 3-5 PC Review page 2

3.2.3 GC Workstation

This service is for the general chair (GC). A Valid ID and a password are required for accessing this page. The services includes the following:

- Input data
- Send Email
- Allocate papers
- Examine paper's reviews
- Output

  When the GC chooses an option, a new window will pop up, which contains the page corresponding to the option.

- Constraints

  All the options are accessible only to the GC.

Each task the GC does is explained as follows:

Input Data

This service for GC manipulates the database and put information directly into the database. The services include the following:

- Create the database elements such as tables
- Input PC member information
- Input Topic Name of the conference
- Update GC name, password, and email address
- Output

Whenever GC finishes a task the corresponding feedback information will be displayed.

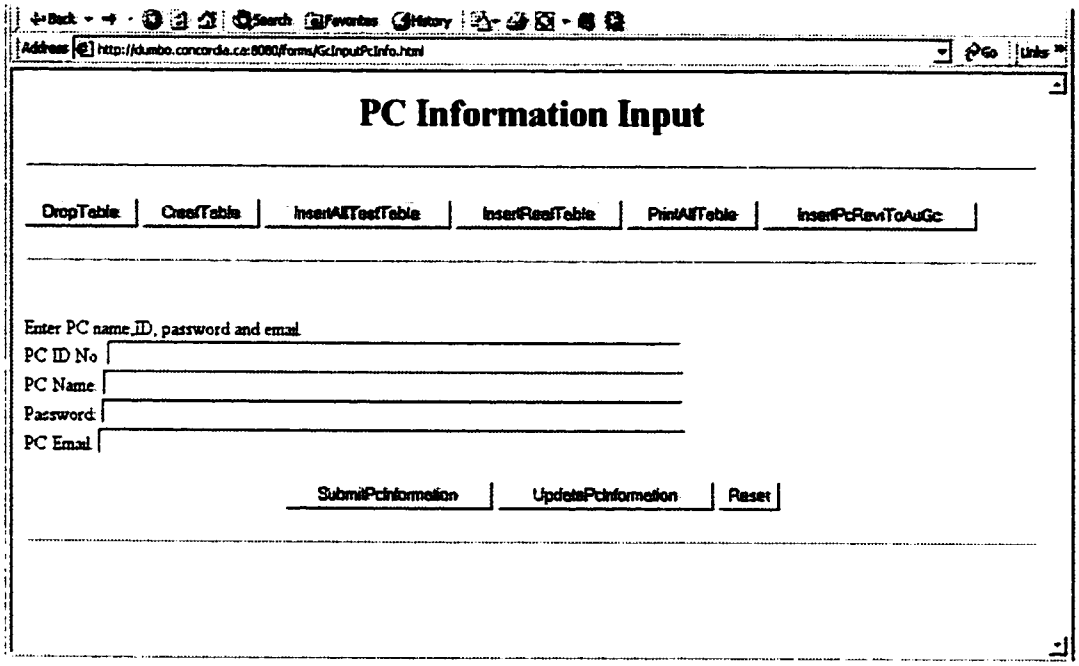


Figure 3-1 PC Information Input

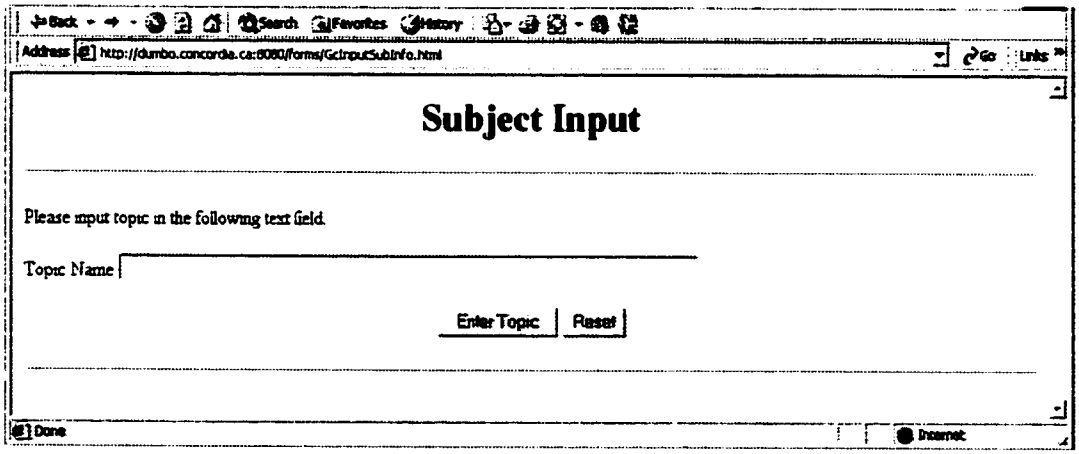


Figure 3-2 Subject Input

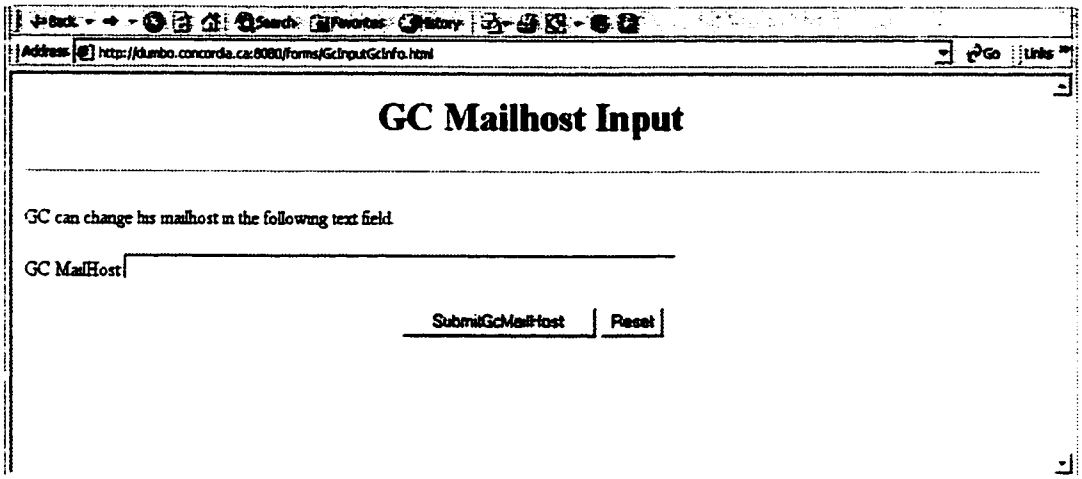


Figure 3-3 GC information Input

Send Email

This service allows the GC to send many types of email automatically. The services include the following.

- Send an email to PC members – giving their ID and password
- Send a message to PC members to remind them to submit topic of interest
- Send a message to PC members (who have not submitted all the reviews) to remind them to submit review for papers
- Send a message to authors
- Input

  For each email to be sent some related information must be input as instruction.

- Output

  Whenever GC finishes some task the corresponding feedback information will be displayed.

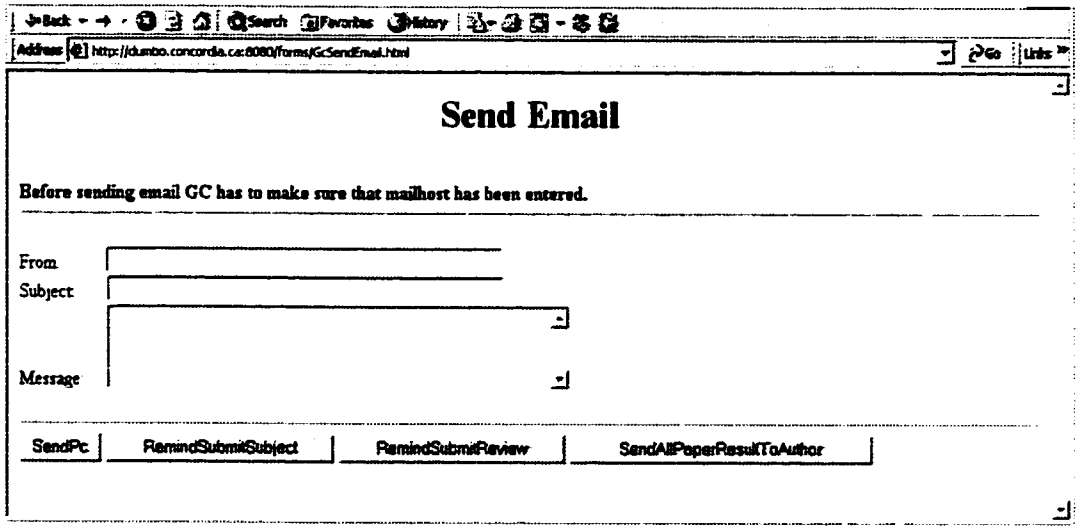


Figure 3-4 Send Email

Allocate Paper

This service allows GC to automatically allocate paper among the PC members. This service includes the following.

- Choose allocate rule
- Allocate paper
- Delete allocated paper from a given PC member
- Choose a new 'At Least' and 'At Most' parameters values to Reallocate paper
- Show all tables in the database
- Input

  Before starting allocate, GC must input the lower and upper number of papers that PC members review.

- Output

Whenever GC finishes some task, the corresponding feedback information will be displayed. For some papers that cannot be allocated, the related information will be displayed.

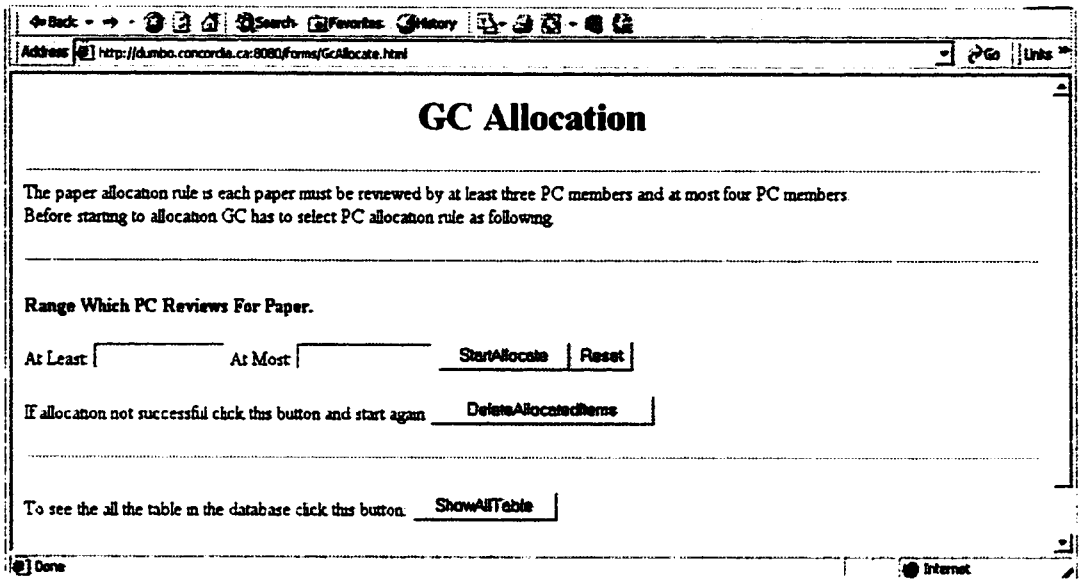


Figure 3-5 Allocate Paper

Examine Paper's Review

This service allows the GC to see the reviews written by PC members and allows GC to add his own review for authors. The services include the following.

- Show overview of all papers
- Submit GC comments to the database
- Input

  GC has to input the each paper's review result to the form and then send it to the authors.

  - Output

When GC click the button 'show overview all paper information', then all the paper', (What happens next?)

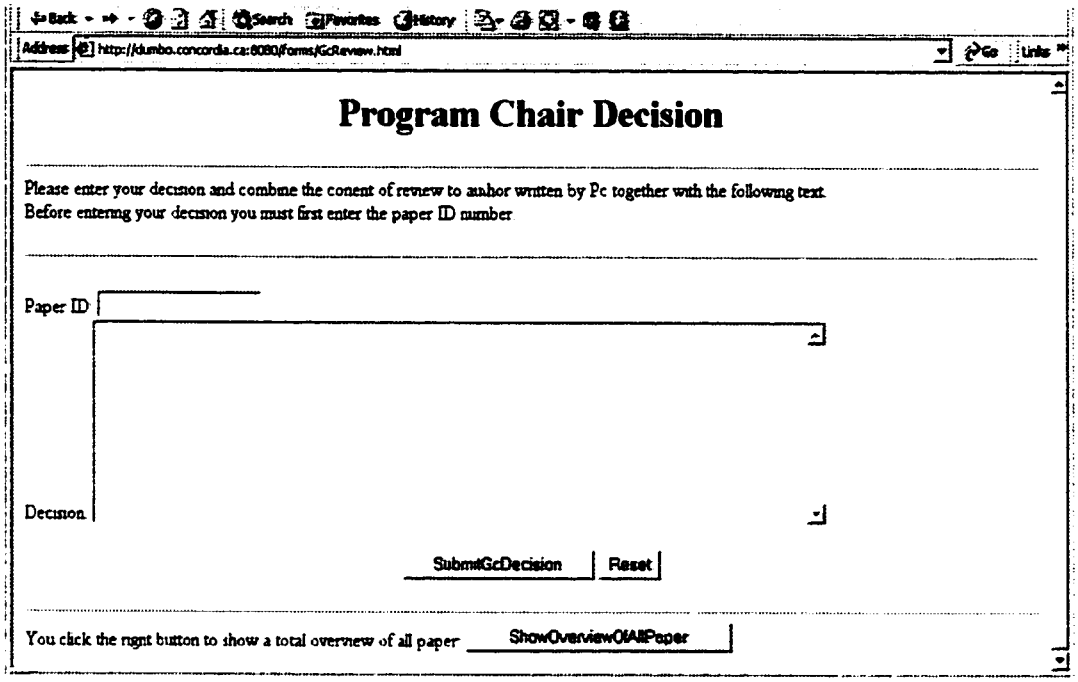


Figure 3-6 Review Paper

3.3 Non-Functional Requirements

Non-functional requirements define system properties and constraints. The properties, for example, are reliability and response time. An example of constraints is the capability of the I/O device attached to system and the data representation used by other systems connected to the required system.

The non-functional requirements of CMS can be classified into three categories: product requirement, organization requirement and external requirement.

3.3.1 System Requirement

Computer Hardware and software requirements

The system should perform all its functionality efficiently on the following platforms:

Web server: any web server that supports Java Servlets

Operating system: Windows NT/Window 98/Linux

Servlet Software: JSWDK

(http://java.sum.com/products/servlet/download.html)

File Upload Program: com.oreilly.servlet package

(http://www.servlets.com/resources/index.html)

DBMS (Database Management System): MySql

End user:

In general, any computer, which can use graphical web browser through the Internet, can access the system.

Performance Requirements

The system should support user to access the servlets concurrently and allow many users access at the same time. System should work correctly and efficiently. The number of users is restricted by database management system and operating system.

4 System Design

4.1 Design Rationale

The characteristics of Conference Management System application includes

- Web based application

  The main users of the CMS system, PC members, GC and authors, apply web browser to exchange conference relevant information through internet/intranet.

- Client/server model

  The clients of CMS system are distributed worldwide and the physical location of the system server is also separated from the clients. The server should be ready to respond to the clients' request at any time. Server side also supports multi-user access concurrently.

- Implementation of web server and database

  CMS system needs web server hosted on World Wide Web and at least one database to store quite large amount of conference information about papers and their abstracts, authors, PC members, etc. It is better to modularize the web server and database separately.

4.2 System Architecture

CMS system applies the Three Tier Application Model. It includes

- Front tier - provides presentation services.
- Middle tier - provides business services.
- Back tier – provides data services.

Figure 4-1 illustrates the overview of major components of CMS system architecture.

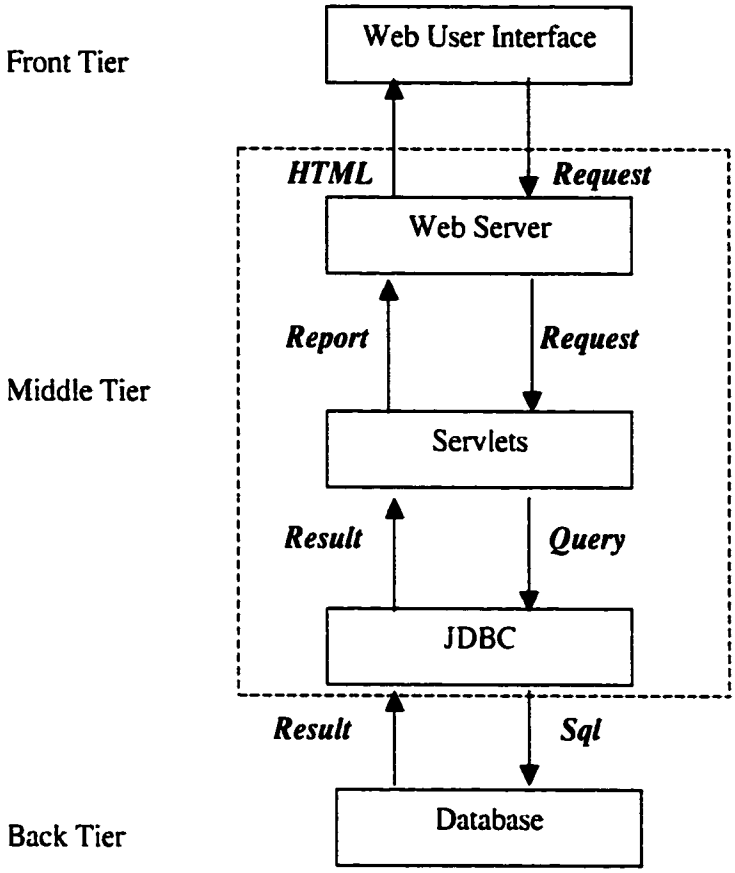


Figure 4-1 Three-Tier Diagram of CMS Architecture

The various components of the system identified by their functions are:

- Web User Interface: This is the front tier of the system that uses HTML format to provide system users the graphical user interface. It performs functions of displaying message, inputting request, downloading/uploading files and so on.
- Web Server: This is a host of HTML files. It accepts the request from the users, communicates with servlets and responds to the requests with information using HTML or through servlets.

- Servlets: This is the central part of the middle tier that integrates the core functions of the whole system, such as request/response mechanism, information processing algorithms, and generating data operation commands to the data module in the back tier. They also respond to and process events from web server.
- JDBC: It provides internal interface between servlets application programs and Database.
- Database: Stores and manages the system information.

According to the above three-tier model, CMS system is divided into three subsystems: user interface subsystem, server subsystem and database subsystem.

The following figure gives the organization details of the CMS system.

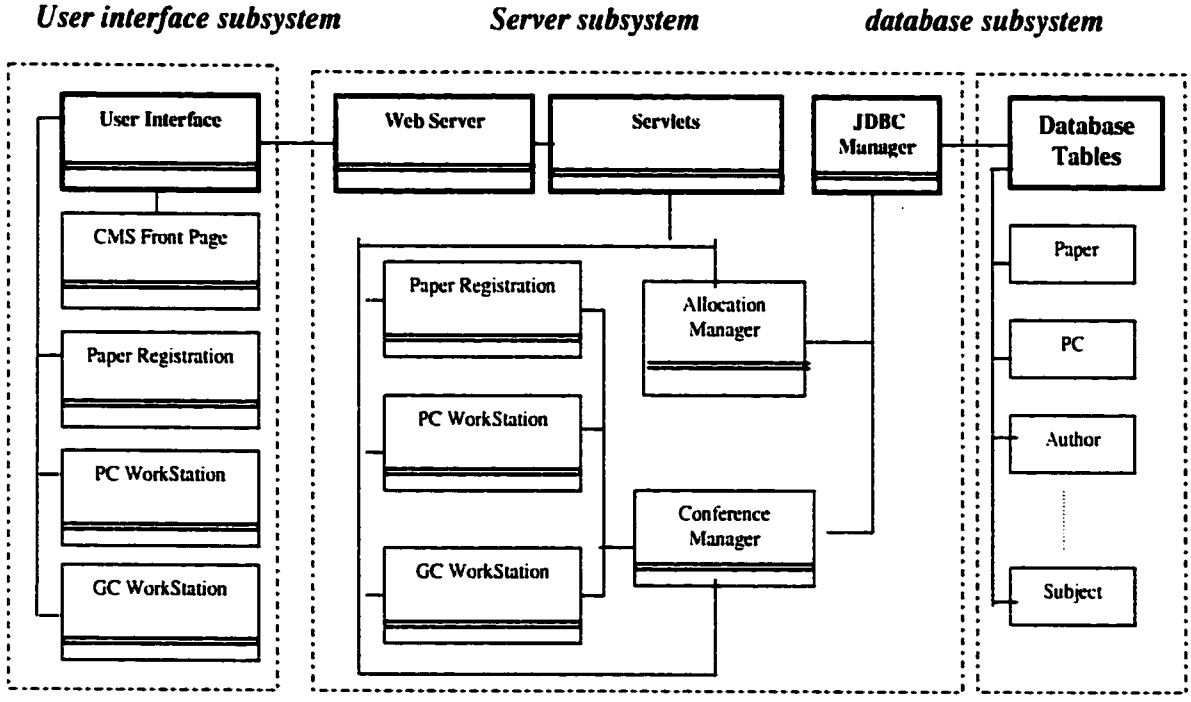


Figure 4-2 Conference Management System Architecture

This section mainly discusses user interface subsystem and server subsystem. Database subsystem will be discussed in the next section.

4.2.1 User interface subsystem

User interface subsystem includes three main components: Paper Registration, PC member WorkStation and GC WorkStation. The main linking hierarchy of the HTML files is shown in figure 4-3.

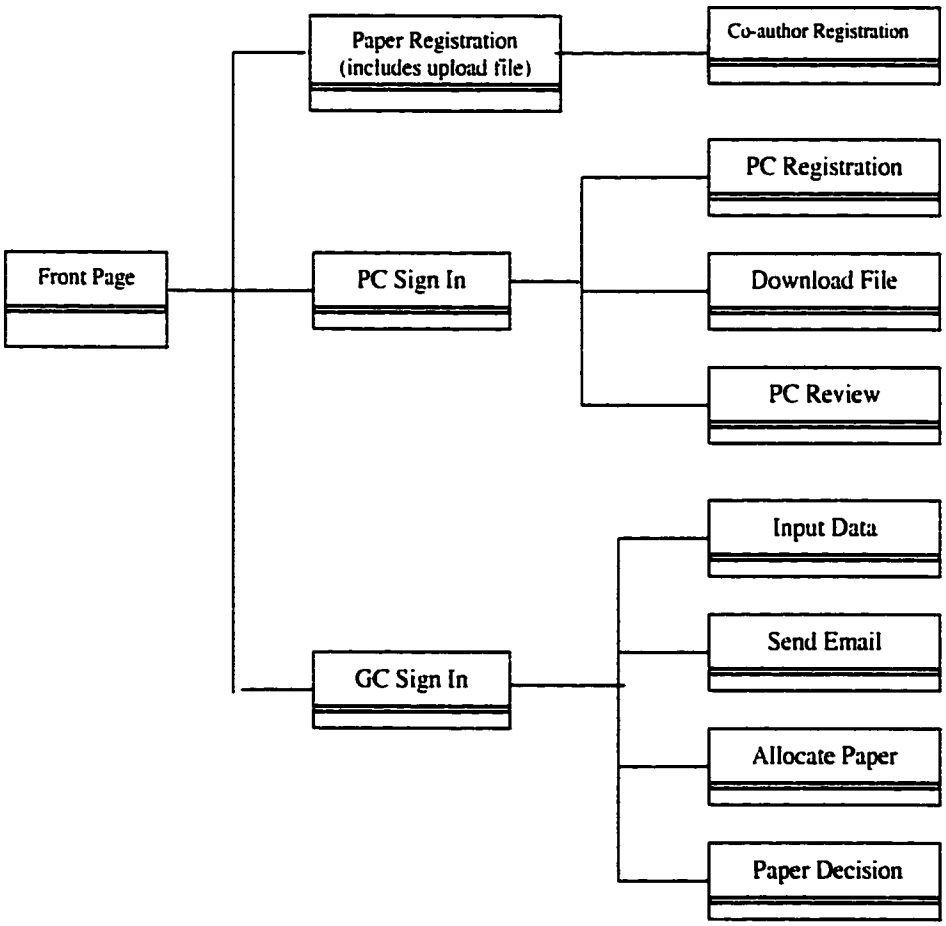


Figure 4-1 Interface Subsystem

4.2.2 Server subsystem

Server subsystem consists of three sections: web server, servlets and JDBC. Figure 4-4 displays its component details.

Servlets is responsible for handling user input, which includes interpreting user input and passing information and instruction between the web server and JDBC within server subsystem. When a client sends a request to the web server, it dispatches a request to a servlet by invoking the servlet 's doGet() or doPost() method to handle the event. When a client needs to retrieve some information from the database, the servlets sends a request to the database subsystem, and then transfer the required data to the user interface subsystem through web server. So in server subsystem servlets play a key role on the server side.

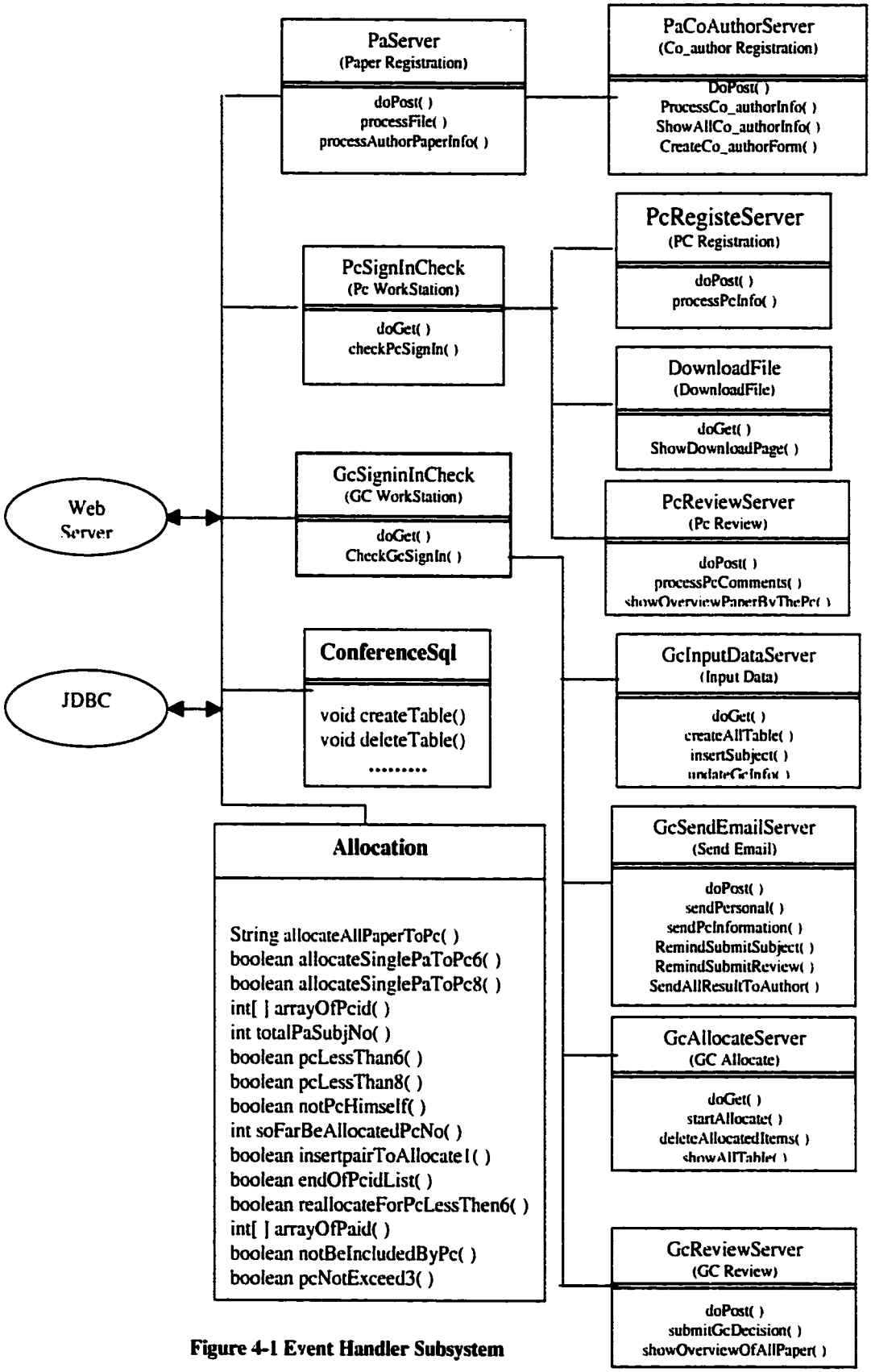


Figure 4-1 Event Handler Subsystem

4.2.2.1 Internal module description

Paper Registration

- createPaperPage(): creates a page and lets authors register their papers and other information.
- createSubjectListBoxInPaperPage(): creates a list box of topics in the paper page.
- processFile(): checks whether file name is duplicated, and if not, stores the file and displays information for the file to the author.
- processAuthorPaperInfo(): receives data from the author and stores it in the database. Then displays the feedback information to the author.
- processCo_authorInfo(): checks input data and stores the co-author information to the database.
- showAllCo_authorInfo(): displays all the authors information for the submitted paper.
- createCo_authorForm(): creates a form to input co-author information.

PC WorkStation

- checkPcSignIn(): checks PC member's password and name.
- createPcRegistePage(): creates a page for PC member to register information.
- processPcInfo(): gets the PC member input information and stores it to the database.
- showDownloadPage(): creates a list of file which can be downloaded by the PC members.
- pageForScoreAndConf(): creates a page for PC member to enter paper's score and confidence in the corresponding fields.

- createPcCommentsPage(): creates a page to let PC member input the paper's comments on it and obtains the paper's score and confidence before storing them into the database.
- processPcComments(): receives the PC member's comments before storing them into the database.
- showOverviewPaperByThePc(): displays the information of the papers PC member reviewed.
- showPcComments(): displays comments made by the PC member.

GC WorkStation:

- checkGcSignIn(): checks GC's password and name.
- createAllTable(): creates all the tables in the database.
- submitPcInfo(): receives PC member's information and then storse them into the database.
- insertSubject(): extracts the topic from the web page then stores into the database.
- updateGcInfo(): updatse the GC's information in the database.
- sendPersonal(): sends emails.
- sendPcInformation(): sends a password and ID number to the PC members.
- remindSubmitSubject(): sends an email to PC members who haven't submitted topics of interest.
- remindSubmitReview(): sends an email to PC members who haven't submitted their reviews.

- sendAllResultToAuthor(): sends an email to all authors about result of paper's review.
- startAllocate(): assigns papers to PC members according to some rule.
- deleteAllocatedItems(): deletes all allocated items in the database.
- showAllTable(): on the web page, displays all the tables in the database.
- submitGcDecision(): stores the GC's evaluation of the paper into the database.
- showOverviewOfAllPaper(): displays all the information for all papers on the web page.

Allocation:

- allocateAllPaperToPc(): assigns all the papers to the PC members according to some rule.
- allocateSinglePaToPc6(): assigns one paper to the PC members according to some rule.
- arrayOfPcid(): finds out array of pcid which contain topics that the paper contains.
- totalPaSubjNo(): finds out total topic number that the paper has.
- pcLessThan6():checks whether the number of papers that PC member was allocated is less than the value of parameter pcAtLeasetReviewPaNo.
- pcLessThan8():checks whether the number of papers that the PC member was allocated is less than the value of parameter pcAtMostReviewPaNo.
- notPcHimself():check whether the author of the paper is the PC member who is going to be allocated for the same paper.
- soFarBeAllocatedPcNo(): finds out number of PC member the paper was allocated.

- insertPairToAllocate(): inserts allocated pair of "paid" and "pcid" to the table which allocates and increments 'total_pa' and 'total_pc' varibles respectively.
- endOfPcidList(): checks whether the array of pcid reaches the end.
- reallocateForPcLessThen6():reallocates papers to the PC member, whose allocated number of paper is less than the value of parameter pcAtLeaseReviewPaNo.
- arrayOfPaid(): finds out the array of paper IDs that contain the topics the PC member has.
- notBeIncludeByPc(): finds out papers which have been allocated to the PC member.
- pcNotExceed3(): checks whether the paper's allocated total PC member number is less than 4.

ConferenceSql:

All the method names of this class indicate their functions directly.

4.3 System data flow

The previous section describes the static picture of what CMS system and its components include. This section will explore how the data or message moves within CMS system and the interactions between its subsystems and subsystem components in the instance of different user task scenarios.

4.3.1 Paper Registration Scenario

When a user selects the Paper Registration option, the browser generates a request and sends the request to the server. Then the web server dispatches a request to a servlet,

PaPageServer, which is a java class (Figure 4-5). The web server invokes doGet() method of PaPageServer class with parameters of an object of HttpServletRequest class and an object of HttpServletResponse class. The doGet() method will call createPaperPage() method of the same class to create the paper register page for authors to register paper and input information. Also createSubjectListBoxInPaperPage() method is invoked to create a paper topic list to let authors choose the topics of the paper.

After filling out the form, the author clicks the submit button, another servlet, PaServer's doPost() method is called and again, this doPost() method invokes the processFile() method to accept the data and uploaded file from the author and saves it to the conference database. At last, the processFile() displays the received information to the author.

On another page, there is a form to input co-author information. The author will enter all these co-authors information. The co-authors information will also be stored in the database by invoking the methods of ConferenceSql class.

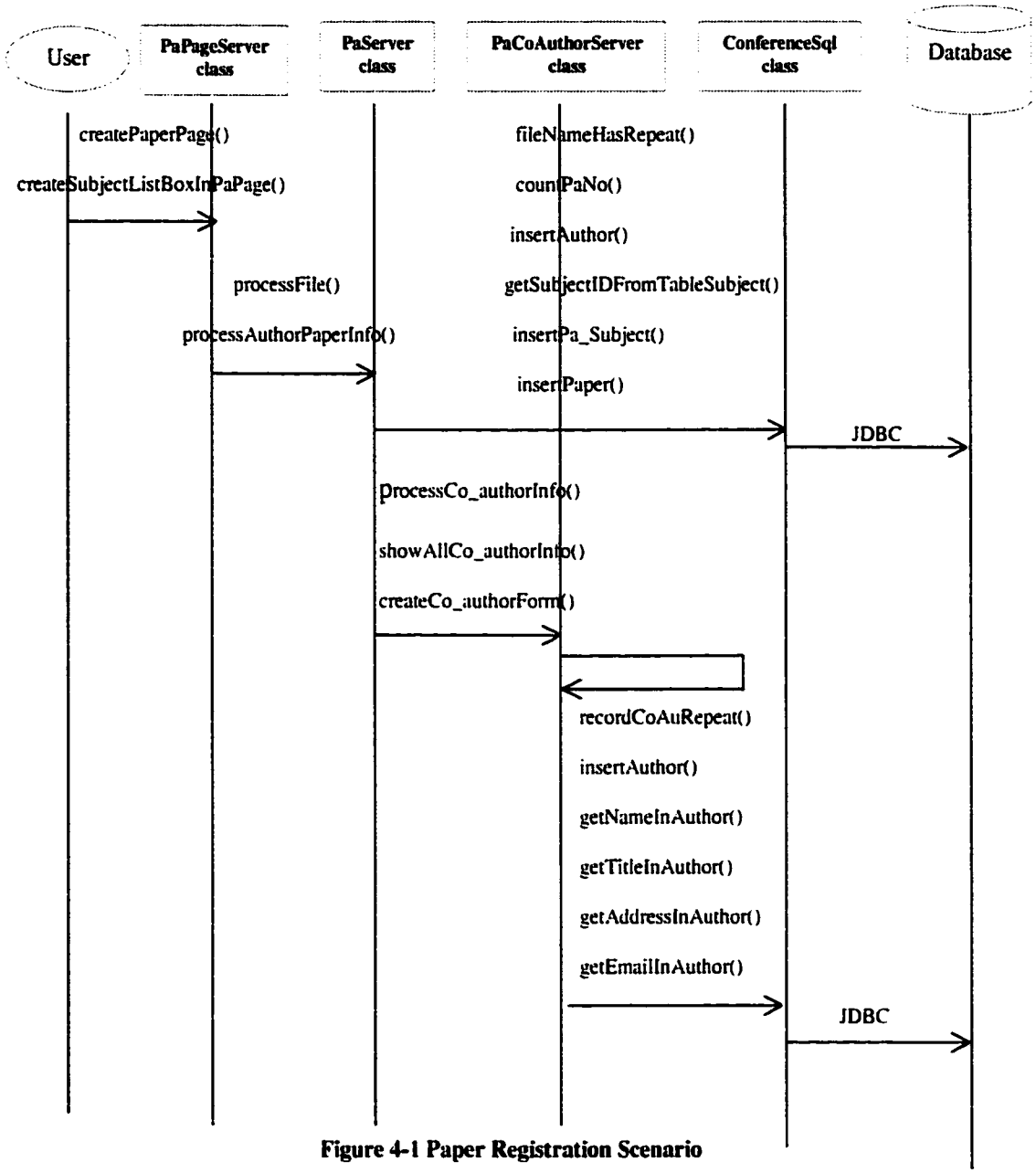


Figure 4-1 Paper Registration Scenario

4.3.2 PC WorkStation Scenario

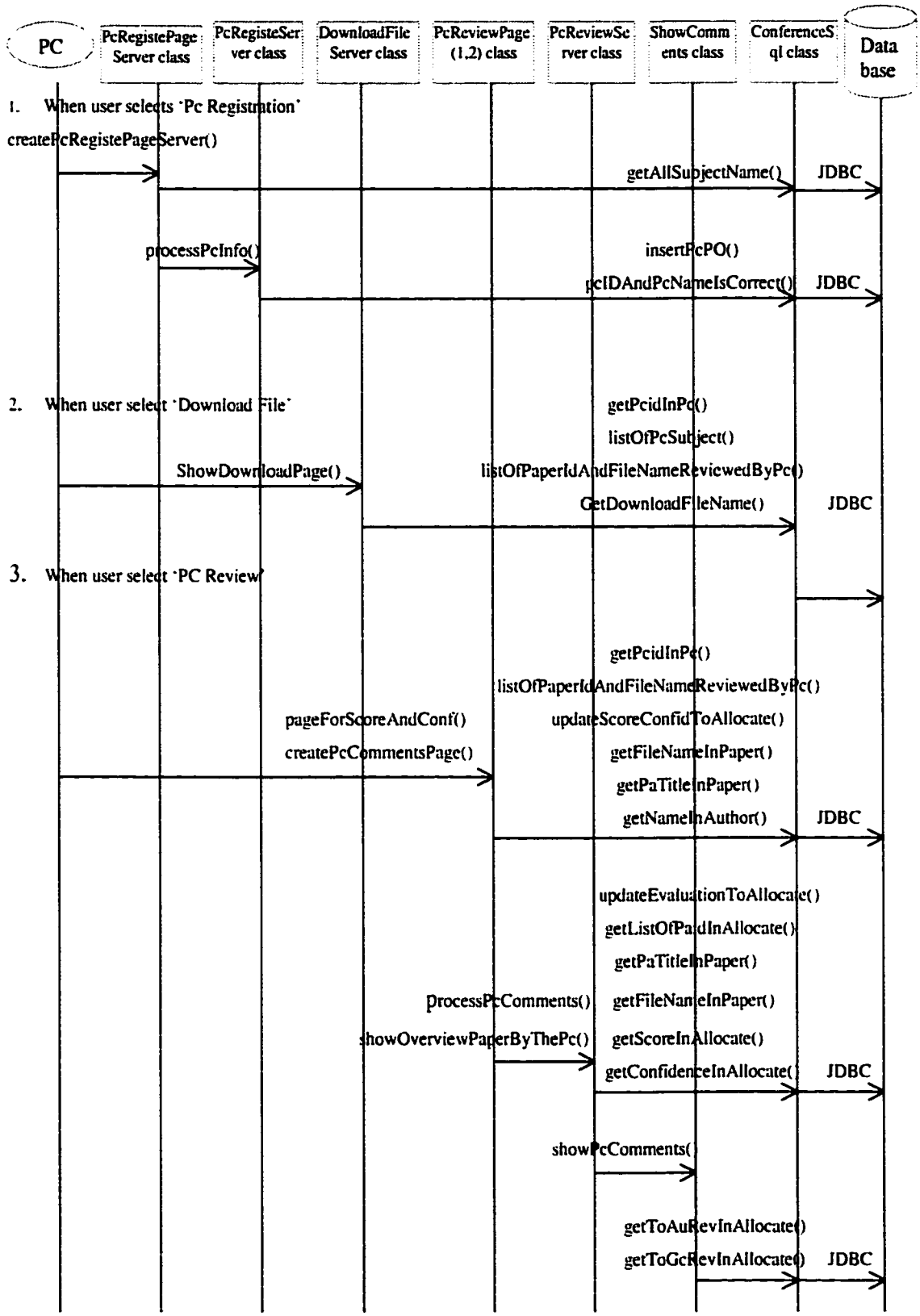


Figure 4-1 PC WorkStation Scenario

The user who selects the PC WorkStation option must first login with correct username and password before entering the PC working environment. Once the user provides the username and password and clicks the 'sign in' button, the servlet, CheckPcSigninServer class, is invoked to check the username and password. The next page will display only when the correct username and password is input.

When PC member selects the 'PC Registration', an object of PcRegistePageServer class is called to create a web page to let PC member enter his/her personal information. After 'submit Personal information' button is clicked, an object of PcRegisteServer class is invoked to deal with PC member's information. This process includes receiving, storing and displaying data for the PC member by invoking processPcInfo() method of PcRegisteServer class.

When PC member select the 'Download File' option, an object of DownloadFile class is invoked to display the file names of paper that are to be reviewed by the PC member, the topics of interest for the PC member, and all files which can be downloaded by the PC member.

When PC member selects the 'PC Review' option, an object of PcReviewPage1 class is invoked. When PC member submits the paper's score and confidence, another web page which is to let PC member continues to enter comments on the paper. After that the object of PcReviewServer class will invoke the processPcComments() method to receive and store it in the conference database. If PC member clicks the

'showOverviewPapaByThePc' button, all the information such as scores and evaluation for all papers reviewed by the PC member will be displayed.

4.3.3 GC WorkStation Scenario

Like PC WorkStation, GC WorkStation option also has username and password check enforced by CheckGcSigninServer class.

When GC selects the 'Input Data' option, server will invoke GcInputDataServer's doPost() method. In this method, it calls createAllTable() method to create all conference tables in the database. It also calls method insertPcInfo(), inserSubject() and updateGcInfo() to let GC update database. Method insertPcInfo() inserts PC member's organization information to database; method inserSubject() inserts available topics of the conference to the topic table; method updateGcInfo() lets GC change his own username, password, organization and email address.

When GC selects 'Send Email'option, server will invoke GcSendEmailServer class to let GC set some parameters and automatically create the email text to be sent. It calls sendPersonal() method to send email to a specific person. It calls method sendPcInformation() to send PC members email to inform PC members of their PC ID number and password. It calls method remindSubmitSubject() and remindSubmitReview() to send email to reminds PC member submit her/his topics and paper's evaluations. It calls method sendAllResultToAuthor() to send email to all the authors who had submitted a paper through the conference system's web pages.

When GC selects 'Allocate Paper' option, sever will use GcAllocateServer class to assign the papers to member of the PC. If allocation is not successful, click the 'deleteAllocatedItems' button and deleteAllocatedItem() method is called. Then you can type in another group of parameters to reallocate. This web page also provides a 'ShowAllTable' button which will display all the tables in the database.

When GC selecst the 'Review Paper' option, server will invoke servlet GcReviewServer class's doPost() method. After finishing the paper's evaluation, GC clicks the button 'submitGcDecision' to call the method submitGcDecision to store the information to the database. Also GC can click the 'ShowOverviewOf AllPaper' button to see all the papers's ID number, title, file name, all the scores and confidences of specific paper, and the PC member's evaluation.

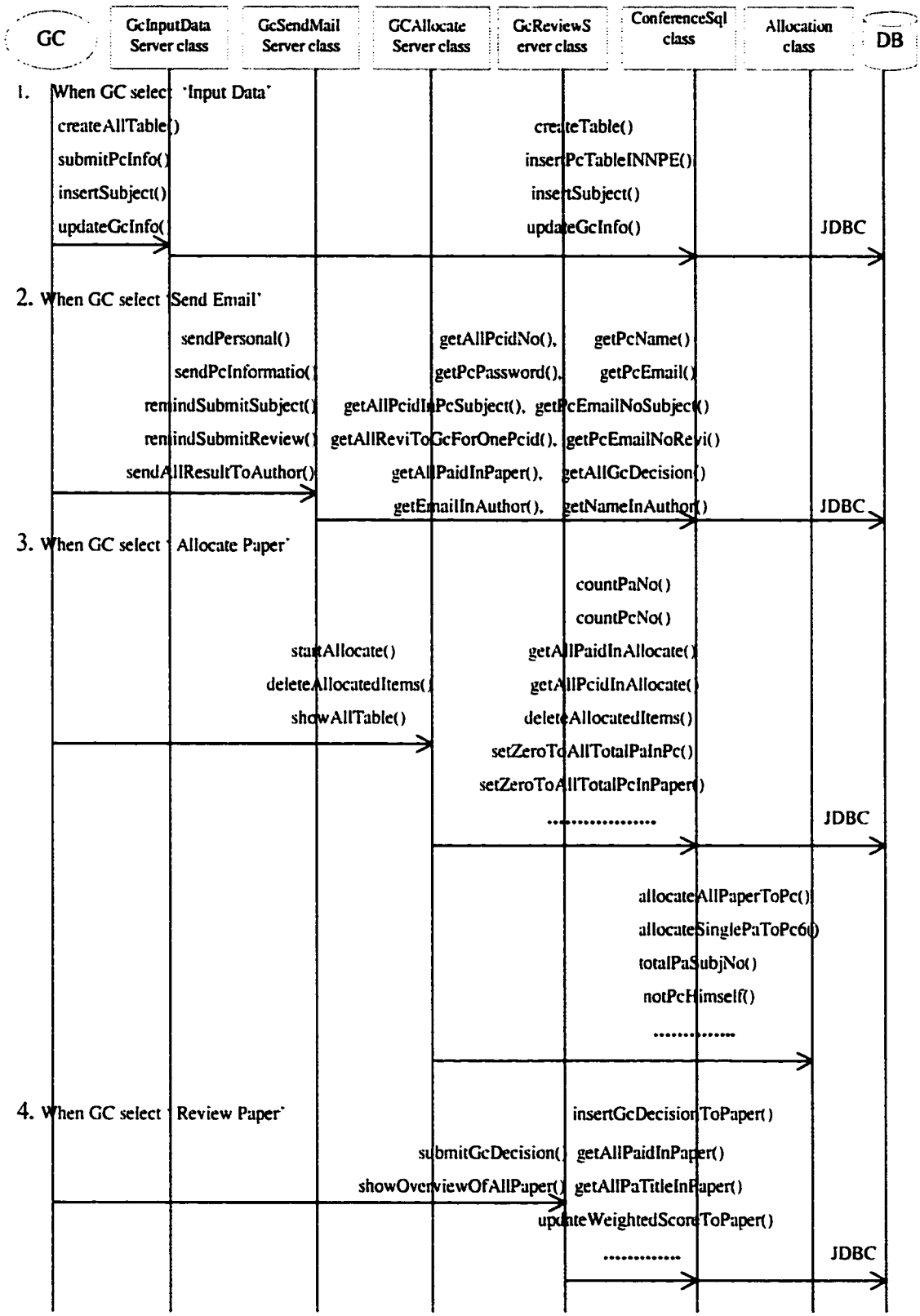


Figure 4-1 GC Workstation Scenario

5 Database Design

An effective information system must provide users with timely, accurate, and relevant information. There are alternative ways of organizing data and representing relationships among data in the database. Despite the use of excellent hardware and software, designing of database and choice a suitable database management system to launch and manipulate the oparations on the database are also critical issues.

5.1 Selection of Database Model

Ralational data model is currently a dominant database model in use. This model represents all data in the database as two-dimentional tables called relations. The relational data model presents a logical view of a database. This is a simple and intuitive view of data that hides all the complex details of how the data is actually stored and accessed within a computer. The information in more than one table can be easily extracted and combined using DBMS providing language.

5.2 Selection of Database Management System

In this project, MySql is chosen as the Database Management system. MySql is one of the relational database management systems to be used by small to medium applications. It is an implementation of the SQL-92 standard Structured Query Language, a universal language that can be used to define, query, update, and manage a relational database.

5.3 Database Design

5.3.1 Design assumptions

- General Chair is responsible for administrating the Conference Management System and database. GC's ID number is set 1.
- The meeting has more than one topics.
- Each paper may have one or more topics of conference.
- Each paper may be written by one or more authors.
- A PC member cannot review a paper that is written by himself/herself.
- Each paper is to be assigned to a minimum of three and a maximum of four PC members.

5.3.2 Entity-Relationships Diagram

A semantic data model for the logical database design is used here. The ER diagram is showed in Figure 5.1. It clearly shows the entities and their relationships.

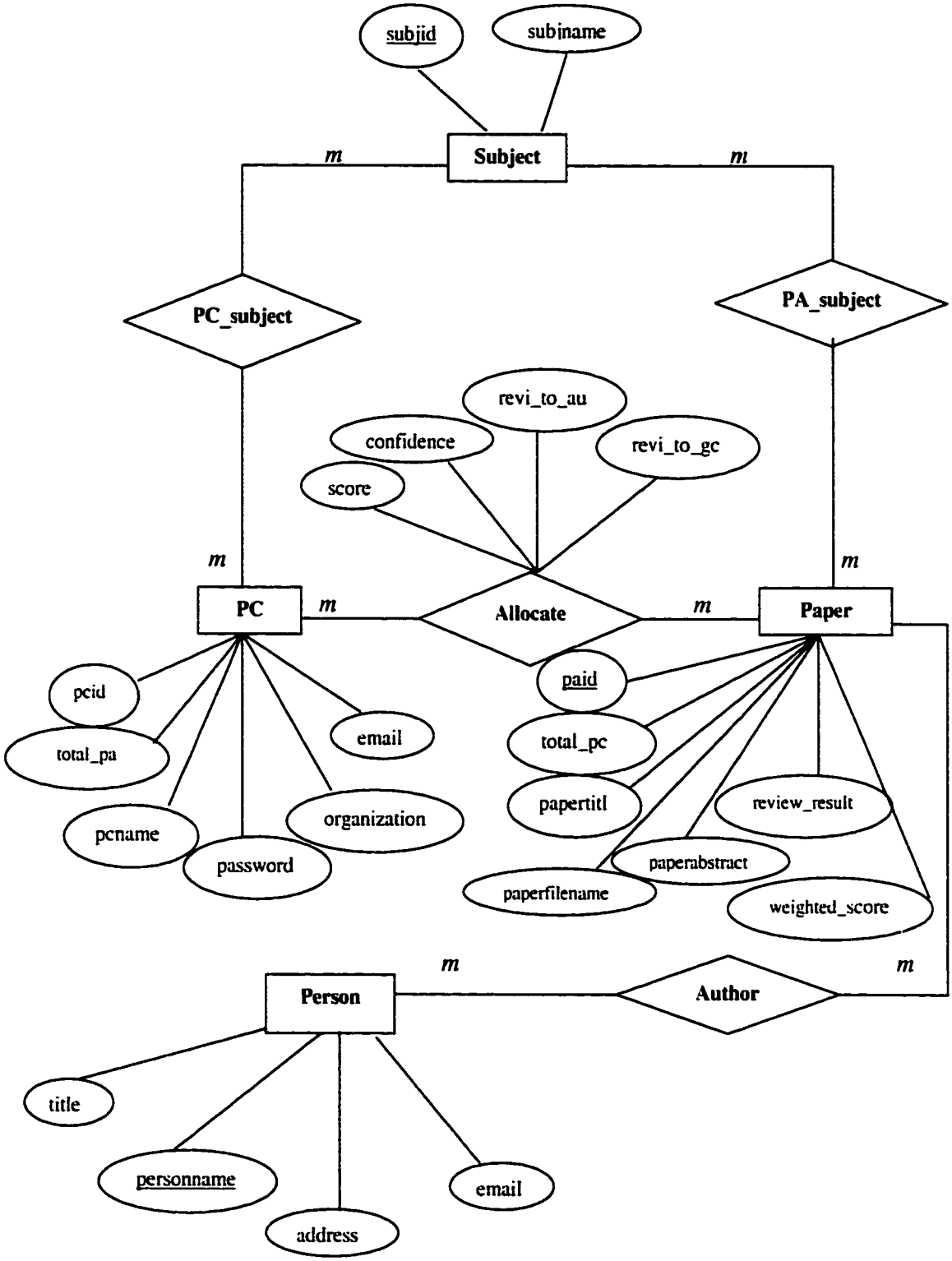


Figure 5-1CMS Entity-Relationships Diagram

5.3.3 Database Schema

Schemas are defined using SQL as followings.

pa_subject (paid:integer, subjid: varchar)

pc_subject (pcid: integer, subjid: varchar)

pc (pcid: integer, total_pa: integer, pcname: varchar, password:varchar, organization: varchar, email: varchar)

paper (paid: integer, total_pc: integer, papertitle:varchar, paperfilename: varchar, paperabstract: varchar, review_result: varchar, weithted_score: varchar)

allocate (paid: integer, pcid: integer, score: double, confidence: double, revi_to_au: varchar, revi_to_gc: varchar)

author (paid: integer, authorname: varchar, title: varchar, address: varchar, email: varchar)

Subject (subjid: integer, subjname: varchar)

5.4 Data Dictionary

This data dictionary contains the tables including entities and relations, and attributes and their descriptions. It will allow you to understand the database design and database implementation.

Data Dictionary Schema

Table Name* Description for table**	**Attributes*****	**Description for Attributes********

* Name of tables

** Description of table

*** Name of attributes

**** Descriptions of attributes

Relation Specification	Attributes	Descriptions
pa_subject information about topics each paper has	paid subjid	Paper ID number Topic ID number
pc_subject information about topics each PC member has	pcid subjid	PC ID number Topic ID number

Table 5-1 Data Dictionary (1)

Relation Specification	Attributes	Descriptions
Pc PC member information about name, password, organization and email address and total reviewed paper number.	Pcid total_pa pcname password organization email	PC ID number Paper number which is reviewed by PC member PC member name PC member password PC member's oranization PC member's email address

Paper paper information about paid (paper's file name), paper abstract, paper's review result and PC members total number which review the paper and weighted score.	Paid total_pc papertitle paperfilename paperabstract review_result weighted_score	Paper ID number pc number which review the paper paper title paper file name paper abstract the finial result of review the final score computed by some rule
Allocate the information about which paper should be reviewed by which PC member and correspond PC member's evaluation to author and GC. Also include the information about paper' score and confidence.	Paid pcid score confidence revi_to_au revi_to_gc	paper ID number PC ID number paper' score made by PC member paper's confidence made by PC member paper's evaluation made by PC member paper's evaluation made by PC member

Table 5-2 Data Dictionary (2)

Relation Specification	Attributes	Descriptions
Author information about author name, title, address, email and paid.	paid authorname title address email	paper ID number author name author title author address author email
Subject information about topics of conference	subjid subjname	Topic ID number Topic name

Table 5-3 Data Dictionary (3)

6 Testing

6.1 Objectives

Software testing is one of the major processes of system development. The objective of testing is to ensure that the system conforms to its requirement specifications (i.e. Software Verification) and the system implementation has met the expectation of the customer (i.e. Software Validation). A successful testing process should systematically uncover different classes of errors in a minimum amount of time and with a minimum amount of effort. It also demonstrates that the software appears to be working as stated in the specifications. The data collected through testing can provide an indication of the software's reliability and quality.

6.2 Testing Methods

I selected two stages of testing process, that is, Unit testing and Integration testing during implementation of the software.

- Unit testing — Individual components are tested. These components includes: paper registration, PC workStation and GC workStation.
- Integration testing — This includes sub-system testing and system testing to test collections of modules which have been integrated into the sub-system and system. The integration testing was done for the client, the server, and the database. Incrementally a new module is added to test the system, until all modules are

combined together and work successfully. The integration testing represents all aspects of the complete system under all feasible and extreme conditions.

In order to satisfy the objectives of the verification and validation process, both static and dynamic techniques have been applied. Static verification is a method that involves static analysis and checking of system represented by the requirement document, design diagrams and the program source code at all software process stages. Static verification is applied to all processes. The dynamic validation has been used once the executable program was available. The top down approach was used in feasibility study and bottom up approach was used in Unit testing. It is carried out to ensure:

- The system Function can meet the customer's expectation.
- The system performance is acceptable
- The system reliability
- The system's robustness in terms of handling exceptions or incorrect input.

6.2.1 Input Control Testing

To verify the system the input control and assignment algorithm are highly dependable and remain capable of ensuring the desired degree of accuracy and system integrity, we document them to highlight our testing targets.

Control input is defined as the procedural controls necessary to handle data prior to computer processing. Input data must be handled very carefully since they are most probable source of errors in the entire system. For instance, in file uploading, if the name

of the uploading file being uploaded has already been used in the destination directory, the system will give a prompt and let the user change the file name. This will avoid an existing file being overwritten. If wrong data is input into the computer, the error results they generate may spread throughout the entire system. In order to keep errors to a minimum, the input items in the forms use selections as much as possible rather than typing. The data input validation and intelligent prompt will effectively speed up the input process and reduce data errors, such as entering user ID, name or password.

6.2.2 Allocation Algorithm Testing

Another important testing part is the assignments of papers to PC member for its complexity of the algorithm. Two groups of data samples were used for its testing. One is the small group of testing data that includes six PC members and five papers. When pcAtLeastReviewPaNo is 3 and pcAtMostReviewPaNo is 4, the result of allocation is successful. Another is a relatively bigger group of testing data that contains eight PC members and twenty papers. When pcAtLeastReviewPaNo is 6 and pcAtMostReviewPaNo is 8, the result of allocation is successful too.

But occasionally, when I changed the order of testing data the result was incorrect and zero value appeared in the paper ID number. After the allocation algorithm verification, the possible cause of this problem could come from the method allocateSinglePaToPc6() or method allocateSinglePaToPc8(). Those might make zero as part of pcid or paid for allocating PC member or paper. The result of individual testing case doesn't represent all the possible results. Using algorithm verifications can be more efficient than applying

testing case if the data quantity is quite large. Therefore the testing also involves those algorithm verification.

For real data allocation testing, the parameter list of pcAtleastReviewPaNo as 6 and pcAtMostReviewPaNo as 8 was used. The result of allocation shown was unsuccessful and the system threw an ArrayIndexOutOfBoundsException. So I checked all the code related to this allocation and fixed the size of arrays to make it big enough. But after extending the array size, the result is still unsuccessful, like "paid = 25" and "pcid =0". Actually no PC ID number is 0. When the system tries to select pcname form pc table, pcid='0' threw a java.lang.NullpointException.

After testing this allocation algorithm, I analyzed and checked the code repeatedly. Finally I found that I missed the extreme condition (that is when the topics for many papers are of interest to a very limited number of PC members) that causes the error. And I fixed the bug.

6.3 Testing Plans and Test Cases

This section presents a series of testing plans and testing cases. I choose black box strategy to test all the Functions of CMS system based on the Functional Requirement Specifications in part 3. In the following SRD&S is abbreviation of System Requirement Definition and Specification.

6.3.1 Selection in Home Page

Testing cases:

- Enter 'Paper Registration' page.
- Enter 'PC Sign In' page.
- Enter 'GC Sign In' page.

Type	Selected Test Input Data	Expected Test Results	Testing Results	Comments (Special Requirements)
Function	Click on link 'Paper Registration'	Display 'Paper Registration' Page	Display 'Paper Registration' Page	Result satisfies SRD&S
Function	Click on link 'PC WorkStation'	Display 'PC Sign In' Page	Display 'PC Sign In' Page	Result satisfies SRD&S
Function	Click on link 'GC WorkStation'	Display 'GC Sign In' Page	Display 'GC Sign In' Page	Result satisfies SRD&S

Table 6-1 Admission Testing Case

6.3.2 Paper Registration

Testing cases:

- Show all the information that author submitted.
- Receive the paper files that author submitted.
- Show all the information about author and co-author.
- Upload files from web page.
- Back to the home page.

Test Type	Selected Test Input Data	Expected Test Results	Testing Results	Comments (Special Requirements)
Function	Author input all the information about paper and author then click submit button	Display all the information author just put in, and co_author form.	Display all the information author just put in, and co_author form.	Result satisfies SRD&S
Function	Go to jswdk1.0.1->webpage->collectedPapers directory to check just input file name.	The file author just input should be in the collectedPaper directory.	The file author just input should be in the collectedPaper directory.	Result satisfies SRD&S
Function	Author input all the co_author information into the co_author form then submit it.	Display all the author and co_author information and co_author form.	Display all the author and co_author information and co_author form	Result satisfies SRD&S
Function	Submit paper file which name has existed in the 'collectedPapers' directory	Display error message to let author rename file name.	Display error message to let author rename file name.	Result satisfies SRD&S
Function	Click on the link 'finish'	Return to the home page.	Return to the home page.	Result satisfies SRD&S

Table 6-1 Paper Registration Testing Case

The following figure is to show the data entered in Paper Registration form

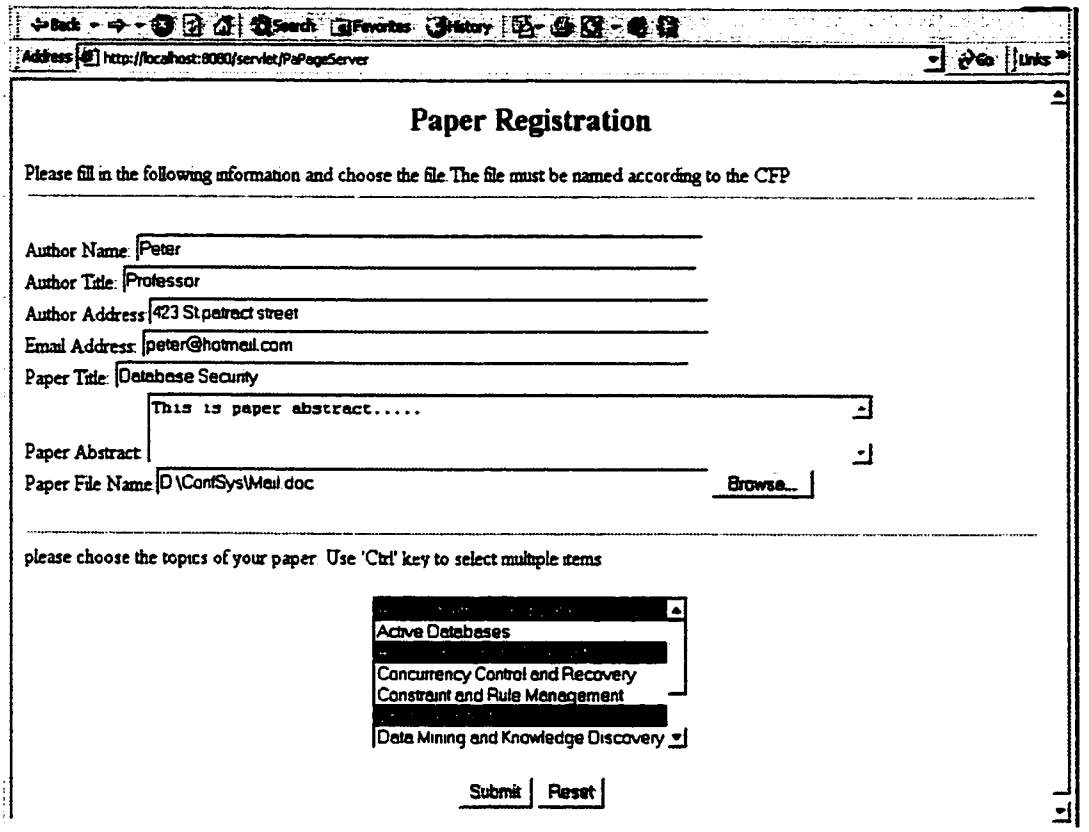


Figure 6-1 Data Entered in the Paper Registration Form

When an author finishes the above information and clicks the "Submit", the corresponding feedback information shows as following figure6-2.

In the feedback information form, when user press the ok button the form of co-author's information will displayed.

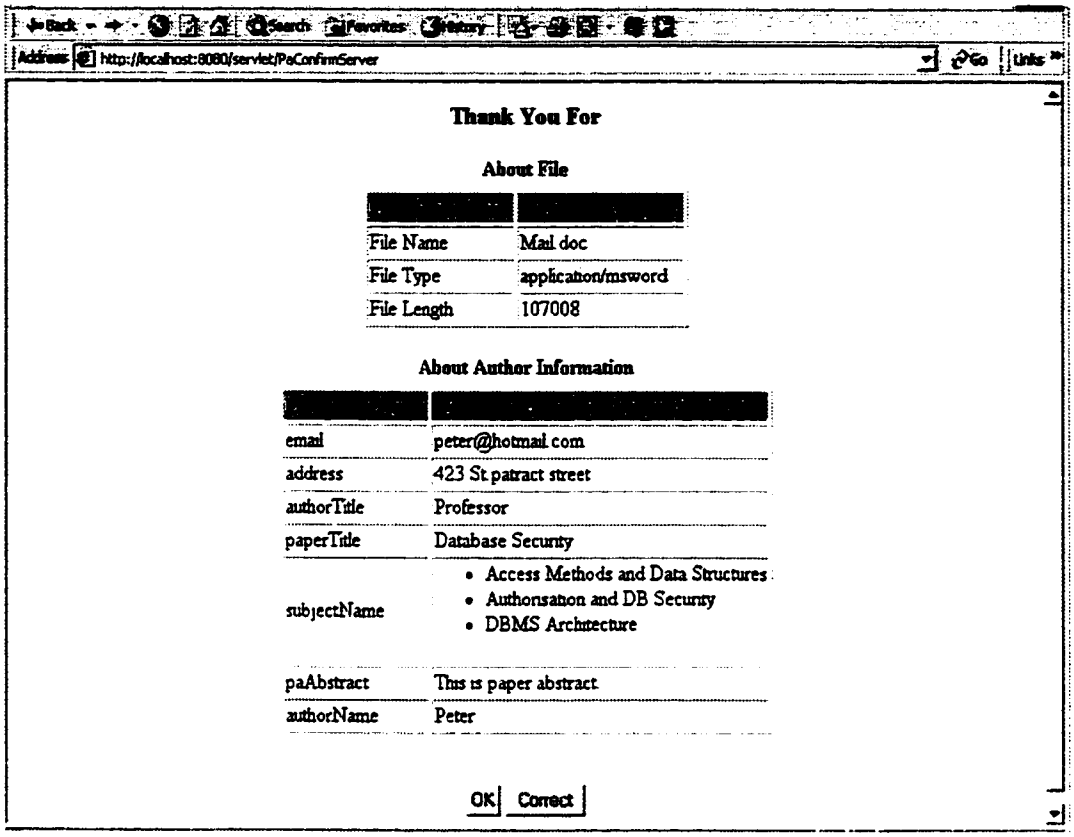


Figure 6-2 Feedback information entered in the paper registration form

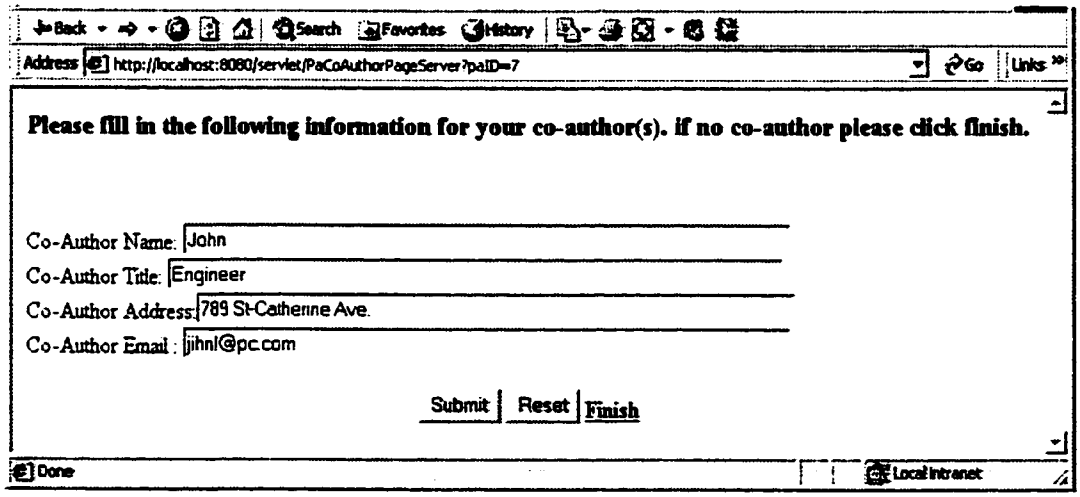


Figure 6-3 Data entered in the co-author form

The following figure shows the information of the above two co_ authors of the paper.

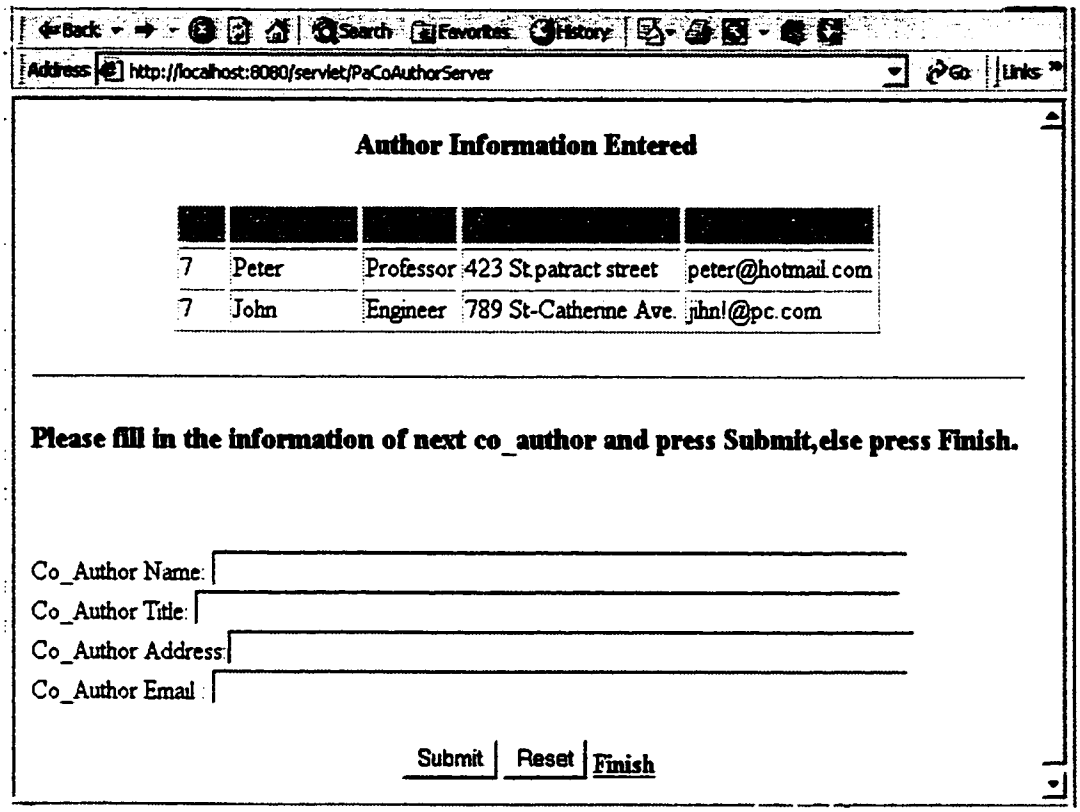


Figure 6-4 Show all the authors of a paper

6.3.3 PC Registration

Testing cases:

- PC member input correct name and password in "PC sign in" page.
- PC member input incorrect name or password in "PC sign in" page.
- PC member enters the "PC Registration" page.
- PC member input correct ID number and name and organization in PC registration page.
- PC member input incorrect ID number or name in PC member registration page.

Test Type	Selected Test Input Data	Expected Test Results	Testing Results	Comments (Special Requirements)
Function	PC member input the correct name and password in PC member sign in page then click the "Sign In" button.	Display the page that has four option for PC member	Display the page that has four option for PC member	Result satisfies SRD&S
Function	PC member input the incorrect name or password in PC member sign in page then click the "Sign In" button.	The error message will be displayed to let PC member type it again.	The error message will be displayed to let PC member type it again.	Result satisfies SRD&S
Function	PC member click the link 'PC Registration'	Display 'PC Registration' page.	Display 'PC Registration' page.	Result satisfies SRD&S
Function	PC member input correct ID number and name and organization. Then click submit button in the PC member Registration page.	Display information PC member just put in.	Display information PC member just put in.	Result satisfies SRD&S
Function	PC member input incorrect ID number or name. Then clicks submit button in the PC member Registration page.	Display error message and let member reenter PC member information	Display error message and let PC member reenter PC member information	Result satisfies SRD&S

Table 6-1 PC Registration testing case

In the following PC registration window, when PC member enters the PC member name, PC ID number, PC member organization, and clicks Submit PC member Information, a window which will display the information just enter appears.

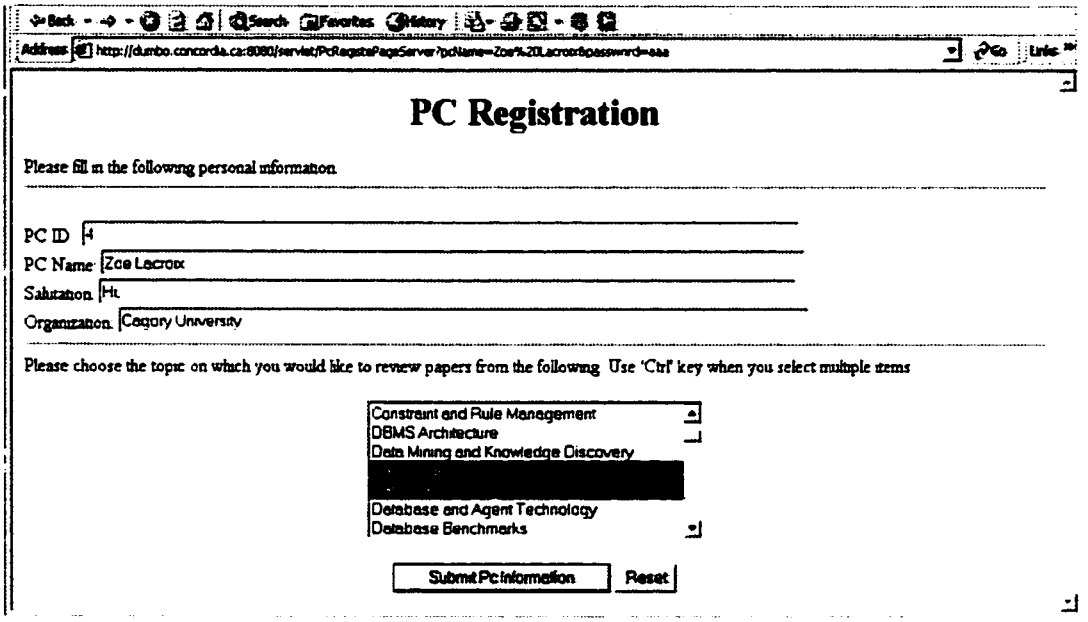


Figure 6-1 PC Registration

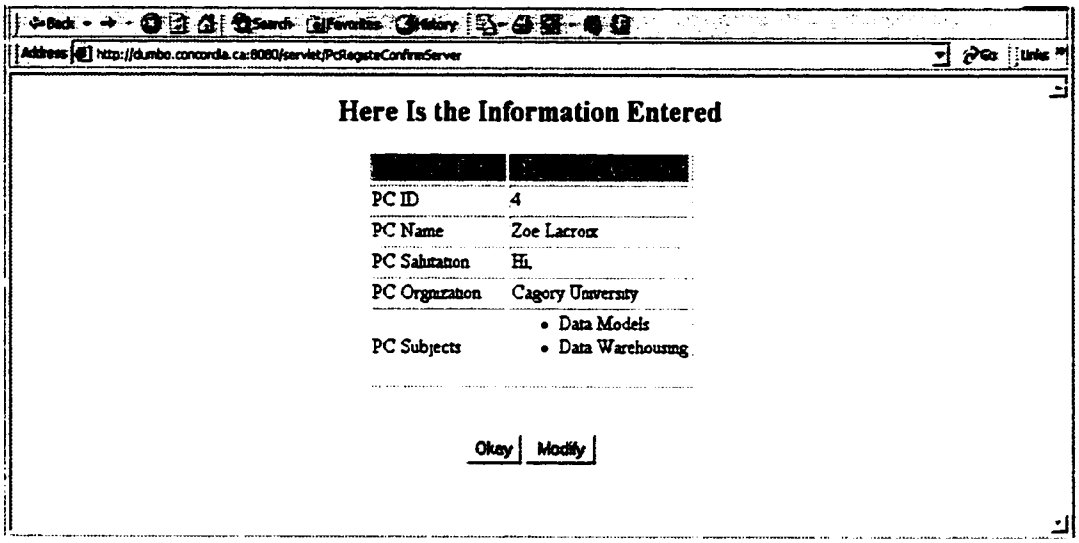


Figure 6-2 Show PC Information Just Put In

6.3.4 PC Download File

Testing cases:

- Enter the 'PC Download File' page.
- Check the list of files that the PC member is responsible for reviewing.
- Check the list of topics that PC member entered from the web pages.
- Download all the listed files which are going to be reviewed by the PC member.

Test Type	Selected Test Input Data	Expected Test Results	Testing Results	Comments (Special Requirements)
Function	PC member click the link 'PC Download File'	Display 'PC Download File' page	Display 'PC Download file' page	Result satisfies SRD&S
Function	Check the list of files names.	the list of files are the right files which the PC member is allocated for reviewing.	the list of files are the right files which the PC member is allocated for reviewing.	Result satisfies SRD&S

Test Type	Selected Test Input Data	Expected Test Results	Testing Results	Comments (Special Requirements)
Function	Check the list of topics	The listed topics are entered by the PC member from the web page before.	The listed topics are entered by the PC member from the web page before.	Result satisfies SRD&S
Function	Check whether the listed files can be download.	PC member can download all the listed files.	PC member can download all the listed files.	Result satisfies SRD&S

Table 6-1 PC Download file Testing Case

The following figure appears when PC member clicks the link 'PC Download File' option in the PC member work selection page. The following figure shows all the downloadable files name which are to be reviewed by this PC member. For example, PC member name is bbb and is allocated paper file name is shown in Figure 6-10.

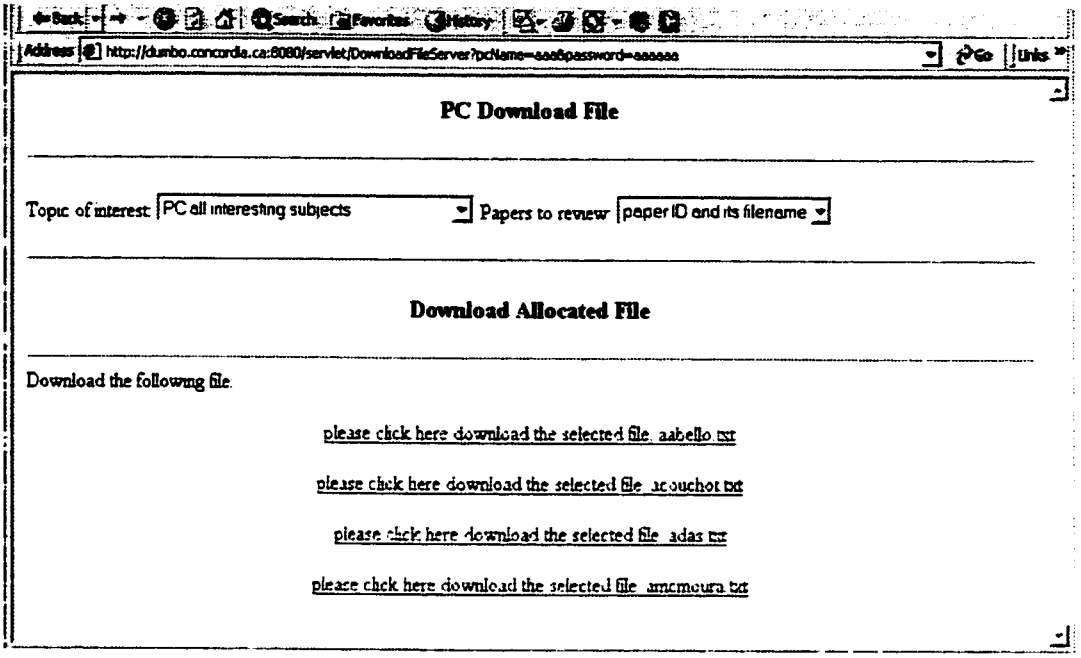


Figure 6-1 PC Download File

6.3.5 PC Review

Testing cases:

- Enter the 'PC Review Page1' page.
- Enter the 'PC Review Page2' page.
- Click the 'show Overview Paper Reviewed By The PC' button in the 'PC Review Page2' page
- Click the submit button in the 'PC Review Page2' page

Test Type	Selected Test Input Data	Expected Test Results	Testing Results	Comments (Special Requirements)
Function	PC member click the link 'PC Review'	Display 'PC Review Page1' page	Display 'PC Review Page1' page	Result satisfies SRD&S
Function	PC member click the button in the 'PC Review	Display 'PC Review Page2' page	Display 'PC Review Page2' page	Result satisfies SRD&S

Test Type	Selected Test Input Data	Expected Test Results	Testing Results	Comments (Special Requirements)
	Page1' page.			
Function	Click the 'show Overview Paper Reviewed By The PC' button in the 'PC Review Page2' page	Display ID, name, score, confidence, evaluation of all the papers reviewed by PC member.	Display ID, name, score, confidence, evaluation of all the papers reviewed by PC member.	Result satisfies SRD&S
Function	Click the submit button in the 'PC Review Page2' page	The database contains the information about paper's score, confidence, evaluation to author and GC the PC member just put in.	the database contains the information about paper's score, confidence, evaluation to author and to GC the PC member just put in.	Result satisfies SRD&S

Table 6-1 PC Review Page Testing Case

6.3.6 GC Input Data

Testing cases:

- GC input correct name and password in PC member sign in page.
- GC input incorrect name or password in PC member sign in page.
- GC enters the 'Input Data' page.
- Create database tables.
- Submit PC member's information to the database.
- Insert topic information into the database.

- Update GC's information in the database.

Test Type	Selected Test Input Data	Expected Test Results	Testing Results	Comments (Special Requirements)
Function	GC input the correct name and password in GC sign in page then click the 'Sign in ' button.	Display the page that has four options for GC	Display the page that has four options for GC	Result is correct
Function	GC input the incorrect name and password in GC sign in page then click the 'Sign in ' button.	The error message will be displayed to let PC member type it again.	The error message will be displayed to let PC member type it again.	Result is correct
Function	Click the link 'Input Data'	Display ' Input Data' page	Display ' Input Data' page	Result is correct
Function	Click 'create table' button	Database created all tables	Database created all tables	Result is correct
Function	Enter some PC member information then click submit button	In PC member table we can see the PC member's information just put in	In PC member table we can see the PC member's information just put in	Result is correct
Function	Insert topic name into the database topic table	From the topic table, the topic name just put in can be seen	From the topic table, the topic name just put in can be seen	Result is correct
Function	Update GC's some information in the PC table	From the PC table, we can see the updated GC's information.	From the PC table, we can see the updated GC's information.	Result is correct

Table 6-1 GC Input Data

6.3.7 GC Send Email

Testing cases:

- Enter 'Send Email' page.
- Send email to special personal.
- Send email to all the PC members to inform their ID number and password.
- Send email to some PC members who don't submit their topic yet.
- Send email to some PC members who don't submit their review yet.
- Send email to all the authors to inform the review results.

Test Type	**Selected Test Input Data**	**Expected Test Results**	**Testing Results**	**Comments (Special Requirements)**
Function	Click the link 'Send Email'	Display ' Send Email' page	Display ' Send Email' page	Result is correct
Function	From wang@cs.concordia.ca send email to Mr. Zang	Mr. Zang should receive the email from wang@cs.concordia.ca	Mr. Zang receives the email from wang@cs.concordia.ca	Result is correct
Function	GC send email to all the PC members. Click the 'sendInfoToAllPc' button	All the PC members should receive the email from GC	All the PC members receive the email from GC	Result is correct
Function	Set PC(ID=1) who doesn't submit topic. Then GC click the 'sendToPcRemindSubject ' button.	PC member whose ID number is 1 should receive the email form GC.	PC member whose ID number is 1 receives the email form GC.	Result is correct

Test Type	Selected Test Input Data	Expected Test Results	Testing Results	Comments (Special Requirements)
Function	Set PC (ID =2) who doesn't submit paper's evaluation to author and GC. Then GC click 'sendPcRemindReview' button	PC member whose ID number is 2 should receive the email from GC.	PC member whose ID number is 2 receives the email from GC.	Result is correct
Function	GC send email to all the tested author	All the tested authors should received the email from the GC.	All the tested authors should received the email from the GC.	Result is correct

Table 6-1 GC send e-mail testing case

6.3.8 GC Allocate Paper

Testing cases:

- Enter 'Allocate Paper' page.
- Check the allocation result.
- Delete all the allocated items in database.
- Choose another group of parameters to reallocate.
- Show all the tables in the database.

Test Type	Selected Test Input Data	Expected Test Results	Testing Results	Comments (Special Requirements)

Test Type	Selected Test Input Data	Expected Test Results	Testing Results	Comments (Special Requirements)
Function	Click the link 'Allocate Paper'	Display ' Allocate Paper' page	Display ' Allocate Paper' page	Result is correct
Function	Choose PC member review paper number then click ' startAllocate' button	If chosen parameters is right for the reality situation then will show successful result, otherwise show unsucessful information	If chosen parameters is right for the reality situation then will show successful result, otherwise show unsucessful information	Sometime result have some problem.
Function	If allocation is not success then click the 'deleteAllocatedItems' button	In the database all the allocated items should be deleted.	In the database all the allocated items are deleted.	Result is correct
Function	Reenter the parameters about paper number reviewed by PC member then click 'StartAllocate' button	If chosen parameters is right for the reality situation then will show successful result, otherwise show unsucessful information	If chosen parameters is right for the reality situation then will show successful result, otherwise show unsucessful information	Sometime result have some problem.
Function	Click 'ShowAllTable' button.	From web page GC can see most of tables just same as one in the database.	From web page GC can see most of tables just same as one in the database.	Result is correct

Table 6-1 GC allocate paper testing case

All the details about allocation test are discussed in section 6.2.2. For the convenience to GC, the button 'ShowAllTable' is set in GC Allocation form. By doing this, GC can check the information in the database.

6.3.9 GC Review Paper

Testing cases:

- Enter 'Review Paper' page.
- Submit GC's evaluation of paper.
- Show overview of all papers.

Test Type	Selected Test Input Data	Expected Test Results	Testing Results	Comments (Special Requirements)
Function	Click the link 'Review Paper'.	Display 'Review Paper' page	Display 'Review Paper' page	Result is correct
Function	After finishes the paper's evaluation then click submit button.	In the database paper table the GC's evaluation should be seen.	In the database paper table the GC's evaluation can be seen.	Result is correct
Function	Click 'showOverviewOf AllPapers' button.	In the web page all the papers's ID, title, file name, score, confidence, and so on, should be displayed.	In the web page all the papers's ID, title, file name, score, confidence, and so on, can be displayed.	Result is correct

Table 6-1 GC review paper testing case

In the GC review form there is a button allowing GC to view all the paper information. This is convenient for GC to check the reviews about paper before evaluating the paper. The following screen shot is provided as an example of each test data.

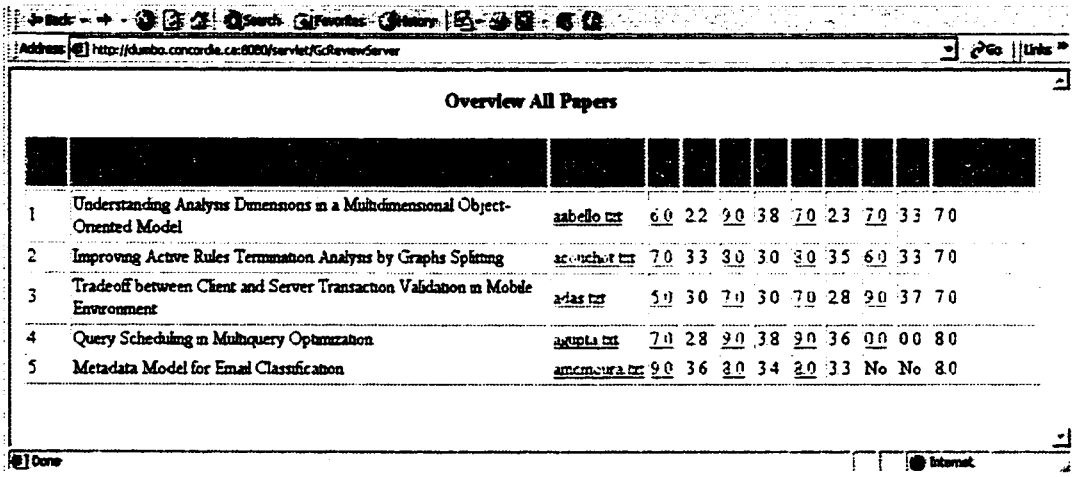


Figure 6-1 Show overview of all papers

7 Future Improvement

7.1 Assignment Papers to the members

The allocation algorithm is a complicated topic in CMS system. In many situations, no result can be found. The possibility of finding the result depends on the difference between the distribution of the interested topics of PC members and the distribution of paper's topics. The smaller this difference, the easier the assignment result can be found. If the difference is too big, no result can be found. This is one of the reasons why the paper assignment is difficult to be successful.

To some degree, the GC can control the distribution of PC member's interested topics, but not the distribution of Paper's topics. If the GC can receive most papers of the conference earlier and then invite the PC members, the GC may make the PC member's interested topics closer to the distribution of the papers. But generally it is not easy.

In this project, I have utilized one algorithm and several groups of parameters. It is quite difficult to get a perfect assignment result by doing it once according to the predetermined rule (each PC member needs to reviewed at least six papers and at most eight papers). So when the assignment was unsuccessful, the parameter tuning was necessary.

Also it is more efficient to write a small program to generate testing data group automatically and test the allocation algorithm directly. There is a lot of space to improve the allocation algorithm.

7.2 Graphic User Interface Design

A good interface design can help to avoid the inconsistency between system functions and appearances. It also serves as a guideline to the system developers and users.

In this project, Graphic User Interface design is not emphasized as an important part of the overall system. Current system user interface mostly use our intuitions without strictly following some GUI design principles. So some interface designs are not satisfied. For example, in the GC workstation form help information should be added to tell GC how to interact with the system. In the GC review form, when GC decides which paper is to be reviewed, all the information about this paper should be displayed. It is convenient for GC to retrieve the information related to some paper.

7.3 Software Development Process

Object Oriented techniques are applied in the process of software development in this project. However, Object Oriented Analysis was not carried out as a distinct phase in this practice, System design can be extended by using heterogeneous design schemes such as function design etc.

References

1. Marty Hall, 2000, Core Servlets and JavaServer Pages, Sun Microsystems Press.
2. Jason Hunter with William Crawford, 1998. Java Servlet Programming, O'Reilly & Associates, Inc.
3. Robert Orfali and Dan Harkey, 1998, Client/Server programming with Java and CORBA, Wiley Computer Publishing.
4. Y. Daniel Liang, 1999, Introduction to Java Programming, second Edition.
5. Timothy Budd, 1997, An Introduction to Object-Oriented Programing.
6. James Rumbaugh and William Premerlani, etc, Object_Oriented Modeling and Design.
7. Prashant Sridharan, 1997, advanced Java networking.
8. Abraham Silberschatz and Henry F. Korth, 1998, Database System Concepts.
9. http://www.servlet.com
10. http://www.servletguru.com
11. http://www.novocode.com/doc/servlet-essentials

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

ASSOCIATIVE DATA MODEL AND CONTEXT MAPS

MINGHUI HAN

A MAJOR REPORT

IN

THE DEPARTMENT

Of

COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE

CONCORDIA UNIVERSITY

MONTREAL, QUEBEC, CANADA

AUGUST 2001

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the major report prepared

By: **Minghui Han**

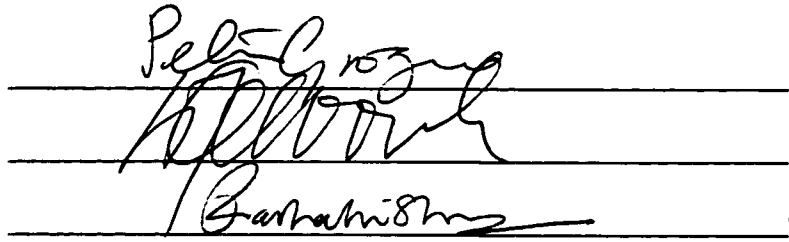
Entitled: **Associative Data Model and Context Maps**

And submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

Complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:



_____ Examiner

_____ Supervisor

_____ Co-Supervisor

Approved _____

Chair of Department or Graduate Program Director

SEP 19 2001, _____

Dr.Nabil Esmail, Dean

Faculty of Engineering and Computer Science

Abstract

Associative Data Model and Context Maps

Minghui Han

This report presents the possibility of using *context maps* to represent *associative data model.* This new technology for *associative data model* can be presented as the *joined maps* (*jMaps*) of concepts and relationships. The solution for converting a set of *context maps* into one database or retrieving information from the database to *context maps* was developed. The software was developed by using *VBA* (Visual Basic for Application), which can give us access to *Microsoft Office* for integration with databases. The implementation for this technology was demonstrated by using *MS Excel* spreadsheet to display the *associative model* of data and *MS Access* to store a set of converted *context maps*.

Acknowledgements

I wish to thank all those who made the final realization of this dissertation possible. It is not possible to mention all their names, however I would like to express my special gratitude to the following contributors.

I am pleased to express my gratitude to Professor T. Radhakrishnan, Chair of Department of Computer Science, for acceptance to be my co-supervisor. He made many useful comments for this report. My thanks are extended to Prof. Peter Grogono, an examiner for my work, for his valuable comments on this report. My thanks also go to Halina Monkiewicz, the Graduate Program Secretary, for her support and collaboration. I am greatly indebted to my supervisor, Prof. W.M. Jaworski, who encouraged my interest in the subject of software information system, for his technical advice, generous attention, constant help and critical remarks throughout this work. He patiently guided to me with his knowledge about information systems and encouraged me in tackling various difficult issues.

My sincere thanks are due to the support given by my company, Motorola Canada Software Center, Montreal, Quebec, Canada. My special thanks to Gerald Dunkelman, Operation Manager and Alf Lund, project Manager for their inspiration, encouragement in many ways during my master program study.

Finally, I would like to express my deep gratitude to my wife, Dawei, for her all the support and she always believe me and I can always rely on her.

Contents

List of Figures

List of Tables

Chapter 1

1. Introduction

1.1 Background

There are many representations and methodologies for information systems and software engineering, such as *CASE* tools and Rational Rose *UML*, which can be presented with graphical notations for the information system views.

However it is a challenge to develop a methodology with safety critical systems which needs to be simple and easy to implement. The *joined maps* viewed as *context maps* in this report is one way to represent the above requirement. The *joined maps*, or *jMaps*, are a notation and a method for representing systems architecture, structures, processes and reusable templates. The *jMaps* can be synonyms with syntax maps. This technology was first introduced by W.M. Jaworski [1995]. The technology was initially developed as a means of recovering and refining knowledge from legacy system. This technology has a history of names. During the late 1970s and early 1980s, based on conceptual graphs introduced by J.F.Sowa [1984], it was named as *ABL*, or *Array Based Language* (Jaworski [1987]). In the late 1980s, it was renamed as *ABL/W4*. *W4* represents as what, when, where and which. In the early 1990s, Prof. Jaworski [1995], by considering existing notations and methodologies, named this technology as *jMaps*. In the late 1990s until now, *jMaps* can be presented as *Context Maps* (Jaworski [1999]). With *jMaps* or *Context Maps* technology, by using the popular concept of a spreadsheet it is feasible to communicate the design information to different audiences. The *jMaps* notation allows

efficient recovery and modeling of generic schemata for processes, objects and views of information systems.

The *associative data model* was developed by Simon Williams [2000]. The *associative model* treats the information in the same way as the human brain, i.e. treats the things with association between them. Those associations can be expressed through the simple subject-verb-object syntax of an English sentence. The *associative model* divides the real–world things with two kinds of sorts: *Entities* and *Associations*. According to Simon Williams [2000], *Entities* are the things that have discrete, independent existence. An entity's existence dose not depend on any other thing. *Associations* are the things whose existence depends on one or more other things, if any of those things ceases to exist, the thing itself ceases to exist or becomes meaningless.

The *associative model* overcomes the limitations of the relational model and avoids the complexities of the object model by structuring information in a more accessible and intuitive manner than either. The *associative model* overcomes two fundamental limitations of current programming practice: the need to write new programs for every new application, and the need to store identical types of information about each instance. It also offers a superior distributed data model, allowing one database to be distributed over many geographically dispersed web servers. Moreover, associative databases may be readily tailored to serve different requirements simultaneously, and different databases may be easily combined and correlated without extra programming

By considering the basic concepts of *associative data model*, it becomes possible for us to use *context maps* to represent *associative data model*.

1.2 Objective of Study

The main purpose of the research work reported herein is to introduce the new method of using *context maps* to represent *associative data model*. Based on this new technology, we will design and develop a software for converting a set of *context maps* into one database or retrieving data information from the database. The *associative data model* will be presented as the *joined maps* (*jMaps*) of concepts and relationships in the *MS Excel* spreadsheet.

The main purpose of this project is based on the *associate model* of information to produce the related *jMap* in the form of *MS Excel* spreadsheet. By considering *context maps* for associative data model, it is focused on using *context maps* to represent the *associative data model*, and exporting *context maps* into database or recovering the data from a database to spreadsheets in the *jMaps* format.

The application software was written by *VB* with emphasis on using Micro Office application. Since our developing software is a small project, the *MS Excel* spreadsheet and *MS Access* database are sufficient in using this project.

1.3 Project Scope

The research work for this project was supervised by Prof. W.M. Jaworski. The work study was started in January 2001. The procedure to develop this project is structured in the following way:

1) Try to get familiar in using associative data model, especially in understanding the basic concepts of this new technology for representing the database model.
2) Analyze the basic requirements for this project. List the relationships between entities and associations for a special example.
3) Do research on the *jMap* notation, and converting the *associative model* with *jMaps* notations into a spreadsheet.
4) Project design, source coding in *MS Excel* by using *VBA*, with special emphasis on converting a set of data into the database and restoring the *jMaps* from the database.
5) Integrate the program, and make all functions work.
6) A deliverables project package will contain a full description of manual, sample Excel file and sample database file
7) Make a conclusion for this research work and provide recommendations for future works.

Chapter 2

2. Associative Model Introduction

2.1 Data Model

In the database management system, we can record the existence and properties of things in the real world. The transition from things which we want to record information into a database relies on using a modeling system. The modeling system consists of three layers of abstraction: a conceptual layer, a logical layer and a physical layer.

- The conceptual layer is the highest level and is more abstract than the other layers. It describes what should the modeling system in representing things in the real world, and sets the rules about how they may be used in the modeling system.
- The logical layer describes the logical building blocks which the database uses to store and access data, and how to map the conceptual layer into logical layer.
- The physical layer is the lowest level which describes the physical building blocks which exist in the computer's memory and are stored and retrieved in its hardware storage. The physical layer decides how the logical building blocks map into physical layer.

In above layers, the conceptual and logical layers together make up the data model. In this case, we can conclude that the data model is a scheme for structuring data in databases, the logical and the physical layers together make up the database aspects.

The data model is fundamental for database management systems. According to Simon Williams [2000], five data models have been proposed and used since computers became available. Those five data models are: the *network model*, the *hierarchical model*, the *relational model*, the *object model*, and the *object/relational model*. In the above models, the two most significant and widely adopted models are the relational model and the object model. Today's database market is dominated by products based on the *relational model*.

2.2 Relational Model

The *relational model* was first described by Dr. Edgar Codd of IBM's San Jose Research Laboratory in 1970. Nowadays, the *relational model* is the foundation of almost every commercial database. The *relational model* stores data in special tables called "relations".

In the *relational model*, each table holds data for a particular type of thing or entity, such as customers, orders, students and so on. Within a table, each row represents one instance of the type of things that the tables stores and each column represents a piece of information that is stored.

Here is a simple example of customers and orders for which the source was taken from Simon Willams [2000]. The customers table has columns for customer number, name, telephone number, credit limit, outstanding balance and so on. The Orders table has columns for order number, date, customer number item, quantity and so on

Customers				
Customer number	*Name*	*Telephone no*	*Credit limit*	*O/S balance*
456	Avis	0171 123 4567	£10,000	£4,567
567	Boeing	0181 345 6789	£2,500	£1,098
678	CA	0123 45678	£50,000	£14,567
789	Dell	0134 56789	£21,000	£6,789

Table 1 Customers Relational Table

Orders				
Order no	*Date*	*Customer number*	*Item*	*Quantity*
11234	2-Mar-99	**567**	ABC345	150
11235	15-Mar-99	**789**	GGI765	25
11236	21-Apr-99	**789**	KLM012	1,000
11237	7-May-99	**456**	GHJ999	£6,789

Table 2 Orders Relational Table

Within each table, rows are uniquely identified by one or more special columns called primary keys. The relationship between an order and the customer who placed it is recorded by putting the customer's number into the "customer number" column of the order's row in the *Orders* table. This is an example of a foreign key. The foreign keys in table are shown in bold.

The *relational model* is the standard architecture for the database management systems. However it has some fundamental limitations such as the following:

- Each new relational database application requires a new set of programs. So the cost of application software increases.

- The relational database applications are difficult to customize for individual users.
- A relational database can not record a piece of data about a particular thing that is not relevant to all others of same type.
- It is difficult and sometimes not possible to combine two relational database.

2.3 Associative Model

2.3.1 General

The *Associative Model* is the first major advance beyond the *Relational Model.* The *Associative Model* of Data is the name given by Simon Williams [2000] to the set of concepts, structures and techniques underlying the *Sentences* database management system. The *Sentences(TM)* is an innovative database management system written in the Java language and based on the *Associative Model of Data.* The *associative model* builds on a body of academic research that includes: semantic networks, binary-relational techniques and the entity relationship model. We have added several important and unique concepts.

The *associative model* sees information in the same way as our own brains: as things and associations between them. These associations are expressed through the simple **subject-verb-object** syntax of an English sentence. For example:

The lake *is* coloured blue
Sherry *is* sister to Jim
Lee *has* a credit limit of $5,000
Montreal *is* located in Province of Quebec

A sentence may itself be the subject or object of another sentence, so the *associative model* can express quite complex concepts:

(Flight BA123 *arrives at* 20:15) *on* Monday
The Bible *says* (God created the World)

For previous Customers relational table, the sentence in the *associative model* can be described as following

Avis *is* a Customer
Avis *has* telephone number 0171 123 4567
Avis *has* credit limit £10,000
Avis *has* outstanding balance of £4,567
Boeing *is* a Customer
Boeing *has* telephone number 0181 345 6789
Boeing *has* credit limit £2,500
Boeing *has* outstanding balance £1,098

...and so on.

2.3.2 Associative Model Structure

According to Simon Williams [2000], an associative database comprises two data structures:

- **Items**, each of which has a unique identifier, a name and a type.
- **Links**, each of which has a unique identifier, together with the unique identifiers of three other things, that represent the source, verb and target of a fact that is

recorded about the source in the database. Each of the three things identified by the source, verb and target may each be either a link or an item.

The following example shows how the *associative model* would use these two structures to store the piece of information.

Example sentence:

"*Flight AC1234 arrived at Montreal Doval on 12-Aug-2001 at 10:25am*".

In the above sentence, we could divide seven items with:

the four things:

Flight BA1234,
Montreal Doval,
12-Aug-2001
10:24am

and the three verbs or prepositions

arrived at
on
at.

In this case, we need three links to store the data. They are:

Flight AC1234 arrived at Doval Airport
... *on* ***12-Aug-2001***
... *at* ***10:25am***

We can see that each line is one link. The first link uses "arrived at" to associate *Flight AC1234* and *Doval Airport*. The second link uses "on" to associate the first link and *12-Aug-2001*. The third link uses "at" to associate the second link and *10:25am*.

We can simply put brackets around each link. Written this way, our example would look like this:

((Flight BA1234 arrived at Doval Airport) on 12-Aug-2001) at 10:25am

This may look more like human language than the contents of a database, but if we chose for a moment to view the *associative model* through the eyes of the *relational model*, we see that any associative database can be stored in just two tables: one for items and one for links. Each item and link has a meaningless number to act as its primary key.

Items	
Identifier	Name
01	Flight AC1234
02	Montreal Doval
03	12-Aug-2001
04	10:25am
05	arrived at
06	On
07	At

Table 3 Items Associative Table

Links			
Identifier	Source	Verb	Target
11	01	05	02

12	11	06	03
13	12	07	04

Table 4 Links Associative Table

2.3.3 The Benefits of Associative Model

The *associative model* has following advantages:

- One program can be used to implement many different applications without being altered or rewritten in any way. The a*ssociative mode* allows users to create new applications from existing ones. This will significantly reduce the costs of software development.

- By using the *associative model*, applications can permit features to be used or ignored selectively by individual users without the need for complex parameters or customisation.

- A database can record information that is relevant only to one thing of a particular type, without demanding that it be relevant to all other things of the same type.

- Separate databases can be readily correlated or merged without extra programming, and multiple databases distributed across many servers can be accessed by applications as though they were a single database.

2.4 The Bookseller Example

In this section, we will describe the bookseller example to look at the more sophisticated problem and to show how the *associative model* deals with this problem. The example was taken directly from Simon Williams [2000]. This example will be also represented by *context maps* in later chapter.

The domain of bookseller problem as described following:

> *An Internet retail bookseller operates through legal entities in various countries. Any legal entity may sell books to anyone. People are required to register with the legal entity before they can purchase.*
>
> *For copyright and legal reasons not all books are sold in all countries, so the books that each legal entity can offer a customer depend on the customer's country of residence.*
>
> *Each legal entity sets its own prices in local currency according to the customer's country of residence. Price increases may be recorded ahead of the date that they become effective.*
>
> *Customers are awarded points when they buy, which may be traded in against the price of a purchase. The number of points awarded for a given book by a legal entity does not vary with the currency in which it is priced.*

With *associative data model*, the schema that describes the structure of orders for this problem is as follows. The items in bold are entity types.

Legal entity sells **Book**
... worth **Points**
... in **Country**
... from **Date**
... at **Price**
Person lives in **Country**

Person customer of **Legal entity**

... has earned **Points**

... orders **Book**

... on **Date**

... at **Price**

In above data itself, the items in italics are entities. Now we define the group of them that we are using; two legal entities, two books, two customers and two countries:

Amazon is a **Legal entity**

Bookpages is a **Legal entity**

Dr No is a **Book**

Simon Williams is a **Person**

Simon Williams lives in *Britain*

Mary Davis is a **Person**

Mary Davis lives in *America*

Britain is a **Country**

America is a **Country**

Spycatcher is a **Book**

Next comes the price list:

Amazon sells *Dr No*

... worth *75 points*

... in *Britain*

... from *1-Jan-00*

... at *£10*

... in *America*

... from *1-Mar-00*

... at *$16*

Amazon sells *Spycatcher*

... worth *50 points*

... in *Britain*

... from *1-Jun-00*

... at £7

... in *America*

... from *1-Jun-00*

... worth *35 points*

... in *Britain*

... from *1-Jan-00*

... at *£8*

... in *America*

... from *1-Jan-00*

... at *$14*

Bookpages sells Spycatcher

... worth *35 points*

... in *America*

... from *1-Jun-00*

... at *$13*

Here, for each of our two customers we record the number of points awarded to date, together with a single order:

Simon Williams customer of *Bookpages*

... has earned *1,200 points*

... orders *Dr No*

... on *10-Oct-00*

... at *£10*

Mary Davis customer of *Amazon*

... has earned *750 points*

... orders *Spycatcher*

... on *19-Oct-00*

... at *$12*

Here is the metadata for the bookseller problem in diagrammatic form. The ovals represent items; the lines represent links. The circles on the lines are the anchor points for links between items and other links.

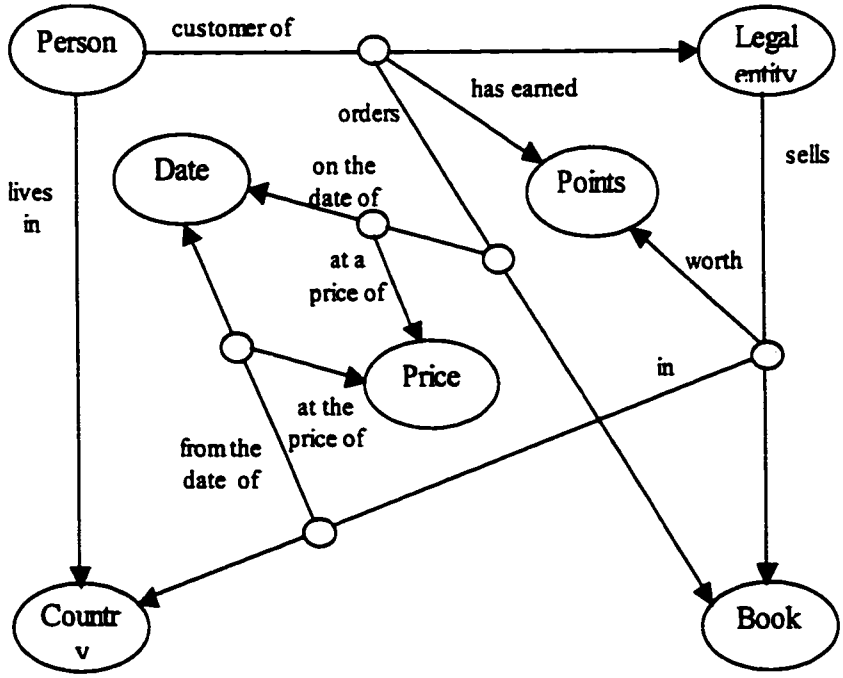


Figure 1 The Bookseller Problem in the Diagrammatic Form

Comparison with *associative model* and *relational model*, the associative schema usually take much less lines that record the same data as *relational model* requires to store an equivalent database.

Chapter 3

3. Context Maps

3.1 Context Paradigm

From the source of Dr. Jaworski at the website of www.gen-strategies.com, *Context maps* introduces the concept of creating style sheets to control knowledge-based information access and navigation. *Context maps* enable us to create virtual information maps for the information system. In a technical sense, *Context maps* describe what an information set is about, by formally declaring topics, and by linking the relevant parts of the information set to the appropriate topics.

Context tuple is a generic association of set members cast in roles. In the extended spreadsheet a column of roles and the related set members define context tuple. From the graphical view, context tuple, in fact, is represented by a compound edge and the connected compound nodes. A directed edge object consists of tail object, middle object and head object. Context can be defined by an aggregation of context tuples. While context tuples represents action-able system behaviors, processes, tasks, procedures or programs. The aggregated context tuples will form a *context map*.

3.2 *jMap* Technology

The *joined maps* or *jMaps* is a notation and method for representing systems architecture, structures, processes and reusable templates. This technology was first introduced by Dr. W.M. Jaworski [1995]. The technology was initially developed as a means of recovering and refining knowledge from legacy systems. By using the popular concept of spreadsheet structure, it is feasible to describe and process conceptual information. The *jMaps* notation allows efficient recovery and modeling of generic schemata for processes, objects and views of information systems.

jMaps represents the knowledge in a spreadsheet format with the relationships represented by vertical *tuples/columns*. Connecting the words jointed and map produced the term "*jMap*". The *jMaps* represent the relationship between different information nodes and provide functionality of arrays, graphs, relational tables, etc. The 'j' stands for *jointed*, because a *jMap* can be a collection of different information connected together in a strong logical way. By that we mean that you can manipulate the logical query to get the specific information that you seek from the map.

3.3 *jMap* Syntax and Process

The syntax of *jMaps* is based on the Relationship Oriented paradigm, or on relating sets and set members. In *jMaps* the relationships are represented by (vertical) tuples/columns. The *kTuple* (knowledge tuple) construct is the fundamental structure defined by the concepts and instances related by roles.

The relating mechanism is implemented by allocating roles to sets in schema and their instance to set members/instances in map. Compared to diagrams, maps are very compact, offering a rich context within limited space of a computer screen. Maps are created or edited within an organized electronic page – spreadsheet which assures efficient manipulation of relationships (columns) and heavy reuse of components (row).

Figure 2 (source from W.M.Jaworski) demonstrates associations of descriptor strings to arcs and nodes. The character "f" ("t") associate the strings to the "tail" ("head') of an arc. The character "m" signifies that the string is attached to the "body" of arc or node. Clustering of arcs - and connected nodes - into graphs is shown by tagging columns with character "&". Graphs are connected if they share at least one node. As is illustrated by graph (A) and (B), reordering of columns and/or rows is an information-preservation operation, i.e. the shape of the graph might change but not the meaning. Descriptors of arcs and nodes are set members. Sets are identified by {<set name>} and are defined by enumeration. The schema-level view of a map is obtained first by hiding set instances and then by hiding redundant columns (Figure 3). The schema provides information about *joined maps* (*jMaps*) structure and size.

A	A	A	3	1	{Graph}
&	&	&	3		{A}
M	M	M	3	3	{Arc}
m			1		string4
	m		1		string5
		m	1		string6
F	F	F	3	3	{Node}
	t	f	2		string1
t	f	t	3		string2
f		t	2		string3

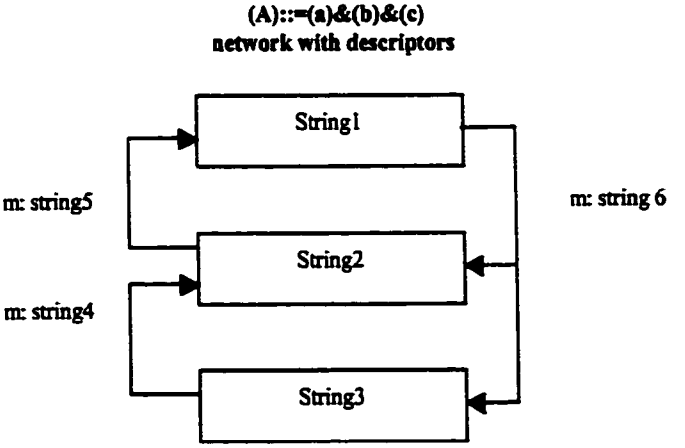


Figure 2 Diagrams Defined by Map Patterns

In Figure 2, for the diagrams on the right side, we can have the map as shown on the left side of figure. Map with Patterns contains three sets namely **{Graph}, {Arc}, and {Node}**. There are three roles namely **'A', 'M'** and **'F'** and four instance roles namely '&', 'm', 'f' and 't'. Role **'A'** was allocated to **{Graph}** to allow clustering of columns (i.e. relationships of instances) with instance role '&'. Role **'M'** was allocated to **{Arc}** to allow allocating of instance role 'm' to the instances 'string4', 'string5' and 'string6'. Role **'F'** was allocated to **{Node}** to allow allocating the instance roles 'f' and 't' to the members/instance of these sets.

A	A	A	3	1	{Graph}
&	&	&	3		{A}
M	M	M	3	3	{Arc}
F	F	F	3	3	{Node}

Figure 3 Schema view of Map with Pattern

If we need to develop large *jMaps* models, we can hide irrelevant columns and rows, editing visible cells and inserting new columns and new rows.

In general, abstract concepts appear on the right of the map in bold and between curly brackets. They can be thought of as a heading of a table column or a row. The instances would then be the actual contents listed in the table. Each column is to be read vertically using the syntax that was described above. For every "variable" you come across when reading down a column, you must read across towards the right of the map, to see which

concept or instance the variable is referring to. Beside each instance, and under the total number of instances, represents the number of times the instance is referred to.

3.4 The *Joined Map* Notation

jMap notation can addresses many topics such as following:

- Information system architecture
- Recovery and reuse of system patterns
- Evolving information systems
- Software evaluation and renewal
- Systems workstations
- Automation of system design
- Modeling of web sites and knowledge hubs

Following are explaining for some *jMap* notations

- **The concepts could be one of the following:**

  A - Template Aggregation

  T – Template

  Y – Dominant

  Z – Descriptive

  K – Identifier

  O – Identity

  H – Hierarchy

  I - Generalization - "parent" or "heir"

  P - Aggregation - "whole" or "part"

U - Uses or used

D - Dependence

S - Sequence - position in a sequence

F - Flow "from" or "to"

L - Flow "from", "to" and "loop"

X - Unique Qualifier

M - Association

G - Guard or Goal

E - Event

V - Value

? - User defined

- **The different instances that exist for the concepts:**

1 ... * - identifier or value

o - column marker

h - tree root

1 ... * - branch

f - from:

t - to:

b - both

m - many or middle:

d - destination:

s - source:

l - loop

a - assertion

e - exception

x - unique row marker

v - related

c - composite

t - true

f - false

o - otherwise

t - implied true

f – implied false

e – enabled

d - disabled

? - user defined

u - update

3.5 *Associative Model* Recovered with *jMaps*

We will go back to earlier *Customer* and *Order* tables (Table 1 and Table 2) and convert *associative model* data to *jMaps*. Rewriting of the *associative model* with *jMaps* should be done by performing of the following activities:

1) Identify component types i.e. identification of sets by name.
2) Enumerate sets and identify connector types
3) Create connectivity columns/map by `connecting' components with characters "f" and "t".
4) Use "M" to identify association. Enhance connectivity columns by using characters "m" to represent association between the attributes
5) Use characters "v" to stress uniqueness/identity of an entity.
6) Use characters "F" to identify columnwise for the sets with members connected by "f" or "t".
7) Create schema view by first hiding set members and then hiding redundant columns.

Products of the process for this example, i.e. relevant *jMaps* and schema are shown in Figure 4 and Figure 5 .

For more complex example as described in earlier Book Seller problem, Figure 6 shows recovered *associative model* with relevant *jMap* and schema.

A	A	A	A	A	A	A	A	8	4	{View}
V	V	V	V	V	V	V	V	8		ADM
A	A	A	A	A	A	A	A	8	2	{Entity}
V	V	V	V					4		Customers
				V	V	V	V	4		Orders
F	F	F	F	F	F	F	F	8	4	{Customer number}
F	F	F	F					4	4	{Name}
F	F	F	F					4	4	{Telephone no}
F	F	F	F					4	4	{Credit limit}
F	F	F	F					4	4	{O/S balance}
				F	F	F	F	4	4	{Order number}
				F	F	F	F	4	4	{Date}
				F	F	F	F	4	4	{Item}
				F	F	F	F	4	4	{Quantity}
M	M	M	M	M	M	M	M	8	1	{Association}

Figure 4 Schema of Customers and Order *Associative Model* represented by *jMaps*

			f		t	t		3		789
F	F	F	F					4	4	{Name}
t								1		Avis
	t							1		Boeing
		t						1		CA
			t					1		Dell
F	F	F	F					4	4	{Telephone no}
t								1		020 7123 4567
	t							1		020 8345 6789
		t						1		0123 45678
			t					1		0134 56789
F	F	F	F					4	4	{Credit limit}
t								1		£10,000
	t							1		£2,500
		t						1		£50,000
			t					1		£21,000
F	F	F	F					4	4	{O/S balance}
t								1		£4,567
	t							1		£1,098
		t						1		£14,567
			t					1		£6,789
				F	F	F	F	4	4	{Order number}
				f				1		11234
					f			1		11235
						f		1		11236
							f	1		11237
				F	F	F	F	4	4	{Date}
				t				1		02/03/1999
					t			1		15/03/99
						t		1		21/04/99
							t	1		07/05/1999
				F	F	F	F	4	4	{Item}
				t				1		ABC345
					t			1		GGI765
						t		1		KLM012
							t	1		GHJ999
				F	F	F	F	4	4	{Quantity}
				t				1		150
					t			1		25
						t		1		1000
							t	1		50
M	M	M	M	M	M	M	M	8	1	{Association}
m	m	m	m	m	m	m	m	8		has / is

Figure 5 Customer and Orders *Associative Model* represented by *jMap*

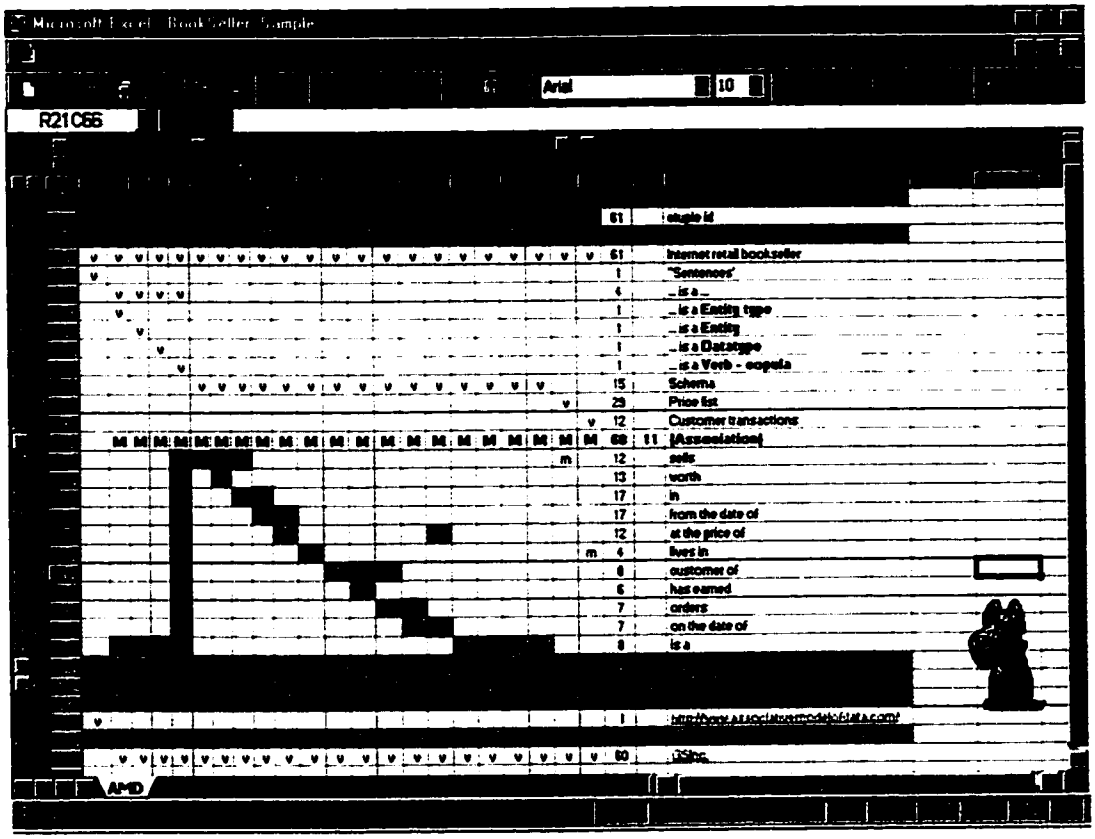


Figure 6 Book Seller Problem with *jMap* converted *Associative Model*

Chapter 4

4. Application Program

4.1 Introduction

This project deals with the recovery of the information structure knowledge from database, to generate the *jMap* (in the background) and then launches *MS Excel* with the resultant map. The program runs only on computers equipped with *MS Excel*. The subsequent reuse of this recovered knowledge can be represented as *associative data model*. The central element in the process of information manipulation is based on the *jMap* formal notation technology.

The program will provide the user with a number of options including the options to recover information from the database, and the options to normalize *jMap* sheet to save into the database. A comprehensive on-line help about what each of these options mean will be provided. For users seeking more detailed sample will also be provided.

4.2 Development Tool

The development tool is described as the follows: (most are members of the Microsoft family of products)

- **Microsoft Excel:**

Excel is a spreadsheet that allow you to organize data, complete calculations, make decisions, graph data, develop professional-looking reports, convert Excel files for use on the database, access the database.

The three major parts of Excel are:

1) Worksheets, that allow you to better calculate, manipulate, and analyze data such as numbers and text (the term worksheet means the same as spreadsheet.).
2) Charts, that pictorially represent data. Excel can draw a variety of two-dimensional and three-dimensional charts.
3) Databases, that manage data. For example, once you enter that data, you can search for specific data, and select data that meets the criteria.

- **Microsoft Access**

Microsoft Access is a database which makes difficult database technology accessible to general business users. *Microsoft Access* ensures that the benefits of using a database can be quickly realized. With its integrated technologies, *Microsoft Access* is designed to make it easy for all users to find answers, share timely information, and build faster solutions.

Microsoft Access has a powerful database engine and a robust programming language, making it suitable for many types of complex database applications. For small project, to chose *Microsoft Access* is suitable to store the data information.

- **VBA**

  A Visual Basic Application can provide us with the means to accomplish a wide range of the programmatic results. With *VBA*, we can create full-fledged custom applications in *Microsoft Excel.*

  Visual Basic support a set of objects that correspond directly to elements in *Microsoft Excel.* Every element in *Microsoft Excel*, such as workbook, worksheet, chart, cell, and so on, can be represented by an object in *Visual Basic.* By creating procedures that control these objects we can automate tasks in *Microsoft Excel.*

4.3 Project Functions

One of the main features needed for this project is the seamless nature of its operation. This entails minimum work by the system's user. The main system's functions as seen by the user can be summarized as:

1) Providing a mechanism through which the user can handle operation
2) Providing a mechanism for the user to enter his/her selected options,
3) Providing context-sensitive help,
4) Providing a visual indicator for the user to know the process' progress, and
5) Seeing the resultant *jMap* in *MS Excel.*

The system has more functions that are done in the background. These include:

1) Create the Unique Ids for each Context Tuple (aka Column ID) and Context Item (aka Set X Member ID), then introduce Unique IDs for each of Sets, Members, Spreadsheets, Workbooks, Tuples, Chapters, DBs etc.
2) The identifier is a surrogate, that is automatically assigned by the system.
3) Enable new item data be grouped into two tables: cTuple and cItem, and then could be saved into the database.
4) For obtained information from database, extracting and then refining the those data needed for the *jMap* generation.
5) Generating the associated data model *jMap* with the needed features, and launching *MS Excel* with the resultant *jMap*.
6) For both directions: database convert to *jMap* or *jMap* to Database, it will be taken care about the larger data with constrain of few spreadsheets and few workbooks
7) Query database and display results as *jMaps*.

4.4 General Constraints

The software is constrained only to run MS Windows operating system (WIN NT or WIN95/98/2000). The user also needs to know basic operations of *MS Excel*.

Chapter 5

5. Program User Manual

This manual concerns the extraction of an associative data model information and conversion of the selected information to and represented in *jMap* notation. The *jMap* is based on the *Excel* spreadsheets. Therefore, it is necessary for users to have elementary *Excel* knowledge.

5-1 System Requirement

Before you try to run this program, you need check if your system meets following requirements:

Hardware:

Pogrom shall operate with the following hardware requirements:

- CPU 486 or later
- Monitor – SVGA (800x600) or latter
- RAM – 16 MB
- Mouse or equivalent pointing device

Software:

You have to set up the following software in your machine

- Microsoft Excel
- Microsoft Access
- Visual Basic

Platform:

The program can run on the following platforms

- Windows 95
- Windows 98SE
- Windows NT
- Windows 2000

5-2 Start Program

In the software package, soon after open **ADMjMap.xls** file, there will be a Microsoft Excel popup dialogue as shown in following:

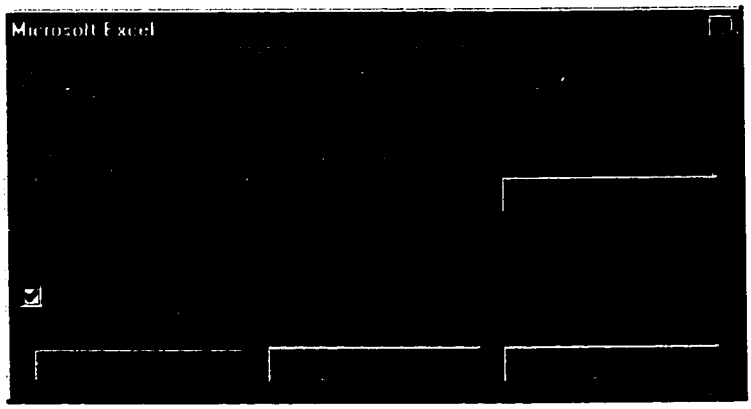


Figure 7 MS Excel Macros Enable Dialogue Interface

Click **Enable Macros** button to open the file, if you select **Disable Macros** button, then you will be unable to run the Macros in the program. After **Enable Macros** button is clicked, it will show following Welcome Interface:

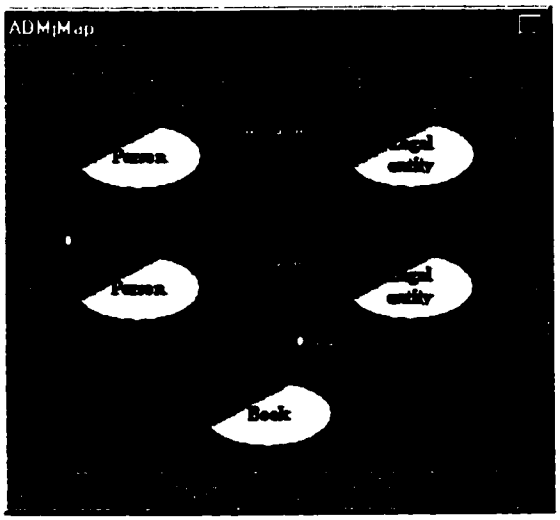


Figure 8 The Welcome Interface of ADMjMap Software

By clicking any area of welcome interface, you will hear one beep sound, after that the ADMjMap Excel file is ready to use.

5-3 Program Functionality

After ADMjMap is opened and is ready to use, you will find there is a *jMap* test sheet in the book. This test sheet is just for user to test the program's functionality. In Figure 9, you will find that a menu bar named ADMjMap has been created. When open this menu, as we can see, there are six operation sub-menus:

- **Create Tables**------to create the new sheet with generated ID for Sets, Members, Spreadsheets, and Workbooks.
- **Remove Tables** ------to remove the created sheets of {cItem}, {cTuples} and all sheet name which have brace {} covered will be removed
- **jMap->DB** ------to save the created tables to Access Database
- **DB->jMap** ------recover tables information from the Access Database with formatted *jMap* notation to the new sheet
- **DB jMap Analysis** ------analyze data tables from Database to produce the *jMap* results
- **Help** ------to get help information for using this program

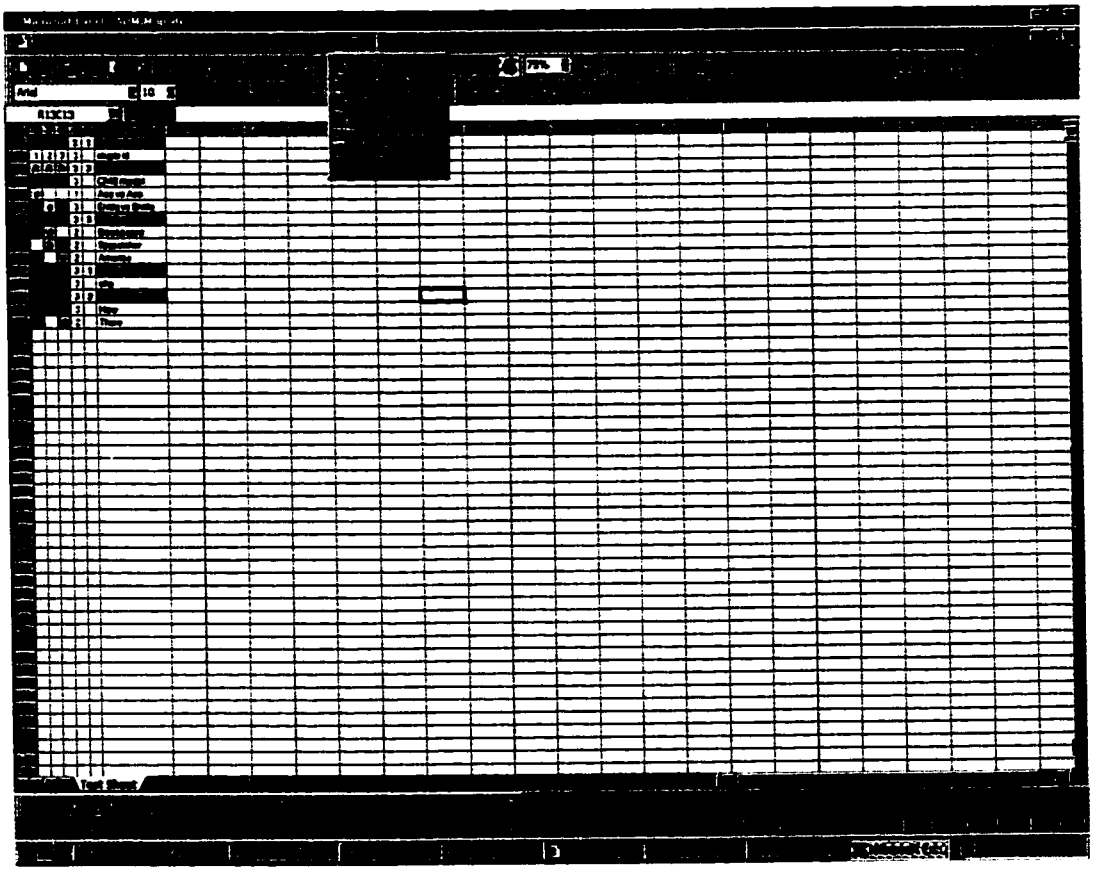


Figure 9 ADMjMap Menus and Test Sheet

5-4 Create Normalized *jMap* Tables

On the top menu **"ADMjMap"**, by clicking **"Create Tables"**.

- You will be asked to select sheet ID and book ID from a given Combo Box interface.

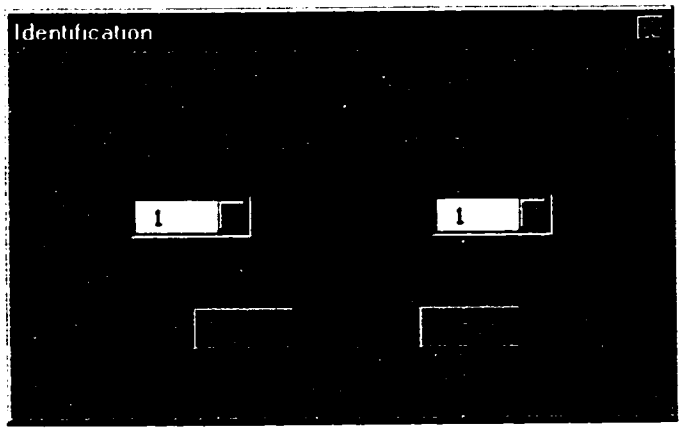


Figure 10 Sheets and Book Identify Dialogue

- Based on Normalized information from original active sheet, the program will create two tables which present as cItem and cTuple properties.
- It will create a new sheet name as: "< + "Original Sheet Name" + > ". This new sheet will present generated ID for Sets, Members, spreadsheet and workbook from the original sheet. The two tables will be in two new created sheet named as: {cItem} and {cTuples.
- If two sheet tables already exist, the created sheet name will be changed to {cItem}1 and {cTuples}1, or {cItem}2 and {cTuples}2, and so on. As the same, new sheet name for generated ID for Sets, Members, spreadsheet and workbook, if it exists, its name will also be updated with increasing number.

Figure 11 shows the created new cItem sheet based on generated ID for Sets, Members, spreadsheet and workbook from original sheet. Figure 12 also shows the results of new cTuple sheet normalized from original sheet.

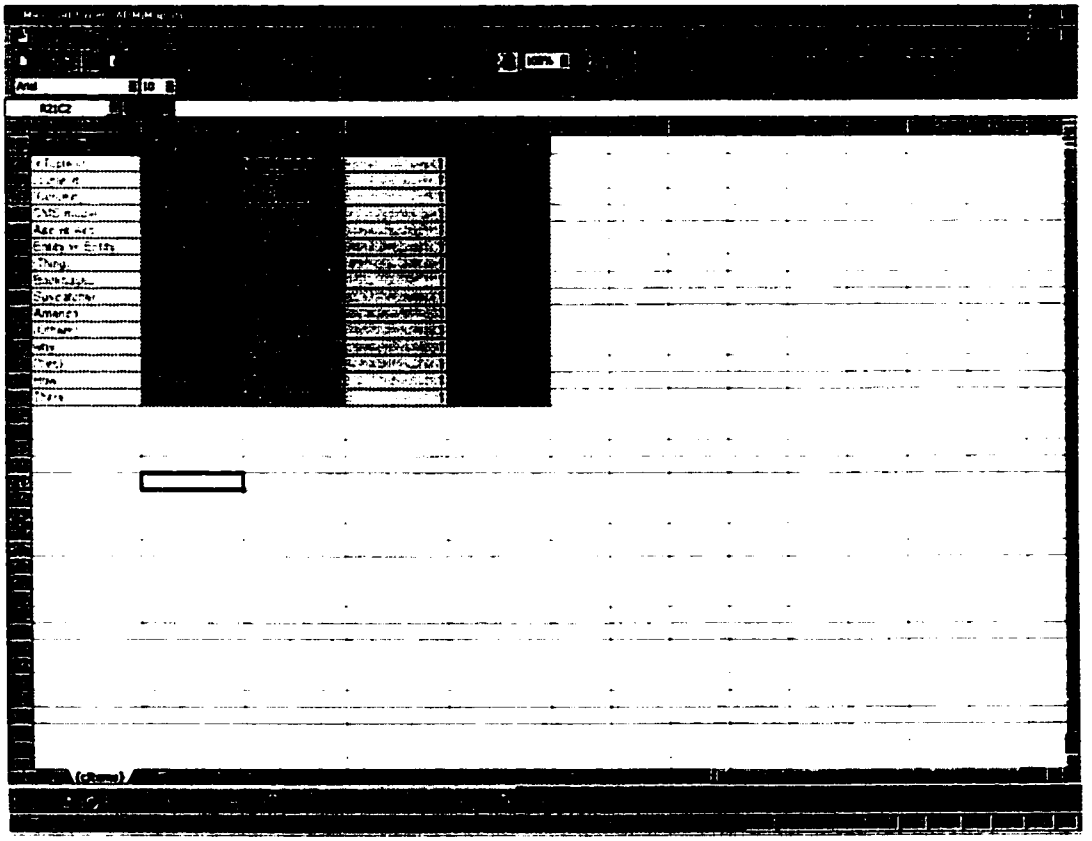


Figure 11 Created cItems Table Sheet

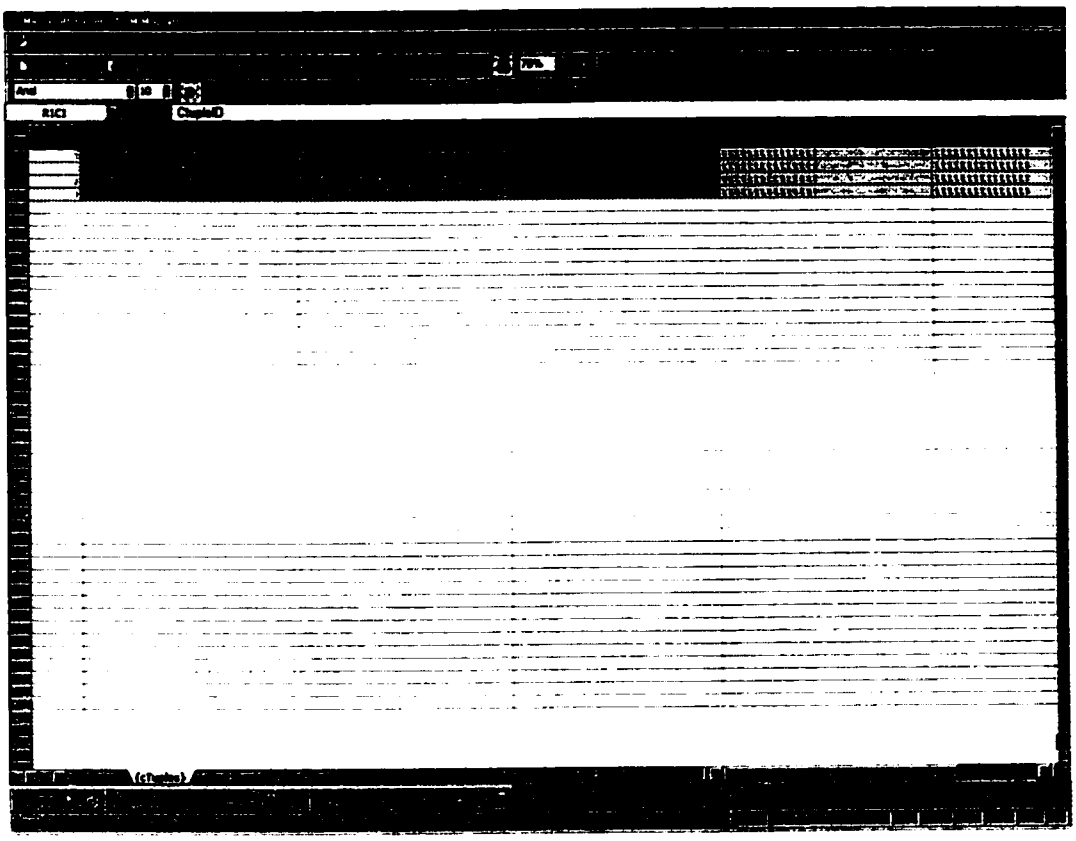


Figure 12 Created cTuples Table Sheet

Figure 13 shows normalized new sheet for original sheet in which the new sheet has been generated ID for Sets, Members, spreadsheet and workbook. This sheet information will be ready for creating the cItems and cTuple tables.

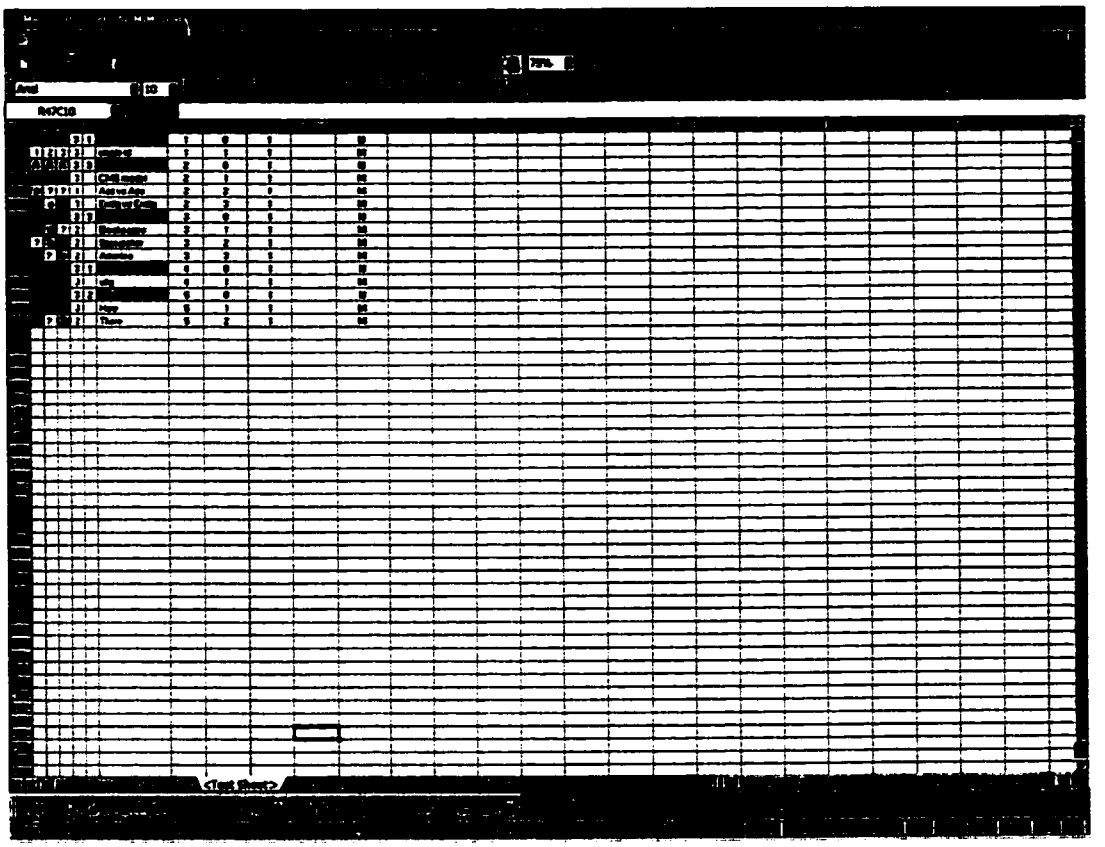


Figure 13 Normalized New Sheet

5-5 Remove Tables

On the top menu "**ADMjMap**", by clicking "**Remove Tables**", the created sheets of {cItem}, {cTuples} and all sheet names with {}or <> covered will be removed.

5-6 Save *jMap* to Database

On the top menu "**ADMjMap**", by clicking "**jMap->DB**", it will display a dialog box allowing the user to save the created tables to *Access Database.*

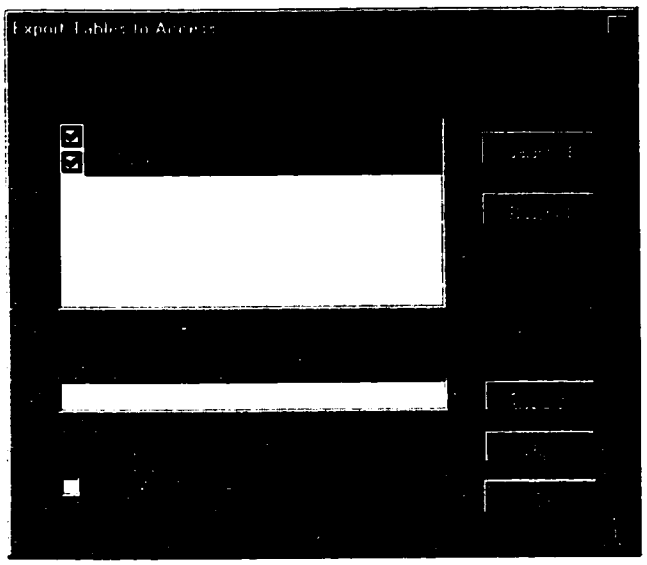


Figure 14 Export Tables to Access Dialog Box

In Figure 14, by clicking **Save As** button, the program will show following dialog box with default file name, if select Save button, the cItem Table and cTuple Table will be saved to *Access Database*. User can change the file name. If file name already exists, the tables information will be still added into this database in a changed table name as {cItem}1 and {cTuples}1, or {cItem}2 and {cTuples}2, and so on.

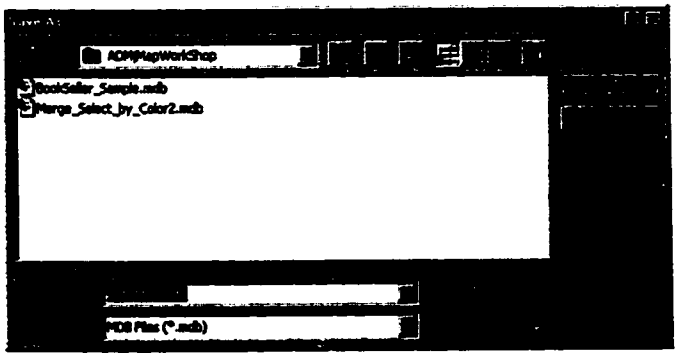


Figure 15 Save As Dialog Box

After the database file has been saved, the following message box will inform the user that the file has been saved.

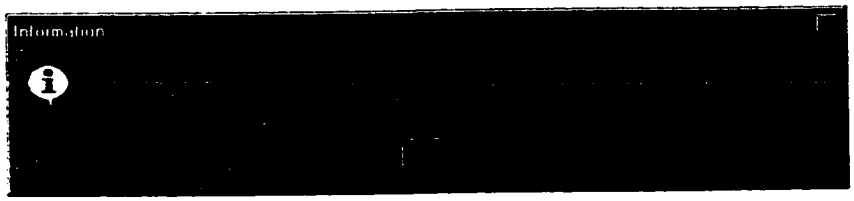


Figure 16 Information Message Box for Data Export from *jMap*

If check box "**Open Access after Export**" has been checked in the Export Tables to Access Dialog Box (see Figure 14), after Database File has been saved, computer system will automatically open the *Microsoft Access* for user to review the saved information.

5-7 Recover Database to *jMap*

On the top menu "**ADMjMap**", by clicking "**DB->jMap**", it will display a dialog box allowing the user to customize for recovering Database to *jMap*.

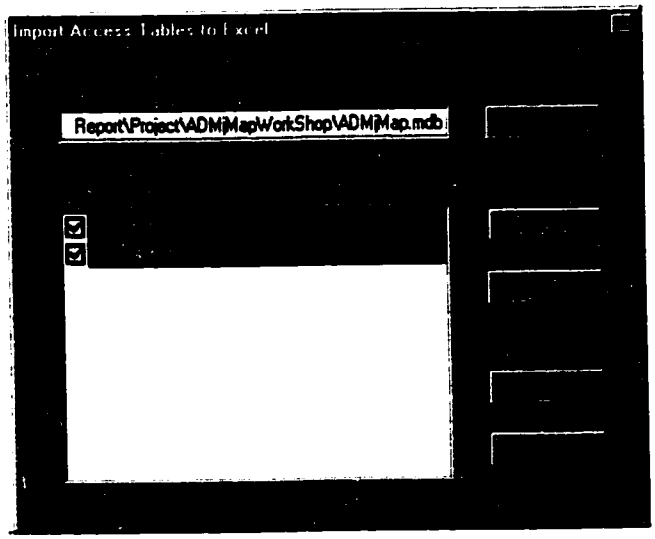


Figure 17 Import Tables to Excel Dialog Box

In Figure 17, after by clicking **Select All** button or check selected Table, with clicking **OK** button, the program will load the cItem Table and cTuple Table data to Excel

importing in different sheets. When it has been finished import Tables to the sheet, the following message box will display the file from the path has been import to Excel

Figure 18 Information Message Box for Data Import from Access

At end, the program will display a pop-up message box which will ask user if user wants to convert table information to *jMap*. Click **Yes** button, it will restore the *jMap* into the sheet.

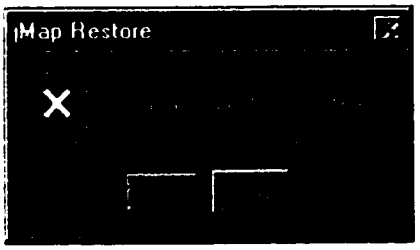


Figure 19 *jMap* Restoring Message Box

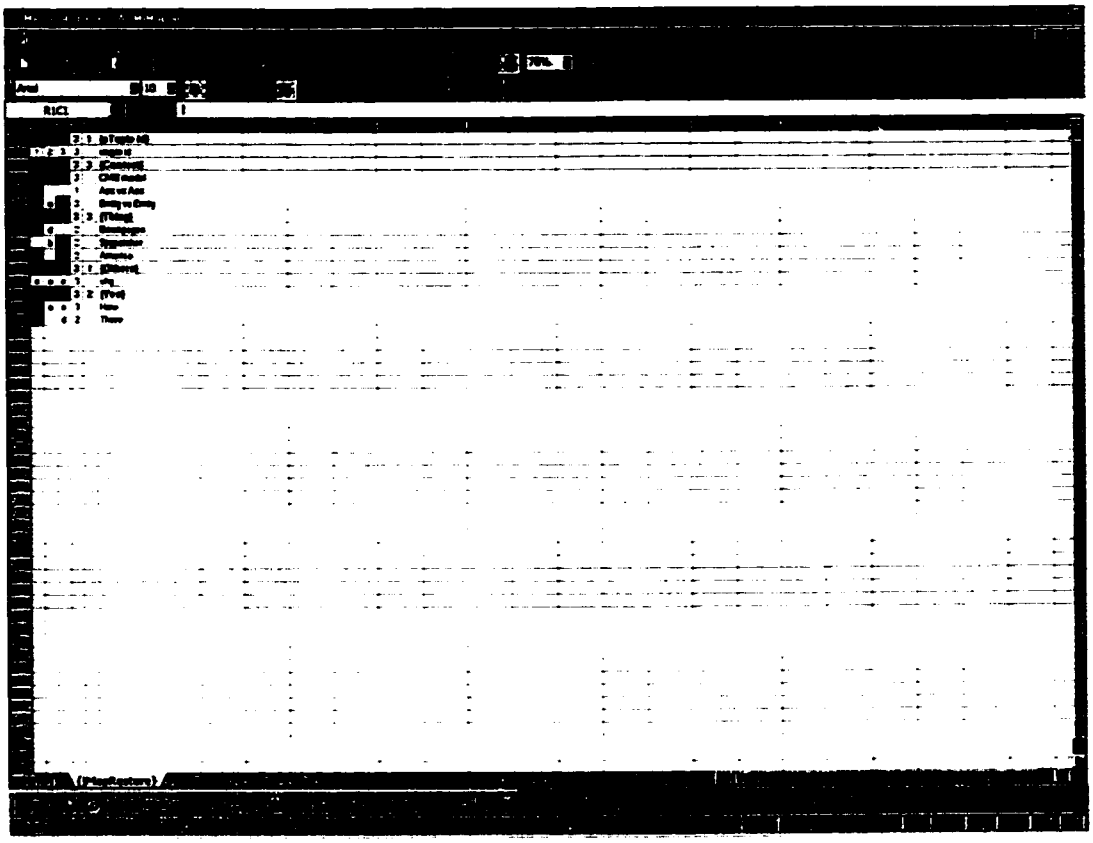


Figure 20 Restored *jMap* Results

5-8 Analyze Database Property with *jMap*

On the top menu "**ADMjMap**", by clicking "**DB jMap Analysis**", computer system will display a dialog box allowing the user to select a Table for analysis.

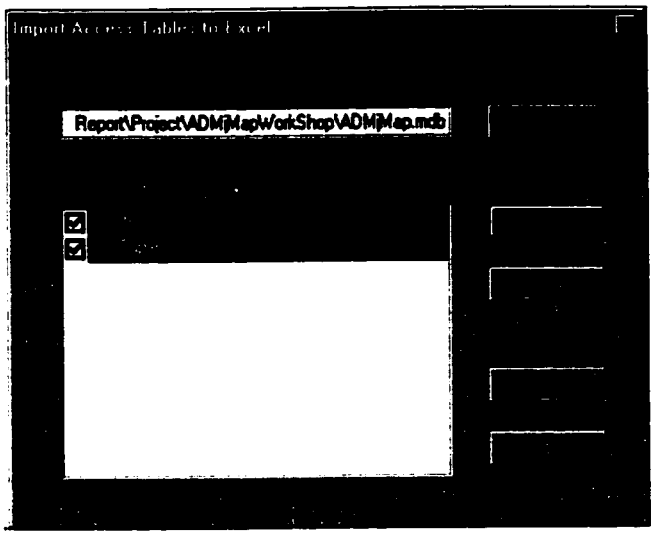


Figure 21 Import Tables to Excel Dialog Box for Analysis Table Properties

In Figure 21, after by clicking Select All button or check selected Table, with clicking OK button, it will analysis the saved tables information from the Access Database to produce the *jMap* results

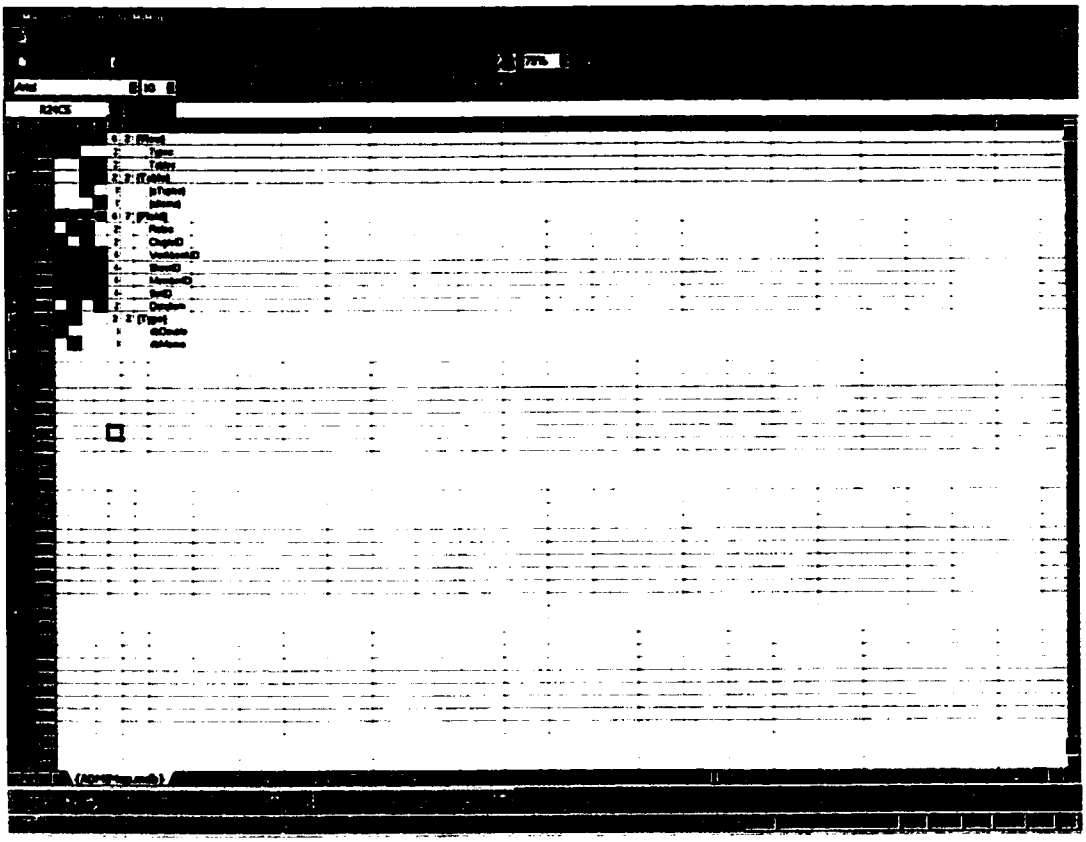


Figure 22 The Tables Analysis Results with *jMap* Notation

5-9 Get Help

On the top menu "**ADMjMap**", by clicking "**Help**", the program will open a help HTML web page for user to get help. Figure 20 shows this help page.

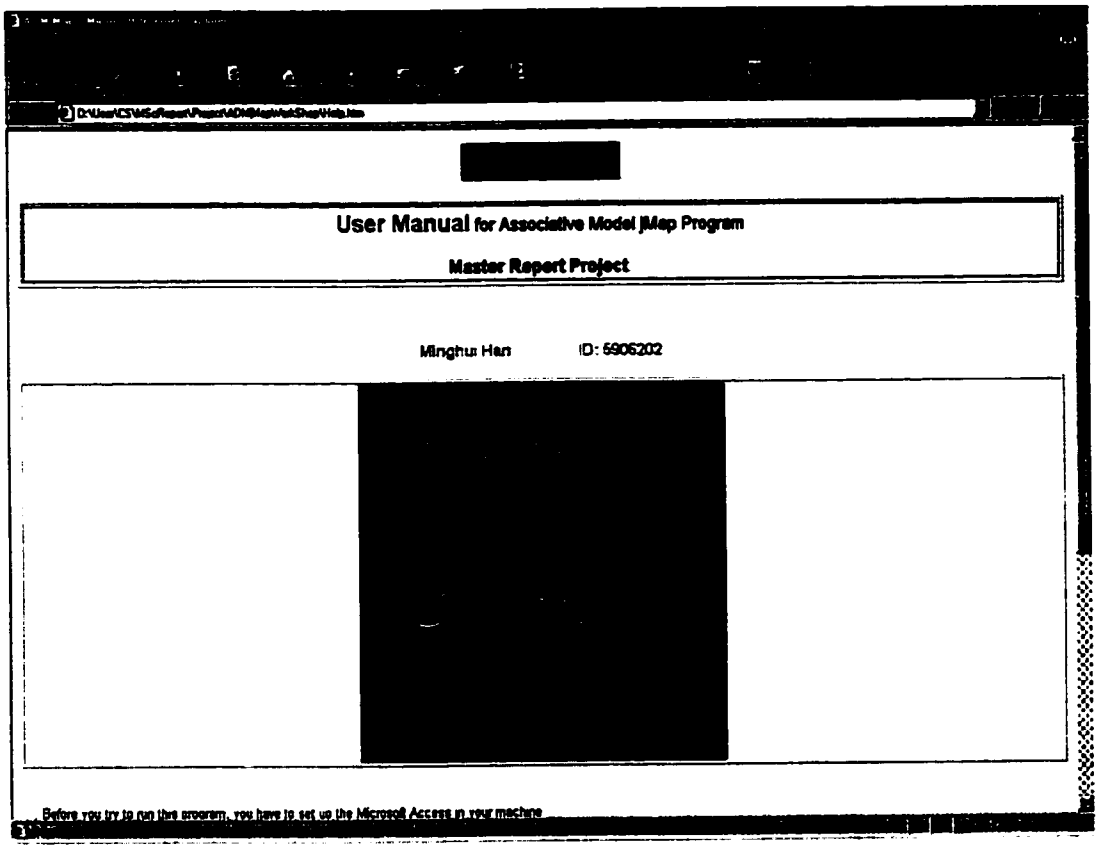


Figure 23 Program Help Page

Chapter 6

6. Conclusion And Recommendation

6.1 General Conclusion

The following conclusions are drawn from the results of this study:

1) The *associative model* views the information in the same way as the human brain, i.e. treats the things with association between them. Those associations can be expressed through the simple subject-verb-object syntax of an English sentence.
2) The *associative model* is simple. It overcomes the limitations of the *relational model* and avoids the complexities of the object model by structuring information in a more accessible and intuitive manner than either of the other model.
3) *Context maps* enable us to create virtual information maps for the information system. *Joined maps - jMaps* are a notation and method for representing systems architecture, structures, processes and reusable templates. The *jMaps* notation allows easy recovery and modeling of generic schemata for processes, objects and views of information systems.
4) *jMaps* syntax is simple and robust. *jMaps* models are pattern rich, allow users to specify, query and control the model views. Different views are generated algorithmically to be useful for compilers or end users
5) The *associative data* model can be presented as the *joined maps* (*jMaps*) of concepts and relationships using the popularly available *MS Excel* spreadsheet.

6) An application program was developed by considering *context maps* for associative data model. This program can present *context maps* exported into database or recovery data from a database to spreadsheets with *jMaps* notation which represented as the associative data model.

7) The application program can also treat any standard *jMap* sheet to convert *jMap* into a database system.

6.2 Recommendations for Future Works

From the results of this study, it is noted that there are still more detail works need to be carried out for improving use the application program. The following are recommended for future enhancement.

1) There is much future work in implementation of *joined maps* for dealing with complex systems. Future work is expected to lead to a better and more complete theory of *Context Maps.*

2) A more complete application program to convert *jMaps* into Database, or from Database to *jMaps*, needs to be developed.

3) In developed application program, to query different tables and data types from *Associative Model* database is necessary for future work.

4) For a larger data *jMap* sheet, it really takes time to get results in running the current program on a PC. It is necessary to improve program-running speed.

5) Designing of more user-friendly interface is yet another work needs to be done.

6) For large amounts of data, using *Excel* as a repository of *jMaps* has its limitations. Only 256 columns are available in the *Excel*. Although to some extent this project has considered this issue, to develop more efficient method for storing "context tuples" is necessary.

Bibliography

A- Printed Materials

1) **Grady Booch, James Rumbaugh, Ivar Jacobson,** "The UML User's Guide", Addison Wesley, 1998.

2) **Derek Coleman, etc.,** "Object-Oriented Development: the Fusion Method", Prentice-Hall, Inc., 1994.

3) **Minghui Han, etc.,** "jMapper, Web-Page *jMap* Generator For Key words and Keyphrase", Concordia University, COMP657, 2000.

4) **Minghui Han,** "jMapper, Web-Page *jMap* Generator For Key words, Keyphrase and XML tree view, Version 2.0", Concordia University, COMP695, 2000.

5) **W.M. Jaworski,** "Comp 457/657 Course Notes", Concordia University, 2000.

6) **W.M. Jaworski,** "*jMaps*: Conceptual Spreadsheets for Data and Knowledge", Warehousing, 1995.

7) **W.M. Jaworski,** "System Analysis and Design in the Classroom: InfoMAPs Teaching Factory", *Modeling and Simulation Conference*, Pittsburgh, Pa.,May 3-4, 1990.

8) **W.M. Jaworski,** "Michailidis A. A., Recovery and Enhancement of System Patterns: InfoSchemata and InfoMaps", *NATW94*, University of Massachusetts - Lowell, Massachusetts, June 1994.

9) **W.M. Jaworski,** "Conceptual Spreadsheets for Data and Knowledge Warehousing", *ATW95 - USA 1995*, University of New Hampshire, Durham, New Hampshire, May 31 - June 1, 1995.

10) **W.M. Jaworski,** "Cooperative Engineering Issues by Examples: Mapping of Mil498 and NSDIR with *jMaps*", *ATW96-USA 1996*, Electronic Systems Center, Hanscom Air Force Base, August 6-9, 1996.

11) **W.M. Jaworski,** "Representing Processes, Schemata and Templates with *jMaps*", *Expanded version of the paper presented at Conference on Notational Engineering (a.k.a. NOTATE96)*, The George Washington University, Washington, DC., May 23-25, 1996.

12) **W.M. Jaworski, Michailidis A. A.,** "Recovery and Enhancement of System Patterns: InfoSchemata and InfoMaps", *ATW '94*, University of Massachusetts - Lowell, Lowell, Massachusetts, June 1994.

13) **W.M. Jaworski,** "InfoMaps: Conceptual Spreadsheets for Data and Knowledge Warehousing", *ATW '95*, University of New Hampshire, Durham, New Hampshire, June 1995.

14) **W.M. Jaworski,** et al. "The ABL/W4 methodology for system modeling", *System Research Journal 4(1)*, 23-37, 1987.

15) **W.M. Jaworski,** et al. "Representing processes, schemata and templates with *jMaps*", *Semiotica* 125(1/3), 229-47, 1999.

16) **W.M. Jaworski,** "Representing System Schemata and Templates with *jMaps*", *NOTATE'96*, George Washington University, Washington, D.C., May 23-25, 1996.

17) **James Rumbaugh, etc.,** "Object-Orinted Modeling and Design, Prentice-Hall, Inc.", 1991.

18) **Ian Sommerville,** "Software Engineering", Addison-Wesley, 5th edition, 1995.

19) **Simon Williams,** "The *Associative Model* of Data", Lazy Software, 2000.

B- On-Line Sources

1) **General Strategies Inc.** http://www.gen-strategies.com

2) **Lazy Software,** http://www.lazysoft.com

3) **Lazy Software**, http://www.associativemodelofdata.com/

4) **Concordia University,** Thesis preparation and thesis examination regulations, http://www-gradstudies.concordia.ca/SGS_WWW/publications.html

5) **Rob Kremer**, A Concept Map Meta-Language, http://www.cpsc.ucalgary.ca/~kremer/dissertation/index.html

6) **Joseph D. Novak**, The Theory Underlying Concept Maps and How To Construct Them, http://cmap.coginst.uwf.edu/info/printer.html

Appendix Source Code

The program was coded by using VB language, the project consists of three parts: user forms, modules and class modules

- A user form contains user interface controls, such as command buttons and text boxes
- A module is a set of declarations followed by procedures—a list of instructions that a program performs.
- A class module defines an object, its properties, and its methods. A class module acts as a template from which an instance of an object is created at run time.

A-1 User Form Source Code

The source code for User Form includes as following created forms:

- **frmBookSheetInfo**
- **frmExportTablesToAccess**
- **frmImportAccessToWks**
- **frmWelcome**

All source code in above forms are listed as following:

A-1-1 frmBookSheetInfo

```
Option Explicit

Private Sub CancelButton_Click()
    On Error Resume Next

    Unload Me
End Sub


Private Sub OKButton_Click()

    Dim varBookID As String
    Dim varSheetID As String
    varBookID = ComboBox_Book.value
    varSheetID = ComboBox_Sheet.value
```

```
    Call MTables.CreateID(varSheetID, varBookID)
    Call MTables.CreateTables

    If MStartup.bjMaptoAccess = True Then

       frmExportTablesToAccess.Show

    End If

    Unload Me

End Sub

Private Sub UserForm_Initialize()
   Dim varCounter       ' Declare variables.

   For varCounter = 1 To 100    ' Count from 1 to 100.
      ComboBox_Book.AddItem varCounter    ' Add the Counter number for Book.
      ComboBox_Sheet.AddItem varCounter  ' Add the Counter number for Sheet
   Next varCounter

End Sub
```

A-1-2 frmExportTablesToAccess

```
'
'Purpose:   this form allows the user to select the worksheets from the active
'           workbook to export to access
'
Option Explicit

Private colSheets As Collection
Private blnOpenADBM As Boolean
Private blnSaveAsClicked As Boolean

Public Property Get SaveAsClicked() As Boolean
    SaveAsClicked = blnSaveAsClicked
End Property

Public Property Get OpenADBM() As Boolean
    OpenADBM = blnOpenADBM
End Property

Public Property Get SelectedSheets() As Collection
    Set SelectedSheets = colSheets
End Property

Private Sub EnableOKAsNecessary()
Dim lngItemCurr As Long

    cmdSaveAs.Enabled = False
    cmdOK.Enabled = False
    With lstTables
        For lngItemCurr = 0 To .ListCount - 1
            If .Selected(lngItemCurr) Then
                cmdSaveAs.Enabled = True
                cmdOK.Enabled = True
            End If
        Next lngItemCurr
    End With

End Sub

Private Sub chkOpenADBM_Click()
    If chkOpenADBM.value = -1 Then
        blnOpenADBM = True
    Else
        blnOpenADBM = False
    End If
```

```
End Sub

Private Sub cmdCancel_Click()
    On Error Resume Next

    Set colSheets = Nothing

    MTables.RemoveTables

    Unload Me


End Sub

Private Sub cmdResetAll_Click()
    On Error Resume Next

    ChangeSelection (False)

End Sub

Private Sub cmdOK_Click()
Dim lngItemCurr As Long

    If txtADBMName = "" Then
        MsgBox "Access Filename(*.mdb) must be entered", vbExclamation, "Error"
        Exit Sub
    End If

    If UCase(Right(txtADBMName, 4)) <> ".MDB" Then
        txtADBMName = txtADBMName + ".mdb"
    End If

    Set colSheets = New Collection

    With lstTables
        For lngItemCurr = 0 To .ListCount - 1
            If .Selected(lngItemCurr) Then
                colSheets.Add .List(lngItemCurr)
            End If
        Next lngItemCurr
    End With

    MExportTablesToAccess.Export txtADBMName
    MTables.RemoveTables

    Set colSheets = Nothing
    Unload Me

End Sub

Private Sub ChangeSelection(ByVal Selected As Boolean)
    Dim lngItemCurr As Long

    On Error Resume Next

    With lstTables
        For lngItemCurr = 0 To .ListCount - 1
            .Selected(lngItemCurr) = Selected
        Next lngItemCurr
    End With

End Sub

Private Sub cmdSaveAs_Click()
    'Defines the variable as a variant data type
    Dim X As Variant

    'Opens the dialog
    X = Application.GetSaveAsFilename(, "MDB Files (*.mdb), *.mdb", 2, "Save As")
```

```
    If X <> False Then
        txtADBMName.Text = X
        blnSaveAsClicked = True
    End If

    txtADBMName.SetFocus

End Sub

Private Sub cmdSelectAll_Click()
    On Error Resume Next

    ChangeSelection (True)

End Sub

Private Sub lstTables_Change()
    EnableOKAsNecessary

End Sub

Private Sub lstTables_Click()
    EnableOKAsNecessary

End Sub

Private Sub UserForm_Initialize()
Dim Wks As Worksheet

    chkOpenADBM.value = 0
    blnOpenADBM = False
    cmdSaveAs.Enabled = False
    blnSaveAsClicked = False
    cmdOK.Enabled = False
    txtADBMName.Text = ""
    lstTables.Clear

    For Each Wks In Worksheets
        If Wks.type = xlWorksheet Then
            If Wks.Visible Then
                If InStr(Wks.Name, "(") Then
                    lstTables.AddItem (Wks.Name)
                End If
            End If
        End If
    Next Wks

    Dim lngItemCurr As Long

    On Error Resume Next

    With lstTables
        For lngItemCurr = 0 To .ListCount - 1
            .Selected(lngItemCurr) = True
        Next lngItemCurr
    End With

End Sub
```

A-1-3 frmImportAccessToWks

```
'
'Purpose:   this form allows the user to specify an access database and choose
'           which tables to import from access

Option Explicit

Private colTables As Collection
```

```
Private blnBrowseClicked As Boolean

Public Property Get BrowseClicked() As Boolean
    BrowseClicked = blnBrowseClicked
End Property

Public Property Get SelectedTables() As Collection
    Set SelectedTables = colTables
End Property

Private Sub EnableOKAsNecessary()
Dim lngItemCurr As Long

    cmdOK.Enabled = False
    With lstTables
        For lngItemCurr = 0 To .ListCount - 1
            If .Selected(lngItemCurr) Then
                cmdOK.Enabled = True
            End If
        Next lngItemCurr
    End With

End Sub

Private Sub cmdBrowse_Click()
    'Defines the variable as a variant data type
    Dim X As Variant

    blnBrowseClicked = False

    'Opens the dialog
    X = Application.GetOpenFilename("MDB Files (*.mdb), *.mdb", 2, "Open", ,
False)

    If X <> False Then
        blnBrowseClicked = True
        txtADBMName.Text = X
        ListTables
    End If

    txtADBMName.SetFocus

End Sub

Private Sub cmdCancel_Click()
    On Error Resume Next

    Set colTables = Nothing
    Unload Me

End Sub

Private Sub cmdResetAll_Click()
    On Error Resume Next

    ChangeSelection (False)

End Sub

Private Sub ListTables()
Dim wrkdefault As Workspace
Dim db As Database
Dim tblList As TableDef
Dim Message As String
Dim Title As String

    On Error GoTo Handler

    ' Get default Workspace.
    Set wrkdefault = DBEngine.Workspaces(0)
```

```
    ' Open database
    If blnBrowseClicked = True Then
        Set db = wrkdefault.OpenDatabase(txtADBMName)
    Else
        Set db = wrkdefault.OpenDatabase(ActiveWorkbook.Path & "\" & txtADBMName)
    End If

    lstTables.Clear

    ' Fetch all the tables
    For Each tblList In db.TableDefs
        If Left(tblList.Name, 4) <> "MSys" Then
            lstTables.AddItem (tblList.Name)
        End If
    Next

    Set db = Nothing

    lstTables.Enabled = True
    cmdSelectAll.Enabled = True
    cmdResetAll.Enabled = True

    Exit Sub

Handler:
    Message = _
                    "Error Number      : " & Err _
        & Chr(10) & "Error Description: " & Error()

    Title = "An error has occured"
    MsgBox Message, , Title
    Message = ""
    Title = ""
    cmdResetAll_Click
    lstTables.Clear

End Sub

Private Sub cmdOK_Click()
    Dim lngItemCurr As Long

    'will add the question Box
    Dim Msg, Style, Title, Help, Ctxt, Response, MyString
    Msg = "  Do you want to make jMap?  "     ' Define message.
    Style = vbYesNo + vbCritical + vbDefaultButton1     ' Define buttons.
    Title = "jMap Restore"     ' Define title.
    Help = "DEMO.HLP"
    Ctxt = 1000     ' Define topic
        ' context.
        ' Display message.

    If txtADBMName = "" Then
        MsgBox "Access Filename must be entered", vbExclamation, "Error"
        Exit Sub
    End If

    If UCase(Right(txtADBMName, 4)) <> ".MDB" Then
        txtADBMName = txtADBMName + ".mdb"
    End If

    Set colTables = New Collection

    With lstTables
        For lngItemCurr = 0 To .ListCount - 1
            If .Selected(lngItemCurr) Then
                colTables.Add .List(lngItemCurr)
            End If
        Next lngItemCurr
    End With

    If MStartup.bAccesstojMap = True Then
```

```
        MImportAccessToWks.Import txtADBMName

        Response = MsgBox(Msg, Style, Title, Help, Ctxt)
        If Response = vbYes Then    ' User chose Yes.

            MRestorejMap.MapTable

        End If

    Else

        MAnalysisAccessToWks.Import txtADBMName

    End If

    Set colTables = Nothing

   Unload Me

End Sub

Private Sub ChangeSelection(ByVal Selected As Boolean)
Dim lngItemCurr As Long

    On Error Resume Next

    With lstTables
        For lngItemCurr = 0 To .ListCount - 1
            .Selected(lngItemCurr) = Selected
        Next lngItemCurr
    End With

End Sub


Private Sub cmdSelectAll_Click()
    On Error Resume Next

    ChangeSelection (True)

End Sub

Private Sub lstTables_Change()
    EnableOKAsNecessary

End Sub

Private Sub lstTables_Click()
    EnableOKAsNecessary

End Sub

Private Sub txtADBMName_AfterUpdate()

    If txtADBMName <> "" Then
        ListTables
    End If

End Sub

Private Sub UserForm_Initialize()

    blnBrowseClicked = False
    txtADBMName.Text = ""
    lstTables.Clear

    cmdSelectAll.Enabled = False
    cmdResetAll.Enabled = False
    lstTables.Enabled = False
    cmdOK.Enabled = False
```

```
End Sub
```

A-1-4 frmWelcome

```
' Show welcome interface when open the workbook
Sub show_Beep()

    On Error Resume Next

    Beep
    Show

End Sub

Private Sub UserForm_Click()
    Beep
    End
End Sub
```

A-2 Modules Source Code

The source code for Modules includes as following created modules:

- **MAnalysisAccessToWks**
- **MColor**
- **MExportTablesToAccess**
- **MimportAccessToWks**
- **MRestorejMap**
- **MShellExecute**
- **MStartup**
- **MTables**

All source code in above modules are listed as following:

A-2-1 MAnalysisAccessToWks

```
Option Explicit

Private db As Database

Sub Import(strADBM As String)
'
'Purpose:   imports an access database into an excel workbook and builds a jmap
'Arguments: string containing the database to import

    On Error GoTo Handler

    'create a jmap object
    Dim map As New AccessJMapBuilder

    ' Get default Workspace.
    Dim wrkdefault As Workspace
```

```
    Set wrkdefault = DBEngine.Workspaces(0)

    ' Open database
    Dim db As Database

    If frmImportAccessToWks.BrowseClicked = True Then
        Set db = wrkdefault.OpenDatabase(strADBM)
    Else
        Set db = wrkdefault.OpenDatabase(ActiveWorkbook.Path & "\" & strADBM)
    End If



    'name the sheet
    map.NameSheet strADBM

    'insert some set's and set members
    map.InsertSetMember "View", "Tables"
    map.InsertSet "Table", "F"
    map.InsertSet "Field", "N"
    map.InsertSetMember "View", "Types"
    map.InsertSet "Type", "F"

    Dim colTables As Collection
    Set colTables = frmImportAccessToWks.SelectedTables

    'For every selected table, import table information
    Dim Tb
    Dim Rs As Recordset
    Dim I As Integer
    Dim RsSql As String
    For Each Tb In colTables

        RsSql = "SELECT * FROM [" & Tb & "]"
        Set Rs = db.OpenRecordset(RsSql, dbOpenDynaset)

        'insert a set member for the tables set
        map.InsertSetMember "Table", Tb

        ' Loop through the Microsoft Access field names and insert into
        ' the set of fields
        map.AddColumn
        For I = 0 To Rs.fields.Count - 1
            map.InsertSetMember "Field", Rs.fields(I).Name
            map.InsertAssociation "Table", Tb, "f", "Field", Rs.fields(I).Name,
"t", "Tables"
        Next I
    Next Tb

    'For every selected table, get the type information
    For Each Tb In colTables
        ' Loop through the Microsoft Access field types and insert into
        ' the set of types
        RsSql = "SELECT * FROM [" & Tb & "]"
        Set Rs = db.OpenRecordset(RsSql, dbOpenDynaset)

        For I = 0 To Rs.fields.Count - 1
            If map.FindSetMember("Type", FieldType(Rs.fields(I).type)) = False
Then
                map.AddColumn
                map.InsertSetMember "Type", FieldType(Rs.fields(I).type)
            End If
            map.InsertAssociation "Type", FieldType(Rs.fields(I).type), "f",
"Field", Rs.fields(I).Name, "t", "Types"
        Next I
    Next Tb

    'Close the database
    db.Close

    'group the sets
```

```
    map.DoRowGrouping "View"
    map.DoRowGrouping "Table"
    map.DoRowGrouping "Field"
    map.DoRowGrouping "Type"

    MColor.ColorItem

    MsgBox "Data Imported from " & strADBM & " to " & ActiveWorkbook.Name,
vbInformation, "Information"

    Exit Sub

Handler:
    Dim Message As String
    Dim Title As String
    Message = _
                  "Error Number      : " & Err _
        & Chr(10) & "Error Description: " & Error()

    Title = "An error has occured"
    MsgBox Message, , Title
    Message = ""
    Title = ""

End Sub

Function FieldType(intType As Integer) As String
'
'Purpose:   converts field type integer to return a field type string
    Select Case intType
        Case dbBoolean
            FieldType = "dbBoolean"
        Case dbByte
            FieldType = "dbByte"
        Case dbInteger
            FieldType = "dbInteger"
        Case dbLong
            FieldType = "dbLong"
        Case dbCurrency
            FieldType = "dbCurrency"
        Case dbSingle
            FieldType = "dbSingle"
        Case dbDouble
            FieldType = "dbDouble"
        Case dbDate
            FieldType = "dbDate"
        Case dbText
            FieldType = "dbText"
        Case dbLongBinary
            FieldType = "dbLongBinary"
        Case dbMemo
            FieldType = "dbMemo"
        Case dbGUID
            FieldType = "dbGUID"
    End Select

End Function
```

A-2-2 MColor

```
Sub ColorItem()
  Dim colNum As Integer
  Dim rowNum As Integer
  Dim rgnSheet As Excel.Range

  AutoAqua = RGB(60, 186, 196)
  AutoLime = RGB(153, 178, 51)
  Autogreen = RGB(0, 251, 0)
```

```
autored = RGB(255, 0, 0)
AutoLightOrg = RGB(222, 144, 51)
AutoPink = RGB(255, 0, 255)
Autopaleblue = RGB(153, 204, 255)
AutoLightPink = RGB(255, 166, 205)
AutoYellow = RGB(238, 192, 65)
AutoBrightYellow = RGB(255, 255, 0)
AutoGray = RGB(128, 128, 128)
AutoLightPurple = RGB(204, 137, 255)
AutoPurple = RGB(255, 0, 255)
AutoDarkGreen = RGB(0, 95, 0)
AutoBrightBlue = RGB(0, 204, 255)

Set rgnSheet = ActiveSheet.UsedRange

For colNum = 1 To rgnSheet.Columns.Count
  For rowNum = 1 To rgnSheet.Rows.Count

  If UCase(rgnSheet.Cells(rowNum, colNum).value) = "" Then
       rgnSheet.Cells(rowNum, colNum).Interior.ColorIndex = xlNone
     ElseIf UCase(rgnSheet.Cells(rowNum, colNum).value) = "A" Then
       rgnSheet.Cells(rowNum, colNum).Interior.ColorIndex = 16
     ElseIf UCase(rgnSheet.Cells(rowNum, colNum).value) = "V" Then
       rgnSheet.Cells(rowNum, colNum).Interior.ColorIndex = 15
     ElseIf UCase(rgnSheet.Cells(rowNum, colNum).value) = "E" Then
       rgnSheet.Cells(rowNum, colNum).Interior.Color = AutoLime
     ElseIf UCase(rgnSheet.Cells(rowNum, colNum).value) = "T" Then
       rgnSheet.Cells(rowNum, colNum).Interior.Color = Autogreen
     ElseIf UCase(rgnSheet.Cells(rowNum, colNum).value) = "F" Then
       rgnSheet.Cells(rowNum, colNum).Interior.Color = autored
     ElseIf UCase(rgnSheet.Cells(rowNum, colNum).value) = "M" Then
       rgnSheet.Cells(rowNum, colNum).Interior.Color = AutoLightOrg
     ElseIf UCase(rgnSheet.Cells(rowNum, colNum).value) = "L" Then
       rgnSheet.Cells(rowNum, colNum).Interior.Color = AutoPink
     ElseIf UCase(rgnSheet.Cells(rowNum, colNum).value) = "S" Then
       rgnSheet.Cells(rowNum, colNum).Interior.Color = AutoBrightYellow
     ElseIf UCase(rgnSheet.Cells(rowNum, colNum).value) = "N" Then
       rgnSheet.Cells(rowNum, colNum).Interior.Color = AutoLightPink
     ElseIf UCase(rgnSheet.Cells(rowNum, colNum).value) = "V" Then
       rgnSheet.Cells(rowNum, colNum).Interior.Color = AutoYellow
     ElseIf UCase(rgnSheet.Cells(rowNum, colNum).value) = "I" Then
       rgnSheet.Cells(rowNum, colNum).Interior.Color = AutoGray
     ElseIf UCase(rgnSheet.Cells(rowNum, colNum).value) = "G" Then
       rgnSheet.Cells(rowNum, colNum).Interior.Color = AutoDarkGreen
     ElseIf UCase(rgnSheet.Cells(rowNum, colNum).value) = "X" Then
       rgnSheet.Cells(rowNum, colNum).Interior.Color = AutoLightPurple
     ElseIf UCase(rgnSheet.Cells(rowNum, colNum).value) = "R" Then
       rgnSheet.Cells(rowNum, colNum).Interior.Color = AutoBrightBlue
     ElseIf UCase(rgnSheet.Cells(rowNum, colNum).value) = "L" Then
       rgnSheet.Cells(rowNum, colNum).Interior.Color = AutoPurple
     ElseIf IsNumeric(rgnSheet.Cells(rowNum, colNum).value) Then
       rgnSheet.Cells(rowNum, colNum).Font.Color = autored
    End If
  Next rowNum
 Next colNum
End Sub
```

A-2-3 MExportTablesToAccess

```
Option Explicit
Private db As Database

Sub Export(strADBM As String)
'
'Purpose:   exports worksheets from the active workbook into access
'Arguments: string containing the database name to create in access

    Dim colSheets As Collection
    Dim Wks, WksTemp
    Dim wrkdefault As Workspace
```

```
    Dim dataSource As String
    Dim Message As String
    Dim Title As String
    Dim TbIndex As Integer

    On Error GoTo Handler

    ' Get default Workspace.
    Set wrkdefault = DBEngine.Workspaces(0)

    ' Create a new encrypted database
    If frmExportTablesToAccess.SaveAsClicked = True Then
        Set db = wrkdefault.CreateDatabase(strADBM, dbLangGeneral, dbEncrypt)
    Else
        Set db = wrkdefault.CreateDatabase(ActiveWorkbook.Path & "\" & strADBM,
dbLangGeneral, dbEncrypt)
    End If

    Set colSheets = frmExportTablesToAccess.SelectedSheets

    'Create a new Table, and use the Worksheet Name as the
    'Table Name. Or Change the Table if the name already exist
    Dim tdfLoop As TableDef

    'For every selected worksheet, export to access
    For Each Wks In colSheets
        WksTemp = Wks
        TbIndex = 1
        With db

        ' Enumerate TableDefs collection.
        For Each tdfLoop In .TableDefs
            'For every table, compare if it exist
            If Wks = tdfLoop.Name Then
                Worksheets(Wks).Select
                Wks = WksTemp + Format(TbIndex)
                ActiveSheet.Name = Wks
                TbIndex = TbIndex + 1
            End If
         Next tdfLoop

        End With

        WksToAccess (Wks)

    Next

    MsgBox "Data Exported from " & ActiveWorkbook.Name & " to " & strADBM,
vbInformation, "Information"

    'Close the database

    db.Close

    'Check whether the open mdb flag is set.  If so, open the newly created access
database.
    'If user clicked on SaveAs, do not access the MDB file using the path name
    'If user entered the MDB filename, then insert active workbook path name in
the MDB string to
    'avoid "file not found" error when opening the database in access.

    If frmExportTablesToAccess.OpenADBM = True Then
        If frmExportTablesToAccess.SaveAsClicked = True Then
            ShellExec strADBM
        Else
            ShellExec ActiveWorkbook.Path & "\" & strADBM
        End If
    End If

    Exit Sub
```

```
Handler:

    ' Error 3204 means that the database already exist
    If DBEngine.Errors(0).Number = 3204 Then
        ' Open the database
         Set db = wrkdefault.OpenDatabase(strADBM)
         Resume Next

    Else
        Message = _
                    "Error Number     : " & Err _
        & Chr(10) & "Error Description: " & Error()

        Title = "An error has occured"
        MsgBox Message, , Title
        Message = ""
        Title = ""

    End If

End Sub

Sub WksToAccess(ByVal Wks)
'
'Purpose:   exports worksheets in active workbook to access
'Arguments: worksheet object
'Returns:

' Declare variables.
Dim Rs As Recordset
Dim td As TableDef
Dim Fd As Field
Dim X As Integer
Dim f As Integer
Dim r As Integer
Dim c As Integer
Dim Message As String
Dim Title As String
Dim LastColumn As Integer
Dim NumberTest As Double
Dim StartCell As Object
Dim LastCell As Object
Dim Response
Dim CreateFieldFlag As Integer
Dim flag As Integer

     CreateFieldFlag = 0
     flag = 0

     ' Turn off Screen Updating.
     Application.ScreenUpdating = False
     On Error GoTo ErrorHandler

     ' Select the worksheet and Cell "A1."
     ' In this example, you need column headers in the first row.
     ' These headers will become field names.
     Worksheets(Wks).Select
     Range("A1").Select

     ' If the ActiveCell is blank, open a message box.
     If ActiveCell.value = "" Then
         Message = "There is no data in the active cell: " & _
             ActiveSheet.Name & "!" & ActiveCell.Address & Chr(10) & _
             "Please ensure that all your worksheets have data on " & _
             "them " & Chr(10) & _
             "and the column headers start in cell A1" & Chr(10) & _
             Chr(10) & "This process will now end."

         Title = "Data Not Found"
```

```
    MsgBox Message, , Title
    Exit Sub
End If


Set td = db.CreateTableDef(Wks)

' Find the number of fields on the sheet and store the number
' of the last column in a variable.
Selection.End(xlToRight).Select
LastColumn = Selection.Column

' Select the current region. Then find what the address
' of the last cell is.
Selection.CurrentRegion.Select
Set LastCell = Range(Right(Selection.Address, _
    Len(Selection.Address) - _
    Application.Search(":", Selection.Address)))

' Go back to cell "A1."
Range("A1").Select

' Enter a loop that will go through the columns and
' create fields based on the column header.
For f = 1 To LastColumn
    flag = 0

    ' Enter a select case statement to determine
    ' the cell format.
    Select Case Left(ActiveCell.Offset(1, 0).NumberFormat, 1)
        Case "G"    'General format
            ' The "General" format presents a special problem.
            ' See above discussion for explanation
            If ActiveCell.value Like "*Zip*" Then
                Set Fd = td.CreateField(ActiveCell.value, _
                    dbMemo)
                Fd.AllowZeroLength = True
                r = LastCell.row - 1
                flag = 1
            Else
                If ActiveCell.value Like "*Postal*" Then
                    Set Fd = td.CreateField(ActiveCell.value, _
                        dbMemo)
                    Fd.AllowZeroLength = True
                    r = LastCell.row - 1
                    flag = 1
                End If
            End If

            ' Set up a test to determine if the field contains
            ' "Text" or "Numbers."
            For r = 1 To LastCell.row - 1
                If flag = 1 Then r = LastCell.row
                CreateFieldFlag = 1
                NumberTest = ActiveCell.Offset(r, 0).value / 2
            Next r

            ' If we get all the way through the loop without
            ' encountering an error, then all the values are
            ' numeric, and we assign the data type to be "dbDouble"
            If flag = 0 Then

               Set Fd = td.CreateField(ActiveCell.value, dbDouble)
            End If

        ' Check to see if the cell below is formatted as a date.
        Case "m", "d", "y"
            Set Fd = td.CreateField(ActiveCell.value, dbDate)

        ' Check to see if the cell below is formatted as currency.
```

```
            Case "$", "_"
                Set Fd = td.CreateField(ActiveCell.value, dbCurrency)

            ' All purpose trap to set field to text.
            Case Else
                Set Fd = td.CreateField(ActiveCell.value, dbMemo)
            End Select

        ' Append the new field to the fields collection.
        td.fields.Append Fd

        ' Move to the right one column.
        ActiveCell.Offset(0, 1).Range("A1").Select

    ' Repeat the procedure with the next field (column).
    Next f

    ' Append the new Table to the TableDef collection.
    db.TableDefs.Append td

    ' Select Cell "A2" to start the setup for moving the data from
    ' the worksheet to the database.
    Range("A2").Select

    ' Define the StartCell as the Activecell. All record addition
    ' will be made relative to this cell.
    Set StartCell = Range(ActiveCell.Address)

    ' Open a recordset based on the name of the activesheet.
    Set Rs = db.OpenRecordset(Wks)

    ' Loop through all the data on the sheet and add it to the
    ' recordset in the database.
    For X = 0 To LastCell.row - 2
        Rs.AddNew
        For c = 0 To LastColumn - 1
            Rs.fields(c) = StartCell.Offset(X, c).value

        Next c
        Rs.Update
    Next X

    Application.ScreenUpdating = True

    Exit Sub

ErrorHandler:
  Select Case Err
    Case 3204    ' Database already exists.
       Message = "There has been an error creating the database." & _
            Chr(10) & _
            Chr(10) & "Error Number: " & Err & _
            Chr(10) & "Error Description: " & Error() & _
            Chr(10) & _
            Chr(10) & "Would you like to delete the existing" & _
            "database:" & Chr(10) & _
            Chr(10) & _
            Left(ActiveWorkbook.Name, Len(ActiveWorkbook.Name) - 4) & _
            ".mdb"
        Title = "Error in Database Creation"
        Response = MsgBox(Message, vbYesNo, Title)
        If Response = vbYes Then
            Kill _
              Left(ActiveWorkbook.Name, Len(ActiveWorkbook.Name) - 4) _
              & ".mdb"
            Message = ""
            Title = ""
            Resume
        Else
            Message = "In order to run this procedure you need" & _
                Chr(10) & "to do ONE of the following:" & _
```

```
                Chr(10) & _
                Chr(10) & "1.  Move the existing database to a " & _
                "different directory, or " & _
                Chr(10) & "2.  Rename the existing database, or" & _
                Chr(10) & "3.  Move the workbook to a different " & _
                "directory, or" & _
                Chr(10) & "4.  Rename the workbook"
            Title = "Perform ONE of the following:"
            MsgBox Message, , Title
            Message = ""
            Title = ""
            Exit Sub
        End If

    ' Check to see if the error was Type Mismatch. If so, set the
    ' file to dbMemo.
    Case 13 ' Type mismatch.
        If CreateFieldFlag = 1 Then
            Set Fd = td.CreateField(ActiveCell.value, dbMemo)
            Fd.AllowZeroLength = True
            flag = 1
            r = LastCell.row - 1
            CreateFieldFlag = 0
            Resume Next
        Else
            Message = _
                            "Worksheet Name   :  " & Wks _
                & Chr(10) _
                & Chr(10) & "Error Number     :  " & Err _
                & Chr(10) & "Error Description:  " & Error() _
                & Chr(10) & Chr(10) & "Worksheet cannot be exported!"

            Title = "Type Mismatch"
            MsgBox Message, , Title
            Message = ""
            Title = ""
        End If

    ' For any other error, display the error.
    Case Else
       Message = _
                        "Worksheet Name   :  " & Wks _
            & Chr(10) _
            & Chr(10) & "Error Number     :  " & Err _
            & Chr(10) & "Error Description:  " & Error() _
            & Chr(10) & Chr(10) & "Worksheet cannot be exported!"

        Title = "An error has occured"
        MsgBox Message, , Title
        Message = ""
        Title = ""
  End Select
End Sub
```

A-2-4 MImportAccessToWks

```
Option Explicit

Private db As Database

Sub Import(strADBM As String)
'
'Purpose:   imports an access database into an excel workbook and builds a jmap
'Arguments: string containing the database to import

    On Error GoTo Handler

    'Get default Workspace.
    Dim wrkdefault As Workspace
    Set wrkdefault = DBEngine.Workspaces(0)
```

```
'Open database
Dim db As Database

If frmImportAccessToWks.BrowseClicked = True Then
    Set db = wrkdefault.OpenDatabase(strADBM)
Else
    Set db = wrkdefault.OpenDatabase(ActiveWorkbook.Path & "\" & strADBM)
End If


Dim colTables As Collection
Set colTables = frmImportAccessToWks.SelectedTables

'For every selected table, import table information
Dim Tb
Dim Rs As Recordset
Dim I, J As Integer
Dim RsSql As String
Dim newsheet, shtName As String
Dim fldName As Field
Dim fldValue As String
Dim strColumnWdLen As Integer

Dim intCount, numRow As Integer

For Each Tb In colTables

'new sheet name
    newsheet = Tb


    On Error Resume Next
    Sheets(newsheet).Select
    On Error Resume Next

    'add new sheet
    Sheets.Add
    ActiveSheet.Name = newsheet
    ActiveWindow.Zoom = 75

    RsSql = "SELECT * FROM [" & Tb & "]"
    Set Rs = db.OpenRecordset(RsSql, dbOpenDynaset)

    'Loop through the Microsoft Access field names and insert into
    ' the set of fields

    For I = 0 To Rs.fields.Count - 1

    intCount = 1
   'strColumnWdLen = 1
    Cells(intCount, I + 1) = Rs.fields(I).Name
    Cells(intCount, I + 1).Interior.ColorIndex = 8
    Cells(intCount, I + 1).Font.Bold = True
    Cells(intCount, I + 1).Borders.LineStyle = xlDouble
    strColumnWdLen = 16
    Worksheets(newsheet).Columns(I + 1).ColumnWidth = strColumnWdLen

    Do Until Rs.EOF
        Set fldName = Rs.fields(I)
        fldValue = fldName.value

        intCount = intCount + 1

        Cells(intCount, I + 1) = fldValue

        If (strColumnWdLen < Len(fldValue)) Then
         strColumnWdLen = Len(fldValue)
         Worksheets(newsheet).Columns(I + 1).ColumnWidth = strColumnWdLen
        End If
```

```
            Rs.MoveNext
            Loop

            Rs.MoveFirst

        Next I

    Next Tb

    db.Close

    MsgBox "Data Imported from " & strADBM & " to " & ActiveWorkbook.Name,
vbInformation, "Information"

    Exit Sub

Handler:
    Dim Message As String
    Dim Title As String
    Message = _
                    "Error Number      : " & Err _
        & Chr(10) & "Error Description: " & Error()

    Title = "An error has occured"
    MsgBox Message, , Title
    Message = ""
    Title = ""

End Sub
```

A-2-5 MRestorejMap

```
Dim rstSheetName, cItemSheetName, cTupleSheetName As String

Sub MapTable()
'
' CreateTables Macro
' Macro recorded 07/12/2001 by Minghui Han

  Dim rstSheetNameTemp, cItemSheetNameTemp, cTupleSheetNameTemp, sTempName As
String
  Dim nCount, nSheetCount As Integer
  Dim nEndStep As Integer

  rstSheetNameTemp = "{jMapRestore}"
  cItemSheetNameTemp = "{cItems}"
  cTupleSheetNameTemp = "{cTuples}"

  nCount = 1

  rstSheetName = rstSheetNameTemp
  cItemSheetName = cItemSheetNameTemp
  cTupleSheetName = cTupleSheetNameTemp

  Call MakejMap
  nEndStep = Sheets.Count

  For nSheetCount = nEndStep To 1 Step -1

    sTempName = cItemSheetNameTemp + Format(nCount)

    If Sheets(nSheetCount).Name = sTempName Then
        rstSheetName = rstSheetNameTemp + Format(nCount)
        cItemSheetName = sTempName
        cTupleSheetName = cTupleSheetNameTemp + Format(nCount)
        nCount = nCount + 1

        Call MakejMap
        nSheetCount = nSheetCount + 1
```

```
    End If

Next nSheetCount

End Sub
Sub MakejMap()

  Dim newsheet As Sheets
  Dim nHorPos, nVerPos, nColumnStart, nColumnEnd, nRowStart, nRowEnd, nToRow As
Integer
  Dim numItems, fndlen As Integer
  Dim strCellValue, searchString, subString, searchChar As String


  On Error Resume Next
  Sheets(rstSheetName).Select
  On Error Resume Next

  'add new sheet
  Sheets.Add
  ActiveSheet.Name = rstSheetName
  ActiveWindow.Zoom = 75

  Sheets(cTupleSheetName).Select

  'Get the Activesheet's range
  Set rgnSheet = ActiveSheet.UsedRange

  nRowEnd = rgnSheet.Rows.Count
  nColumnEnd = rgnSheet.Columns.Count
  nToRow = nRowEnd

  'find the started column for "Roles" data item
  For nRowStart = 1 To nRowEnd
        For nColumnStart = 1 To nColumnEnd

            strCellValue = Cells(nRowStart, nColumnStart)

            If InStr(1, strCellValue, "Roles", vbTextCompare) = 1 Then

                nHorPos = nRowStart         'find first row position of text with
"Roles"
                nVerPos = nColumnStart      'find column position of text with
"Roles"
                GoTo getValue

            End If
        Next nColumnStart
    Next nRowStart

getValue:

  For nRowStart = nHorPos + 2 To nRowEnd

            searchString = Cells(nRowStart, nVerPos)


            fndlen = InStr(1, searchString, ",")

            numItems = 1

            Do Until fndlen = 0

              subString = Left(searchString, fndlen - 1)

              searchString = Mid(searchString, fndlen + 1) ' Returns rest string

              fndlen = InStr(1, searchString, ",")
```

```
            Sheets(rstSheetName).Select

            subString = Trim(subString)
            Cells(numItems, nRowStart - 2).value = subString

            numItems = numItems + 1

            Sheets(cTupleSheetName).Select

          Loop

          'put last char to the new sheet
          Sheets(rstSheetName).Select
          subString = Trim(searchString)
          Cells(numItems, nRowStart - 2).value = subString
          Sheets(cTupleSheetName).Select

  Next nRowStart

  Sheets(rstSheetName).Select
  Set rgnSheet = ActiveSheet.UsedRange

  nRowEnd = rgnSheet.Rows.Count
  nColumnEnd = rgnSheet.Columns.Count

   'replace "?" with empty
   For nRowStart = 1 To nRowEnd

       For nColumnStart = 1 To nColumnEnd

           strCellValue = Cells(nRowStart, nColumnStart)

           If (strCellValue = "?") Then

               Cells(nRowStart, nColumnStart).value = ""

           End If

       Next nColumnStart

   Next nRowStart


   ...........

  Sheets(cItemSheetName).Select

  'Get the Activesheet's range
  Set rgnSheet = ActiveSheet.UsedRange

  nRowEnd = rgnSheet.Rows.Count
  nColumnEnd = rgnSheet.Columns.Count

  'find the started column for "Roles" data item
  For nRowStart = 1 To nRowEnd
        For nColumnStart = 1 To nColumnEnd

            strCellValue = Cells(nRowStart, nColumnStart)

            If InStr(1, strCellValue, "DataItem", vbTextCompare) = 1 Then

                nHorPos = nRowStart          'find first row position of text with
"Roles"
                nVerPos = nColumnStart       'find column position of text with
"Roles"
                GoTo getValueItem

            End If
        Next nColumnStart
    Next nRowStart
```

```
getValueItem:

  numItems = 1

  For nRowStart = nHorPos + 1 To nRowEnd

            searchString = Cells(nRowStart, nVerPos)


            Sheets(rstSheetName).Select

            searchString = Trim(searchString)
            Cells(numItems, nToRow + 1).value = searchString

            numItems = numItems + 1

            Sheets(cItemSheetName).Select


    Next nRowStart


    Sheets(rstSheetName).Select

    'add column numbers for each row
     For nRowStart = 1 To nRowEnd - 1
         numItems = 0
         For nColumnStart = 1 To nToRow - 1

                 If Not Cells(nRowStart, nColumnStart) = "" Then
                    numItems = numItems + 1
                 End If
         Next nColumnStart

         Cells(nRowStart, nToRow - 1) = numItems

     Next nRowStart

     numItems = 0
     'find the started column for data item
     For nRowStart = nRowEnd - 1 To 1 Step -1

         numItems = numItems + 1

         strCellValue = Cells(nRowStart, nToRow + 1)

         If InStr(1, strCellValue, "(", vbTextCompare) = 1 Then

             Cells(nRowStart, nToRow) = numItems - 1

             Range(Cells(nRowStart, 1), Cells(nRowStart, nToRow + 1)).Select
             Selection.Font.Bold = True

             numItems = 0

         End If

     Next nRowStart

     'fomat column width
     Range(Cells(1, 1), Cells(nRowEnd - 1, nToRow)).Select
     Selection.ColumnWidth = 2

     With Selection
         .HorizontalAlignment = xlCenter
         .VerticalAlignment = xlBottom
         .WrapText = False
         .Orientation = 0
         .AddIndent = False
         .ShrinkToFit = False
```

```
        .MergeCells = False

    End With

    Range(Cells(1, nToRow + 1), Cells(nRowEnd - 1, nToRow + 1)).Select
    Selection.ColumnWidth = 20
    MColor.ColorItem

    Cells(1, 1).Select


End Sub
```

A-2-6 MShellExecute

```
'
'Purpose:   this module is needed to display a html page in the default web
'           browser

Option Explicit

Private Declare Function ShellExecute Lib "shell32.dll" Alias _
        "ShellExecuteA" (ByVal hwnd As Long, ByVal lpszOp As _
        String, ByVal lpszFile As String, ByVal lpszParams As String, _
        ByVal lpszDir As String, ByVal FsShowCmd As Long) As Long

Private Declare Function GetDesktopWindow Lib "user32" () As Long

Private Const SW_SHOWNORMAL = 1
Private Const SW_SHOWMAXIMIZED = 3

Private Const SE_ERR_FNF = 2&
Private Const SE_ERR_PNF = 3&
Private Const SE_ERR_ACCESSDENIED = 5&
Private Const SE_ERR_OOM = 8&
Private Const SE_ERR_DLLNOTFOUND = 32&
Private Const SE_ERR_SHARE = 26&
Private Const SE_ERR_ASSOCINCOMPLETE = 27&
Private Const SE_ERR_DDETIMEOUT = 28&
Private Const SE_ERR_DDEFAIL = 29&
Private Const SE_ERR_DDEBUSY = 30&
Private Const SE_ERR_NOASSOC = 31&
Private Const ERROR_BAD_FORMAT = 11&

Sub ShellExec(DocName As String)
    Dim r As Long, Msg As String
    Dim Scr_hDC As Long

    Scr_hDC = GetDesktopWindow()

    r = ShellExecute(Scr_hDC, "Open", DocName, "", "C:\", SW_SHOWNORMAL)

    If r <= 32 Then
        'There was an error
        Select Case r
            Case SE_ERR_FNF
                Msg = "File not found"
            Case SE_ERR_PNF
                Msg = "Path not found"
            Case SE_ERR_ACCESSDENIED
                Msg = "Access denied"
            Case SE_ERR_OOM
                Msg = "Out of memory"
            Case SE_ERR_DLLNOTFOUND
                Msg = "DLL not found"
            Case SE_ERR_SHARE
                Msg = "A sharing violation occurred"
            Case SE_ERR_ASSOCINCOMPLETE
                Msg = "Incomplete or invalid file association"
            Case SE_ERR_DDETIMEOUT
```

```
            Msg = "DDE Time out"
        Case SE_ERR_DDEFAIL
            Msg = "DDE transaction failed"
        Case SE_ERR_DDEBUSY
            Msg = "DDE busy"
        Case SE_ERR_NOASSOC
            Msg = "No association for file extension"
        Case ERROR_BAD_FORMAT
            Msg = "Invalid EXE file or error in EXE image"
        Case Else
            Msg = "Unknown error"
    End Select

    MsgBox Msg, vbInformation

  End If

End Sub
```

A-2-7 MStartup

```
Option Explicit

'====================================================================
'Module Level Constant Declaration Section
'====================================================================

Private Const MACRO_MENU_CAPTION As String = "ADM&jMap" ' added by han

Public bjMaptoAccess As Boolean
Public bAccesstojMap As Boolean


Sub RemovejMapMacroMenu()
  Dim cbct As CommandBarControl
  On Error Resume Next

  For Each cbct In CommandBars.ActiveMenuBar.Controls
    If 0 = StrComp(cbct.Caption, MACRO_MENU_CAPTION, vbBinaryCompare) Then
      Call cbct.Delete
    End If
  Next cbct
End Sub

Public Sub AddjMapMacroMenu()
'
' MStartup Macro
' Macro recorded 3/23/2001 by Minghui Han
'
' Keyboard Shortcut: Ctrl+b
'


  Dim cbpopTopMenu As CommandBarPopup
  Dim cbpopSubMenu As CommandBarPopup
  Dim cbctls As CommandBarControls

  On Error Resume Next

  ' Ensure we have no duplicates
  Call RemovejMapMacroMenu

  bjMaptoAccess = False
  bAccesstojMap = False

  Set cbpopTopMenu = CommandBars.ActiveMenuBar.Controls.Add(type:=msoControlPopup)
  With cbpopTopMenu
    .Caption = MACRO_MENU_CAPTION
```

```
    .OnAction = "mnujMap_OnAction"
    .Visible = True
  End With

  Set cbctls = cbpopTopMenu.Controls

  ' Add the sub items to the menu
  With cbctls.Add(msoControlButton)
       .Caption = "&Create Tables"
       .OnAction = "mnuCreateTables_OnAction"
       .FaceId = 240
  End With

  ' Add the sub items to the menu
  With cbctls.Add(msoControlButton)
       .Caption = "&Remove Tables"
       .OnAction = "mnuRemoveTables_OnAction"
       .FaceId = 2002
  End With

  ' Add the sub items to the menu
  With cbctls.Add(msoControlButton)
       .Caption = "&jMap->DB"
       .OnAction = "mnujMaptoAccess_OnAction"
       .FaceId = 2116
  End With

  ' Add the sub items to the menu
  With cbctls.Add(msoControlButton)
       .Caption = "&DB->jMap"
       .OnAction = "mnuAccesstojMap_OnAction"
       .FaceId = 2109
  End With

    ' Add the sub items to the menu
  With cbctls.Add(msoControlButton)
       .Caption = "DB jMap &Analysis"
       .OnAction = "mnuAnalysisAccesstojMap_OnAction"
       .FaceId = 2114
  End With

  With cbctls.Add(msoControlButton)
         .Caption = "&Help..."
         .OnAction = "mnuHelp_OnAction"
         .BeginGroup = True
         .FaceId = 49
  End With


End Sub

Private Sub mnujMap_OnAction()
  On Error Resume Next

End Sub

Private Sub mnujMaptoAccess_OnAction()

    On Error Resume Next

    bjMaptoAccess = True
    frmBookSheetInfo.Show
    bjMaptoAccess = False


End Sub

Private Sub mnuAccesstojMap_OnAction()

    On Error Resume Next
```

```
    bAccesstojMap = True
    frmImportAccessToWks.Show
    bAccesstojMap = False

End Sub
Private Sub mnuAnalysisAccesstojMap_OnAction()

    On Error Resume Next

    bAccesstojMap = False
    frmImportAccessToWks.Show

End Sub

Private Sub mnuHelp_OnAction()

    On Error Resume Next

    ShellExec ActiveWorkbook.Path & "\" & "Help.htm"

End Sub

Private Sub mnuCreateTables_OnAction()

    On Error Resume Next

    frmBookSheetInfo.Show

End Sub

Private Sub mnuRemoveTables_OnAction()

    On Error Resume Next

    MTables.RemoveTables

End Sub
```

A-2-8 MTables

```
Sub CreateID(ByVal strSheetID As String, ByVal strBookID As String)
'
' CreateTables Macro
' Macro recorded 07/03/2001 by Minghui Han
'
' Keyboard Shortcut: Ctrl+t
'
    Dim nHorPos, nVerPos, nColumnStart, nColumnEnd, nRowStart, nRowEnd As Integer
    Dim numSetID, numMemberID, numItems, numTables, numField As Integer '
    Dim oldsheet, newsheet, strCellValue As String

    nHorPos = 1
    nVerPos = 1
    numSetID = 0
    numMemberID = 0

    oldsheet = ActiveSheet.Name
    newsheet = "<" + oldsheet + ">"

   'copy oldsheet contents to the new sheet and renamed as newsheet name
    Sheets(oldsheet).Copy before:=Sheets(oldsheet)
    ActiveSheet.Name = newsheet

    'Get the Activesheet's range
    Set rgnSheet = ActiveSheet.UsedRange

    nRowEnd = rgnSheet.Rows.Count
    nColumnEnd = rgnSheet.Columns.Count
```

```
    'replace empty cell with "?"
    For nRowStart = 1 To nRowEnd
        For nColumnStart = 1 To nColumnEnd

            strCellValue = Cells(nRowStart, nColumnStart)

            If ((strCellValue = "") And (nColumnStart < nColumnEnd - 2)) Then

                Cells(nRowStart, nColumnStart).value = "?"

            End If

        Next nColumnStart

    Next nRowStart

    'find the started column for data item
    For nRowStart = 1 To nRowEnd
        For nColumnStart = 1 To nColumnEnd

            strCellValue = Cells(nRowStart, nColumnStart)

            If InStr(1, strCellValue, "(", vbTextCompare) = 1 Then

                nHorPos = nRowStart          'find first row position of "("
                nVerPos = nColumnStart       'find column position of "("
                GoTo setID

            End If
        Next nColumnStart
    Next nRowStart

setID:
    For nRowStart = 1 To nRowEnd

            strCellValue = Cells(nRowStart, nVerPos)

            If InStr(1, strCellValue, "(", vbTextCompare) = 1 Then
              numSetID = numSetID + 1
              numMemberID = 0
              Cells(nRowStart, nVerPos + 5).value = "N"

            Else
              Cells(nRowStart, nVerPos + 5).value = "M"
            End If

            Cells(nRowStart, nVerPos + 1).value = numSetID
            Cells(nRowStart, nVerPos + 2).value = numMemberID
            Cells(nRowStart, nVerPos + 3).value = strSheetID
            Cells(nRowStart, nVerPos + 4).value = strBookID

            numMemberID = numMemberID + 1

    Next nRowStart

    'color and format cell's property

    'for SetID column
    Range(Cells(1, nVerPos + 1), Cells(nRowEnd, nVerPos + 1)).Select
    Selection.Font.ColorIndex = 3
    Selection.Font.Bold = True
    With Selection
        .HorizontalAlignment = xlCenter
        .VerticalAlignment = xlBottom
        .WrapText = False
        .Orientation = 0
        .AddIndent = False
        .ShrinkToFit = False
        .MergeCells = False
    End With
```

```
    'for MemeberID column
    Range(Cells(1, nVerPos + 2), Cells(nRowEnd, nVerPos + 2)).Select
    Selection.Font.ColorIndex = 9
    Selection.Font.Bold = True
    With Selection
        .HorizontalAlignment = xlCenter
        .VerticalAlignment = xlBottom
        .WrapText = False
        .Orientation = 0
        .AddIndent = False
        .ShrinkToFit = False
        .MergeCells = False
    End With

    'for SheetID column
    Range(Cells(1, nVerPos + 3), Cells(nRowEnd, nVerPos + 3)).Select
    Selection.Font.ColorIndex = 52
    Selection.Font.Bold = True
    With Selection
        .HorizontalAlignment = xlCenter
        .VerticalAlignment = xlBottom
        .WrapText = False
        .Orientation = 0
        .AddIndent = False
        .ShrinkToFit = False
        .MergeCells = False
    End With

    'for WoorkBookID column
    Range(Cells(1, nVerPos + 4), Cells(nRowEnd, nVerPos + 4)).Select
    Selection.Font.ColorIndex = 7
    Selection.Font.Bold = True
    With Selection
        .HorizontalAlignment = xlCenter
        .VerticalAlignment = xlBottom
        .WrapText = False
        .Orientation = 0
        .AddIndent = False
        .ShrinkToFit = False
        .MergeCells = False
    End With

    'for WoorkBookID column
    Range(Cells(1, nVerPos + 5), Cells(nRowEnd, nVerPos + 5)).Select
    Selection.Font.ColorIndex = 10
    Selection.Font.Bold = True
    Selection.ColumnWidth = 10

    With Selection
        .HorizontalAlignment = xlCenter
        .VerticalAlignment = xlBottom
        .WrapText = False
        .Orientation = 0
        .AddIndent = False
        .ShrinkToFit = False
        .MergeCells = False

    End With

End Sub
Sub CreateTables()
'
' CreateTables Macro
' Macro recorded 03/22/2001 by Minghui Han
'
' Keyboard Shortcut: Ctrl+t
'
Dim nHorPos, nVerPos, nColumnStart, nRowStart As Integer
Dim numItems, numTables, numField As Integer '
Dim oldsheet, newsheet, strCellValue As String
Dim strField1(1 To 500) As String
```

```
Dim strField2(1 To 500) As String
Dim strField3(1 To 500) As String
Dim strField4(1 To 500) As String
Dim strField5(1 To 500) As String
Dim strField6(1 To 500) As String
Dim strColumnWdLen1 As Integer
Dim strColumnWdLen2 As Integer
Dim strColumnWdLen3 As Integer
Dim strColumnWdLen4 As Integer
Dim strColumnWdLen5 As Integer

'Get the Activesheet's range
Set rgnSheet = ActiveSheet.UsedRange

nHorPos = 1
nVerPos = 1
oldsheet = ActiveSheet.Name

For nRowStart = 1 To rgnSheet.Rows.Count
    For nColumnStart = 1 To rgnSheet.Columns.Count

        strCellValue = Cells(nRowStart, nColumnStart)

        If InStr(1, strCellValue, "{", vbTextCompare) = 1 Then
                nHorPos = nRowStart         'find first row position of "{"
                nVerPos = nColumnStart      'find column position of "{"
                strField1(1) = strCellValue
                GoTo firstTable
        End If
    Next nColumnStart
Next nRowStart

firstTable:
    Sheets(oldsheet).Select

    strField1(1) = "DataItem"
    strField2(1) = "SetID"
    strField3(1) = "MemberID"
    strField4(1) = "SheetID"
    strField5(1) = "WorkbookID"

    numItems = 2   'start to get second row's value, and so on

    For nRowStart = nHorPos To rgnSheet.Rows.Count ' - 1

        strField1(numItems) = Cells(nRowStart, nVerPos)
        strField2(numItems) = Cells(nRowStart, nVerPos + 1)
        strField3(numItems) = Cells(nRowStart, nVerPos + 2)
        strField4(numItems) = Cells(nRowStart, nVerPos + 3)
        strField5(numItems) = Cells(nRowStart, nVerPos + 4)

        numItems = numItems + 1

    Next nRowStart

    'ready to add new sheet

    'new sheet name
    newsheet = "{cItems}"
    On Error Resume Next
    Sheets(newsheet).Select
    On Error Resume Next

    'add new sheet
    Sheets.Add
    ActiveSheet.Name = newsheet
    Columns("A:E").ColumnWidth = 15
    strColumnWdLen1 = 1

    ActiveSheet.Name = newsheet
    ActiveWindow.Zoom = 75
```

```
  For nRowStart = 1 To numItems - 1
      Cells(nRowStart, 1) = strField1(nRowStart)
      Cells(nRowStart, 2) = strField2(nRowStart)
      Cells(nRowStart, 3) = strField3(nRowStart)
      Cells(nRowStart, 4) = strField4(nRowStart)
      Cells(nRowStart, 5) = strField5(nRowStart)

      If (strColunmWdLen1 < Len(strField1(nRowStart))) Then
          strColunmWdLen1 = Len(strField1(nRowStart))
          Columns("A:A").ColumnWidth = strColunmWdLen1
      End If

      If nRowStart = 1 Then

           Cells(nRowStart, 1).Interior.ColorIndex = 8
           Cells(nRowStart, 2).Interior.ColorIndex = 8
           Cells(nRowStart, 3).Interior.ColorIndex = 8
           Cells(nRowStart, 4).Interior.ColorIndex = 8
           Cells(nRowStart, 5).Interior.ColorIndex = 8
           Cells(nRowStart, 1).Borders.LineStyle = xlDouble
           Cells(nRowStart, 2).Borders.LineStyle = xlDouble
           Cells(nRowStart, 3).Borders.LineStyle = xlDouble
           Cells(nRowStart, 4).Borders.LineStyle = xlDouble
           Cells(nRowStart, 5).Borders.LineStyle = xlDouble

      Else
           Cells(nRowStart, 1).Font.ColorIndex = 32
           Cells(nRowStart, 2).Font.ColorIndex = 3
           Cells(nRowStart, 3).Font.ColorIndex = 3
           Cells(nRowStart, 4).Font.ColorIndex = 3
           Cells(nRowStart, 5).Font.ColorIndex = 3

           Cells(nRowStart, 1).Interior.ColorIndex = 2
           Cells(nRowStart, 2).Interior.Color = RGB(255, 204, 153)
           Cells(nRowStart, 3).Interior.Color = RGB(204, 255, 204)
           Cells(nRowStart, 4).Interior.Color = RGB(200, 204, 253)
           Cells(nRowStart, 5).Interior.Color = RGB(100, 250, 200)

           Cells(nRowStart, 1).Borders.LineStyle = xlDot
           Cells(nRowStart, 2).Borders.LineStyle = xlDot
           Cells(nRowStart, 3).Borders.LineStyle = xlDot
           Cells(nRowStart, 4).Borders.LineStyle = xlDot
           Cells(nRowStart, 5).Borders.LineStyle = xlDot

      End If

  Next nRowStart

  Range("A1:E1").Select
  Selection.Font.Bold = True
  Cells(1, 1).Select

otherTable:

    strColunmWdLen1 = 1
    strColunmWdLen2 = 1
    strColunmWdLen3 = 1
    strColunmWdLen4 = 1
    strColunmWdLen5 = 1

   'go back to old sheet ready to read the each column's value
    Sheets(oldsheet).Select

    strField1(1) = "CtupleID"
    strField2(1) = "Roles"
    strField3(1) = "SetID"
    strField4(1) = "MemberID"
    strField5(1) = "SheetID"
    strField6(1) = "WorkbookID"
```

```
'new sheet name
newsheet = "{cTuples}"
On Error Resume Next
Sheets(newsheet).Select
On Error Resume Next

'add new sheet
Sheets.Add
ActiveSheet.Name = newsheet
Columns("A:F").ColumnWidth = 10

ActiveSheet.Name = newsheet
ActiveWindow.Zoom = 75

'add field name in the first row to the new sheet, and format them
For numField = 1 To 6

    Cells(1, numField).Interior.ColorIndex = 8
    Cells(1, numField).Borders.LineStyle = xlDouble

    If numField = 1 Then
        Cells(1, numField) = strField1(1)
    ElseIf numField = 2 Then
        Cells(1, numField) = strField2(1)
    ElseIf numField = 3 Then
        Cells(1, numField) = strField3(1)
    ElseIf numField = 4 Then
        Cells(1, numField) = strField4(1)
    ElseIf numField = 5 Then
        Cells(1, numField) = strField5(1)
    ElseIf numField = 6 Then
        Cells(1, numField) = strField6(1)
    End If

Next numField

Sheets(oldsheet).Select

'For all other columns tables
For nColumnStart = 0 To nVerPos - 3

    numItems = 2   'start to get second row's value, and so on
    strField2(numItems) = Cells(1, nColumnStart)
    strField3(numItems) = Cells(1, nVerPos + 1)
    strField4(numItems) = Cells(1, nVerPos + 2)
    strField5(numItems) = Cells(1, nVerPos + 3)
    strField6(numItems) = Cells(1, nVerPos + 4)

    numItems = numItems + 1

    If nColumnStart = 0 Then

        strField2(numItems - 1) = Cells(1, nVerPos + 5)
        'to get last colume's value i.e. N, M...
        For nRowStart = 2 To rgnSheet.Rows.Count

            If Not Cells(nRowStart, nVerPos + 4) = "" Then
                strField2(numItems) = CStr(strField2(numItems - 1)) + ", " +
CStr(Cells(nRowStart, nVerPos + 5))
                strField3(numItems) = CStr(strField3(numItems - 1)) + ", " +
CStr(Cells(nRowStart, nVerPos + 1))
                strField4(numItems) = CStr(strField4(numItems - 1)) + ", " +
CStr(Cells(nRowStart, nVerPos + 2))
                strField5(numItems) = CStr(strField5(numItems - 1)) + ", " +
CStr(Cells(nRowStart, nVerPos + 3))
                strField6(numItems) = CStr(strField6(numItems - 1)) + ", " +
CStr(Cells(nRowStart, nVerPos + 4))

                numItems = numItems + 1
            End If
```

```
            Next nRowStart

        Else

            'to retrive the jMap notation value
            'since the sheet has one more row for title, minus one to get real
rows
            For nRowStart = 2 To rgnSheet.Rows.Count

                If Not Cells(nRowStart, nColumnStart) = "" Then
                    strField2(numItems) = CStr(strField2(numItems - 1)) + ", " +
CStr(Cells(nRowStart, nColumnStart))
                    strField3(numItems) = CStr(strField3(numItems - 1)) + ", " +
CStr(Cells(nRowStart, nVerPos + 1))
                    strField4(numItems) = CStr(strField4(numItems - 1)) + ", " +
CStr(Cells(nRowStart, nVerPos + 2))
                    strField5(numItems) = CStr(strField5(numItems - 1)) + ", " +
CStr(Cells(nRowStart, nVerPos + 3))
                    strField6(numItems) = CStr(strField6(numItems - 1)) + ", " +
CStr(Cells(nRowStart, nVerPos + 4))

                    numItems = numItems + 1
                End If

            Next nRowStart
        End If



       'ready to add new sheet
        On Error Resume Next
        Sheets(newsheet).Select

        'assign the value to new sheet
        Cells(nColumnStart + 2, 1) = CStr(nColumnStart)
        Cells(nColumnStart + 2, 2) = strField2(numItems - 1)
        Cells(nColumnStart + 2, 3) = strField3(numItems - 1)
        Cells(nColumnStart + 2, 4) = strField4(numItems - 1)
        Cells(nColumnStart + 2, 5) = strField5(numItems - 1)
        Cells(nColumnStart + 2, 6) = strField6(numItems - 1)


        'add field data in the new sheet, and format them
        Cells(nColumnStart + 2, 1).Font.ColorIndex = 32
        Cells(nColumnStart + 2, 2).Font.ColorIndex = 3
        Cells(nColumnStart + 2, 3).Font.ColorIndex = 32
        Cells(nColumnStart + 2, 4).Font.ColorIndex = 3
        Cells(nColumnStart + 2, 5).Font.ColorIndex = 32
        Cells(nColumnStart + 2, 6).Font.ColorIndex = 3

        Cells(nColumnStart + 2, 1).Interior.ColorIndex = 2
        Cells(nColumnStart + 2, 2).Interior.Color = RGB(255, 255, 153)
        Cells(nColumnStart + 2, 3).Interior.Color = RGB(204, 255, 204)
        Cells(nColumnStart + 2, 4).Interior.Color = RGB(255, 204, 153)
        Cells(nColumnStart + 2, 5).Interior.Color = RGB(240, 200, 223)
        Cells(nColumnStart + 2, 6).Interior.Color = RGB(220, 250, 230)

        Cells(nColumnStart + 2, 1).Borders.LineStyle = xlDot
        Cells(nColumnStart + 2, 2).Borders.LineStyle = xlDot
        Cells(nColumnStart + 2, 3).Borders.LineStyle = xlDot
        Cells(nColumnStart + 2, 4).Borders.LineStyle = xlDot
        Cells(nColumnStart + 2, 5).Borders.LineStyle = xlDot
        Cells(nColumnStart + 2, 6).Borders.LineStyle = xlDot


        If (strColumnWdLen1 < Len(strField2(numItems - 1))) Then
           strColumnWdLen1 = Len(strField2(numItems - 1))
           Columns("B:B").ColumnWidth = strColumnWdLen1
        End If
```

```
        If (strColunmWdLen2 < Len(strField3(numItems - 1))) Then
            strColunmWdLen2 = Len(strField3(numItems - 1))
            Columns("C:C").ColumnWidth = strColunmWdLen2
        End If

        If (strColunmWdLen3 < Len(strField4(numItems - 1))) Then
            strColunmWdLen3 = Len(strField4(numItems - 1))
            Columns("D:D").ColumnWidth = strColunmWdLen3
        End If

        If (strColunmWdLen4 < Len(strField5(numItems - 1))) Then
            strColunmWdLen4 = Len(strField5(numItems - 1))
            Columns("E:E").ColumnWidth = strColunmWdLen4
        End If

        If (strColunmWdLen5 < Len(strField6(numItems - 1))) Then
            strColunmWdLen5 = Len(strField6(numItems - 1))
            Columns("F:F").ColumnWidth = strColunmWdLen5
        End If

        Range("A1:F1").Select
        Selection.Font.Bold = True
        Cells(1, 1).Select

        Sheets(oldsheet).Select

    Next nColumnStart

End Sub
Sub RemoveTables()
'
' RemoveTables Macro
' Macro recorded 03/22/2001 by Minghui Han
'
'
   Dim Wks As Worksheet

   For Each Wks In Worksheets
        If Wks.type = xlWorksheet Then
            If Wks.Visible Then
                If InStr(Wks.Name, "(") Or InStr(Wks.Name, "<") Then
                    Application.DisplayAlerts = False
                    Wks.Delete
                    Application.DisplayAlerts = True
                End If
            End If
        End If
    Next Wks


End Sub
```

A-3 Class Modules Source Code

The source code for Class Modules includes as following created class:

- **AccessJMapBuilder**
- **Table**

All source code in above Classes are listed as following:

A-3-1 AccessJMapBuilder

```
'Option Compare Database
Option Explicit

Private App As Excel.Application    'pointer to excel application
Private Book As Excel.Workbook      'pointer to excel workbook
Private Sheet As Excel.Worksheet    'pointer to excel worksheet

Private SetTable As Table  'pointer to table, holds the cordinates of the written
sets

Private Sub Class_Initialize()
'
'Purpose: create the workbook and sheet to work in
'BECAUSE: we cannot run build database on a new book

    'create application and workbook, make pointer point to created objects
    Set App = Workbooks.Application
    Set Sheet = App.Sheets.Add
    Sheet.Activate

    'inputting first entry in the spreadsheet
    'first entry is the (View) set
    'done explicitly because insert functions (later in class)
    'depend on this entry to determine where to insert new sets
    Sheet.Cells(1, 1) = 0  'for 0 associations
    Sheet.Cells(1, 2) = 0  'for 0 set members
    Sheet.Range("A1:B1").Select
    App.Selection.Font.ColorIndex = 3  'change color to red

    Sheet.Cells(1, 3) = "(View)"
    Sheet.Range("A1:C1").Select
    App.Selection.Font.Bold = True  'make bold
    App.Selection.ColumnWidth = 2  'column width

    'create the table and insert the first row that we just wrote into the
spreadsheet
    Set SetTable = New Table
    SetTable.InsertRow "View", 1, 3, "V"
End Sub


Private Sub Class_Terminate()
'
'Purpose:  Quit the application and free all allocated space
'Arguments: Filename, full path: where to save workbook

  'free allocated resources
    Set App = Nothing
    Set Book = Nothing
    Set Sheet = Nothing
    Set SetTable = Nothing
End Sub



Public Sub NameSheet(Name As String)
'
'Purpose: Assigns a name to the Active sheet

Dim CurPos As Integer
Dim StartPos As Integer

    CurPos = Len(Name)
    Do Until CurPos < 1
        StartPos = InStr(CurPos, Name, "\")
        If StartPos <> 0 Then
```

```
            Exit Do
        End If
        CurPos = CurPos - 1
    Loop

    Sheet.Name = Mid(Name, StartPos + 1)

    Sheet.Name = "{" + Sheet.Name + "}"

    ActiveSheet.Select
    ActiveWindow.Zoom = 75

End Sub



Public Sub Save(Filename As String)
'
'Purpose:  save the file and close the book
'Arguments: Filename, full path: where to save workbook

    Mid(Filename, Len(Filename) - 2) = "xls"
    Book.SaveAs Filename

End Sub



Public Sub InsertSet(set_name As String, association As String)
'
'Purpose:  Insert a new set into the Jmap
'Arguments: set_name, name of the set, not including curly brackets

    If SetTable.Exists(set_name) = True Then
        Exit Sub
    End If

    Dim row As Integer, Column As Integer
    NewCell row, Column      'get next available place

    'write the set_name in the sheet and make it bold
    Sheet.Cells(row, Column) = "{" & set_name & "}"
    Sheet.Cells(row, Column).Select
    App.Selection.Font.Bold = True

    'insert a value of 0 next to the set, meaning 0 set members
    'then make bold and red
    Sheet.Cells(row, Column - 1) = 0
    Sheet.Cells(row, Column - 1).Select
    App.Selection.Font.ColorIndex = 3
    App.Selection.Font.Bold = True

    'left of the number of set members, onsert another 0 meaning 0 associations
    'then again make bold and red
    Sheet.Cells(row, Column - 2) = 0
    Sheet.Cells(row, Column - 2).Select
    App.Selection.Font.ColorIndex = 3
    App.Selection.Font.Bold = True

    'update our table
    SetTable.InsertRow set_name, row, Column, association

End Sub



Public Sub InsertSetMember(set_name As String, ByVal value As String)
'
```

```
'Purpose:  Insert a new set member into the given set
'Arguments: set_name, name of the set, not including curly brackets
'           value, a string for the set member value

    'cannot insert into a non existant set
    If SetTable.Exists(set_name) = False Then
        Exit Sub
    End If

    'if set exists make sure this is not a duplicate entry
    If FindSetMember(set_name, value) = True Then
        Exit Sub
    End If


    Dim row As Integer, Column As Integer
    SetTable.GetNamedIndexes set_name, row, Column      'get coordinates of set

    'add a new row under the set name
    Sheet.Cells(row + 1, Column).Select
    App.Selection.EntireRow.Insert

    'insert the value into the new row, 1 column to the right of the set
    Sheet.Cells(row + 1, Column + 1) = value

    'insert value of 0 for the number of associations
    '2 columns left of where we just put the set member
    Sheet.Cells(row + 1, Column - 2) = 0
    Sheet.Cells(row + 1, Column - 2).Select
    App.Selection.Font.ColorIndex = 3
    App.Selection.Font.Bold = False


    'update the counter for set members next to the set (add 1). it's 1 column to
the left of the set
    Sheet.Cells(row, Column - 1) = Sheet.Cells(row, Column - 1) + 1

    'update our table (we added a new row at row + 1)
    SetTable.ShiftDown row + 1

End Sub



Public Function FindSetMember(set_name As String, ByVal value As String, Optional
ret_row As Integer, Optional ret_col As Integer) As Boolean
'
'Purpose:  Checks if a set member is already part of a set, if yes returns the
index
'Arguments: set_name, name of the set, not including curly brackets
'           value, a string for the set member value

    ' value on error
    FindSetMember = False

    'cannot search into a non existant set
    If SetTable.Exists(set_name) = False Then
        Exit Function
    End If

    'set exists, get coordinates of set
    Dim row As Integer, Column As Integer
    SetTable.GetNamedIndexes set_name, row, Column

    'if we find the set return it's coordinates else return -1
    Dim I As Integer
    Dim limit As Integer
    Dim set_member_name As String
    limit = Sheet.Cells(row, Column - 1).value
    I = 0
```

```
    For I = 0 To limit
        set_member_name = Sheet.Cells(row + I, Column + 1)
        If StrComp(set_member_name, value) = 0 Then
            ret_row = row + I
            ret_col = Column + 1
            FindSetMember = True
            Exit Function
        End If
    Next I
End Function



Public Sub AddColumn()
'
'Purpose: adds a column to the bigenning of the worksheet

    Sheet.Cells(1, 1).Select
    App.Selection.EntireColumn.Insert
    App.Selection.ColumnWidth = 2  'column width

    'populate the blank columns with the correct association headings

    'update the values in the table
    SetTable.ShiftRight
End Sub



Private Sub NewCell(ByRef row As Integer, ByRef Column As Integer)
'
'Purpose: find the next free space to insert a set
'Arguments: row, places the value of new row where to insert
'           column, places the value of the new column where to insert

'gets location of last set
SetTable.GetLastEntry row, Column

'add the number of set members for last set member
row = row + Sheet.Cells(row, Column - 1)

'one more for a new row
row = row + 1

End Sub



Public Sub DoRowGrouping(Name As String)
'
'Purpose: Groups the rows under a set
'Arguments: name is the name of the set to be grouped

    Dim row As Integer, row2 As Integer, Column As Integer

    SetTable.GetNamedIndexes Name, row, Column  'find coordinates of set
    row2 = row + Sheet.Cells(row, Column - 1)   'calculate the limit
    Sheet.Range(Sheet.Cells(row + 1, Column), Sheet.Cells(row2, Column)).Select
    App.Selection.Rows.Group  'group
End Sub



Public Sub DoGroupSettings()
'
'Purpose: Edits the settings for the row groupings
```

```
    Sheet.Outline.AutomaticStyles = False
    Sheet.Outline.SummaryRow = xlAbove
    Sheet.Outline.SummaryColumn = xlLeft
    Sheet.Outline.ShowLevels RowLevels:=1
End Sub


Public Sub InsertAssociation(Name As String, ByVal Member As String, RelationType
As String, Name2 As String, ByVal Member2 As String, RelationType2 As String, View
As String)
'
'Purpose: Adds an association between two set members

    'find the view part of it. exit if it dosen't exist
    Dim row As Integer, col As Integer
    If FindSetMember("View", View, row, col) = False Then
        Exit Sub
    End If

    'extend the (View) header of the map
    Dim row_h As Integer, col_h As Integer, a As String
    SetTable.GetNamedIndexes "View", row_h, col_h, a

    'if statement to avoid overwriting and over counting
    If Sheet.Cells(row_h, 1) = "" Then
        Sheet.Cells(row_h, 1) = a
        Sheet.Cells(row_h, 1).Select
        App.Selection.Font.Bold = True  'make bold
        Sheet.Cells(row_h, col_h - 2) = Sheet.Cells(row_h, col_h - 2) + 1
    End If


    'insert a "v" for the set member of the (View)
    If Sheet.Cells(row, 1) = "" Then
        Sheet.Cells(row, 1) = "v"
        Sheet.Cells(row, 1).Select
        App.Selection.Font.Bold = False
        'Sheet.Cells(row, 1).Interior.ColorIndex = 8

        Sheet.Cells(row, col - 3) = Sheet.Cells(row, col - 3) + 1

    End If

'---------------- First part of the association -----------------'

    'get the first part of the association and extend the header
    SetTable.GetNamedIndexes Name, row_h, col_h, a

    If Sheet.Cells(row_h, 1) = "" Then
        Sheet.Cells(row_h, 1) = a
        Sheet.Cells(row_h, 1).Select
        App.Selection.Font.Bold = True  'make bold
        Sheet.Cells(row_h, col_h - 2) = Sheet.Cells(row_h, col_h - 2) + 1
    End If

    'insert the actual association
    FindSetMember Name, Member, row, col

    'find the correct column to add the association to
    Dim I As Integer, col2 As Integer
    col2 = 1
    For I = 1 To col - 3
        If Sheet.Cells(row, I) <> "" And I <> col - 3 Then
            col2 = I
            Exit For
        End If
    Next I

    If Sheet.Cells(row, col2) = "" Then
        Sheet.Cells(row, col2) = RelationType
        Sheet.Cells(row, col2).Select
```

```
        App.Selection.Font.Bold = False
        Sheet.Cells(row, col - 3) = Sheet.Cells(row, col - 3) + 1
    End If

'---------------- Second part of the association ----------------'

    'get the second part of the association and extend it's header
    SetTable.GetNamedIndexes Name2, row_h, col_h, a

    If Sheet.Cells(row_h, 1) = "" Then
        Sheet.Cells(row_h, 1) = a
        Sheet.Cells(row_h, 1).Select
        App.Selection.Font.Bold = True  'make bold
        Sheet.Cells(row_h, col_h - 2) = Sheet.Cells(row_h, col_h - 2) + 1
    End If

    'insert the actual association
    FindSetMember Name2, Member2, row, col

    'use col2 from first part of association
    If Sheet.Cells(row, col2) = "" Then
        Sheet.Cells(row, col2) = RelationType2
        Sheet.Cells(row, col2).Select
        App.Selection.Font.Bold = False
        'increment the counter
        Sheet.Cells(row, col - 3) = Sheet.Cells(row, col - 3) + 1
    End If
End Sub
```

A-3-2 Table

```
'MADE the table object DYNAMIC

'Option Compare Database
Option Explicit

Private max As Integer      ' maximum number of entries in the table
Private next_free As Integer  'index of next available place in table

Private Names() As String   'choose not to use variant for effency reasons
Private association() As String  'determiones the type of association
Private Indexes() As Integer 'stored [(row,column),(row,column)...]


Private Sub Class_Initialize()
'
'Purpose: constructor type method used to initilize table.

    next_free = 0  'on creation index 0 is available
    max = 20

'NOTE: indexed from 0 to n, (c++ style)
    ReDim Preserve Names(max + 1) As String
    ReDim Preserve association(max + 1) As String
    ReDim Preserve Indexes(2, max + 1) As Integer
End Sub

'does nothing but if needed add destruction code here
Private Sub Class_Terminate()

End Sub
```

```
Public Sub InsertRow(Name As String, row As Integer, Column As Integer, a As
String)
'
'Purpose: Inserts a row into our table, a row contains the name of a set and it's
coordinates
'Arguments: name = name of set
'           row = row index of set location
'           column = column index of set location


    If next_free = max Then
        'double the size of the array
        max = max * 2

        'NOTE: indexed from 0 to n, (c++ style)
        ReDim Preserve Names(max + 1) As String
        ReDim Preserve association(max + 1) As String
        ReDim Preserve Indexes(2, max + 1) As Integer

    End If

    'insert the actual data
    Names(next_free) = Name
    association(next_free) = a
    Indexes(0, next_free) = row
    Indexes(1, next_free) = Column
    next_free = next_free + 1

End Sub


Public Function FindIndex(Name As String) As Integer
'
'Purpose: Gets the array index in the table of the given set name.
'Arguments: name = name of set
'Returns: an integer which is the index of the given set name or -1 if not found


    FindIndex = -1 'return -1 if not found
    Dim I As Integer
    For I = 0 To max
        If StrComp(Names(I), Name) = 0 Then
            FindIndex = I  'means set name was found reset return to current index
        End If
    Next I
End Function



Public Function Exists(Name As String) As Boolean
'
'Purpose: Checks if a set already exists
'Arguments: name = name of set
'Returns: BOOL true if found false if not

    Exists = False 'return false if not found
    Dim I As Integer
    For I = 0 To max
        If StrComp(Names(I), Name) = 0 Then
            Exists = True  'means set name was found reset return true
            Exit Function
        End If
    Next I
End Function
```

```
Public Function GetNamedIndexes(Name As String, row As Integer, Column As Integer,
Optional a As String)
'
'Purpose: Takes the name of a set and places it's coordinates into the row and
column arguments
'Arguments: name = name of set
'           row = argument to place found row value
'           column = argument to place found column value
'Returns: 0 on success -1 on failure


    Dim I As Integer
    I = FindIndex(Name)
    If I <> -1 Then
        row = Indexes(0, I)
        Column = Indexes(1, I)
        a = association(I)
        GetNamedIndexes = 0
    Else
        Debug.Print "set name not found"
        GetNamedIndexes = -1
    End If

End Function




Public Sub ShiftRight()
'
'Purpose: increment all the column values by one used when we insert a column into
the spreadsheet
'Arguments:
'Returns:

    Dim I As Integer
    For I = 0 To max
        Indexes(1, I) = Indexes(1, I) + 1
    Next I
End Sub



Public Sub ShiftDown(row_index As Integer)
'
'Purpose:  shift all values under the given row
'Arguments: row_index = the row in the spreadsheet where you insert a new row
'           this forces all the entries in the table to be wrong
'           so must call ShiftDown to correct entries in the table

    Dim I As Integer
    For I = 0 To max
        If Indexes(0, I) >= row_index Then
            Indexes(0, I) = Indexes(0, I) + 1
        End If
    Next I
End Sub



Public Sub GetLastEntry(ByRef row As Integer, ByRef Column As Integer)
'
'Purpose:  puts the coordinates of the last set into the arguments
'Arguments: row = place the row index here
'           column = place the column index here
'

'    Debug.Assert next_free <> 0
```

```
    If next_free > 0 Then
        row = Indexes(0, next_free - 1)
        Column = Indexes(1, next_free - 1)
    Else
        Debug.Print "the table is empty"
    End If
End Sub


Public Function GetAllEntries() As Collection
'
'Purpose:  Returns a collection with the names of all the entries in the table

    Dim I As Integer
    Set GetAllEntries = New Collection
    For I = 0 To max
        GetAllEntries.Add (Names(I))
    Next I
End Function
```