

https://dspace.org/

The use of expert systems in CAD

Gritzman, Jonathan Larry

1994
https://hdl.handle.net/10539/28625

Downloaded from a DSpace repository. https://dspace.org/

THE USE OF EXPERT SYSTEMS IN CAD

Jonathan Larry Gritzman

Degree of master of Science in Engineering by advanced course work and research:

A dissertation submitted to the faculty of Engineering, University of the Witwatersrand, Johannesburg, in fulfilment of the requirements for the degree of Master of Science in Engineering

20 May 1994

I declare that this dissertation is my own,unaided work. It is being submitted for the degree of Master of science in Engineering in the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination in any other University.

16 day of August 1994

ABSTRACT

Computer Aided Design, or CAD, is a powerful graphical tool used to produce detailed and complex drawings and to carry out engineering analysis. The operator interprets these drawings and attaches meaning to the entities within the CAD. Expert systems capture expertise within a knowledge base in a specialised area. By incorporating an expert system within the CAD system, forming an "intelligent CAD" system, the computer can manipulate the information within the CAD system in a meaningful way. It can become an on board co-assistant to the designer. The aim of the project report is to describe a software structure which enables a CAD system and an expert system to interact and share information.

The report discusses relevant issues which need to be taken into account. A Basic prototype has been created using object oriented programming techniques with the C++ programming language. Its main focus is to establish the functions of the various components and the type of information that needs to be exchanged between them. It also looks at ways of structuring the software into independent components which are reusable, easily maintainable and independent from one another.

The prototype has highlighted requirements and weaknesses of "intelligent CAD" systems. Data representation is found to be extremely important. It involves the representation of entities and the relationships between other entities. Messages are the means of communication between components and they enable components to be decoupled and independent from one another.

CONTENTS

ABSTRACT i3

CONTENTS i4

LIST OF FIGURES i7

CHAPTER 1 INTRODUCTION 1
- Background 1
- Scope of research 2
- Structure of the project report 2

CHAPTER 2 CAD SYSTEMS 3
- Introduction 3
- Features 4
- Data 5
- Limitations 5
- Conclusion 6

CHAPTER 3 EXPERT SYSTEMS 8
- Introduction 8
- Basic architecture 8
- Knowledge representation 9
- Rules 10
- Inference 11
- Search strategies 11
  - Forward chaining 12
  - Backward chaining 12
- Blackboard systems 13
- Frames 14
- Capabilities 14
- Limitations 15
- Conclusion 15

CHAPTER 4 INTELLIGENT CAD 17
- Introduction 17
- Requirements of Intelligent CAD 17
- Role of CAD 18
- Role of expert systems 19
- Capabilities of intelligent CAD 19
- Limitations of intelligent CAD 20
- Conclusion 20

CHAPTER 5 EXAMPLES OF INTELLIGENT CAD SYSTEMS 21
Introduction 21
Representation of knowledge and entities of the CAD system 21
Representation and search of data 22
Graphics Programming for knowledge guided interaction 23
The Tropic intelligent CAD system 25
Objectives of the Tropic system 26
Structure of Tropic 26
Knowledge of Tropic 28
System input 29
Searching the rules 30
Top down strategy 31
An Expert system in a pipe-work management system 31
Further work in artificial intelligence, graphics and databases 32
Conclusion 32

CHAPTER 6 A PROPOSAL FOR INTELLIGENT CAD 34
Introduction 34
Objectives of the prototype 34
System architecture 35
Importance of data representation 36
Knowledge representation 37
Object oriented paradigm 38
Message passing 39
Shape entities 40
Conclusion 41

CHAPTER 7 IMPLEMENTATION OF THE PROTOTYPE 42
Introduction 42
Shape class 43
Picture class 44
Knowledge base 46
Structure of the knowledge base 50
Inference engine 52
Structure of the shape controller class 54
Interaction between the graphical interface and the user 56
Interaction of shape entities 56
Interaction between the entities and the inference engine 57
Interaction between the entities and the rule base 57
Interaction Between the inference engine and the rule base 59
Interacting between entities and the rule base 60
Conclusion 60

CHAPTER 8 CONCLUSION 63
Importance of Intelligent CAD Systems 63
Lessons learned from the Prototype 63
Recommendations for future work 65
Conclusions 66

APPENDIX A CLASS STRUCTURE OF THE PROTOTYPE 68

APPENDIX B HEADER FILES OF THE PROTOTYPE 70

REFERENCES 84

BIBLIOGRAPHIES 87

LIST OF FIGURES

Figure

1. An Example of Rules in a Rule Base 9
2. Integrating the Interaction Handler, Knowledge base and DataBase 24
3. The Tropic System Design 27
4. Tropic Design Problem Solver 27
5. The Shape Objects within the CAD 45
6. The Picture Object 45
7. The Inheritance Hierarchy of the Knowledge Bases 47
8. Indexing Rules from the Knowledge Bases 49
9. The Control Structure for Determining the Outcome of a Rule 53
10. The Shape Controller Class 54
11. Interaction Between the Shape Controller Class, Picture Object and Inference Engine 58
12. Class Structure of the User Interface for the CAD Prototype 68

CHAPTER 1 INTRODUCTION

Background

In the next few years, South Africa plans to spend over 10 billion rand on providing electrical power to the townships. Over half the costs are cabling costs. The Department of Electrical Engineering at the University of the Witwatersrand, in conjunction with other interested parties, is involved in the Computer Aided Reticulation of Townships (CART) project. The project aims at constructing a Computer Aided Design system which is to be used to configure the cabling in the townships. Computer Aided Design, or CAD, is a technical drawing tool in which a user describes the design of a system in a graphical form. These graphical entities, within the CAD system, are the pieces of data in the system which are manipulated by the user.

Initially a plan of the township is set up within the CAD system. This includes the layout of the stands, the roads, the rivers and other physical entities located on the plan. Once this has been completed, the configuration of the cables and other power items are positioned on the plan of the township. Using the CAD system, the operator can optimise costs so that expenses can be minimised. By cutting a fraction off cabling costs, millions of rands may be saved. The CAD system must possess facilities for optimising the routing and thus reducing the length of cable laid around the township and also choosing the most appropriate and inexpensive cable, while still achieving adequate performance. The software must offer facilities for simplifying the design, optimising costs, and choosing the most desirable type of cable and most effective cable configuration.

Over and above this, the CAD system must act as an on-line assistant to the designer. It must perform integrity checks on the data entered by the user, alert of the existence of errors detected, and possibly even take corrective action. The project report outlines a method of structuring the software in such an intelligent CAD system. The object oriented approach is used to create the individual software components making up the intelligent CAD. These components, being independent of one another, can be moulded together to form the intelligent CAD system. The object oriented approach is chosen for a number of reasons but the principle reasons are those of reuse and maintenance. The basic components required in an intelligent CAD system are: a graphical user interface, a list of entities in the CAD, and an expert system.

In order to create an intelligent CAD system, it is necessary to integrate a CAD system with an expert system. Chapter 1 and chapter 2 describe CAD systems and Expert systems.

Scope of research

The aim of the thesis is:

- To identify and analyse the issues relating to intelligent CAD systems.
- To formulate a conceptual framework that supports the integration of expert systems (or knowledge based systems) with CAD systems.
- To propose a specific architecture that meets the requirements of the framework.

The emphasis of this work is not to prescribe paradigms for intelligent CAD systems but to provide the necessary infrastructure to support interaction between expert systems and CAD systems.

Structure of the project report

Chapter 2 provides a brief background to CAD systems. The facilities, advantages and limitations of such systems are discussed.

Chapter 3 describes the basic architecture of an expert system. An expert system consists of a knowledge base, an inference engine and a number of rules making up the knowledge base. Each of these components is separately described. The chapter goes on to describe searching strategies which may be implemented by the inference engine. Finally a frame based and a blackboard approach of an expert system are described.

Chapter 4 describes the requirements of an intelligent CAD system. It further highlights advantages and drawbacks that may exist in such systems.

Chapter 5 outlines some examples from the literature of intelligent CAD systems. The techniques implemented and findings are briefly described.

Chapter 6 provides some ideas for an intelligent CAD system. It outlines the relevant issues for creating an intelligent CAD system. A description is given of the interaction between the different components of the intelligent CAD system.

Chapter 7 describes the software structure of a crude intelligent CAD prototype that has been developed. Each of the components making up the prototype, encapsulated as a class object, is described. The chapter further explains how these objects interact with one another.

Chapter 8 provides the conclusions drawn from implementing the intelligent CAD prototype. It highlights problem areas for such a system and gives suggestions for implementing such systems in the future.

CHAPTER 2 CAD SYSTEMS

Introduction

CAD systems are powerful tools used to represent designs in a graphical format. The user has extensive capabilities to manipulate and modify the graphical information however he desires. The graphical information in the system may be represented by: text, points, lines, polygons and curves.

When CAD systems were first introduced, only highly qualified technical professionals were capable of using them. They ran on powerful mainframe machines, had limited capabilities and were really only crude drawing packages. With the advent of more powerful machines and software, CAD packages began to appeal to a wider spectrum of people. The packages ran on more powerful machines and the software became more user friendly.

Even though CAD systems have become far easier and more flexible to work with than the earlier systems, they are still technical systems. CAD systems are used in a specific field, such as: engineering, architecture, or manufacturing. First time users are faced with a steep learning curve for a number of reasons:

- Users with a shallow knowledge in the field are required to increase their expertise in that field.
- Users need to learn what each symbol that is used in the CAD system represents and understand its purpose.
- Users must understand how the entities in the CAD system interact with, or are related to, one another.

CAD systems can put together complicated and intricate diagrams. These diagrams may easily model the problem domain. CAD systems provide powerful tools to create detailed and accurate drawings. They provide tools to copy, modify and manipulate the entities in the drawing.

By setting out the diagram visually on a computer, the designer can get a better understanding of the problem. CAD is an excellent communication tool since graphical information can effectively represent the information in the design. The diagrams are passed along to different members of the team for scrutiny. The diagrams are easily understood by people fluent in the problem domain. Flaws may easily be picked up by team members.

As has been mentioned, CAD systems possess many useful features that has ensured their popularity. They enable graphical and textual information to be manipulated in many different ways. Some examples of CAD systems include MICROSTATION, CADDY, CATIA, TURBOCAD and AUTOCAD. The next section looks in more detail at the abundance of facilities that such systems possess and describes some of the tools incorporated in such systems.

Features

Gordon[28] describes AUTOCAD as a popular CAD system that is used on a personal computer (PC). This package is an example of a typical CAD system that is widely used in architectural, mechanical, electrical and many other fields. CAD systems operate on inexpensive equipment (such as a PC), are inexpensive systems, contain an abundance of features, and are useful inventive tools. They have been designed by the professional for the professional. Some of the basic features that CAD systems possesses are:

- Interactive menus and a command structure. The menu system is laid out systematically, so that any option is available when you need it. Pull down menus, hot keys (function keys), a mouse and a digital tablet speed up the access of menu options.
- Entities can be: zoomed and panned into and out of, scaled, rotated about any point, copied, moved, grouped together to form more complex entities, stretched or shrunk, saved and retrieved to and from disk and positioned relative to other entities.
- Text may be intermixed with graphics.
- Different fonts and text styles may be selected.
- Entities may be filled in with various types of fill patterns.
- Different types of line types may be selected such as dotted, dashes and continuous.
- Different modes of drawing may be selected such as free-hand and isometric
- An entity, or entities, may be block marked.These marked entities may be cut and paste or their attributes (such as line style) may be modified.
- Entities in the drawing may be interrogated to extract meaningful information from them such as:
    * The midpoint and length of a line
    * The angle between two lines
    * The distance between two points
    * Coordinates and other attributes of an entity
- The attributes of entities can be modified.
- It is possible to locate the closest point to a line within a search area.

A layering system enables entities to be placed in different layers. Each layer may be manipulated individually or in conjunction with the other layers. There is a drawing layer, hatching layer, structural grid layer, piping layer, electrical layer and a layer with construction guidelines.

CAD systems also have a macro facility to deal with repetitive operations. A series of user key-strokes can be recorded to carry out a series of operations. The user's key-strokes may be played back to redo the operations. This facility is extremely useful if the user is required to

carry out many manual and repetitive operations. CAD systems may contains their own programming language and may link in modules from other programming languages. These languages enable algorithms, which carry out defined operations, to be used with the CAD system.

Data

CAD is a powerful tool for manipulating and working with entities. Entities are the shapes in the CAD and are represented by a name, a unique identity and attributes. An entity may be made up of simpler entities to form a *complex* entity. Complex entities may be manipulated as easily as simple entities.

The entities in the CAD system can be stored in a database. Entities relating to specific fields, such as the motor industry, construction industry or any other industry can easily be created and stored in a database and retrieved into the CAD when required. Both complex and simple entities do not need to be redrawn from scratch, but can be retrieved from a database and copied onto the screen.

The CAD's database keeps a permanent record of the design in progress. The information in the database can easily be analysed or modified. CAD systems can update or manipulate the characteristics of entities with common names, identities or attributes. A relational search may be performed to find and replace the characteristics of a specific type of entity. For example all transformers may be changed from a blue outline colour to a red outline colour.

The database enables CAD to become a complete and accurate documentation tool. The user can keep track of the inventory in a design and print different types of reports on the components in the system. A list of entities such as cables, transformers, etc can be kept in the CAD database. Reports relating to the equipment such as the number of cables or the types of transformers can be quickly printed out. By getting these detailed reports, accurate costing of the projects and inventory control can be implemented.

Limitations

In a typical CAD system, entities are represented on the screen by a pixel map or a vector representation. These entities have no meaning associated with them. It is the responsibility of the user to understand what individual entities represent and the relationship between them.

A user may modify these entities and their relationships to one another as he desires. He is responsible for all changes made. There is no integrity check of his design decisions by the system. It is probable that impractical and unrealisable designs are developed. Often

impracticalities in the design are found in the construction phase. Errors resulting from the CAD have serious repercussions in the financial costs. If these errors could be detected in the CAD system perhaps vast sums of capital could be saved.

Present day CAD is a pictorial representation of the design. The CAD itself does not have any intelligence and puts the entire work load and responsibility of the design in the hands of the user. CAD systems lack a problem solving capability. They are tools for representing information in a pictorial form and storing the parameters of the lines and points of the entities in a database. They are not application specific.

In order to solve problems in a specialised application, complex algorithms need be incorporated. These algorithms must be hard coded into the system to suit the application. For example a problem may arise in the CART system which states that a transformer may not lie within ten metres of a consumers stand. Thus whenever a transformer or a stand is located on a drawing, a procedure is run which checks whether it lies within the restricted area. It is time-consuming work and effort needs to be expended in order to debug and solve these algorithms. It is no trivial task to update and modify these algorithms as an applications requirements change.

Although relevant information is captured in the CAD system, each algorithm that manipulates the data has to be explicitly coded. For the typical CAD system, very few algorithms exist and so the data in the system is barely utilised. CAD systems do not provide:

- Contextual help on the entities in the system.
- A description of restrictions applicable to an entity.
- The restrictions applicable to the attributes of an entity relating to the context that the entity is in.
- Dependency relationships that this entity has to other entities.
- An indication of the maximum and minimum allowable values that the attributes of an entity may have.

Thus the amount of interaction between the user and the computer is limited. The user only has a tool to draw the entity on the screen and assign it attributes. The system performs a limited check on the integrity of the system for out-of-bound conditions.

Conclusion

In the past decade CAD systems become more powerful, easier to use and inexpensive to acquire. Today they are in widespread use by technical and lay people alike. CAD systems are becoming more and more user friendly and advanced.

These systems need to be simplified even further so that someone, with limited knowledge in

the domain of the CAD system, is able to work competently and effectively. What is needed is to reduce the learning curve of such systems so that they can be more widely accepted in the market-place.

There is no doubt that CAD systems are impressive tools for representing drawings. They possess many powerful features which can simplify the task of the designer immensely. These systems contain much useful information which is not, and could be, used to simplify the design process for the user. Expert systems, which are described in the following chapter, are perhaps useful tools for making use of this "unused" data in a CAD system.

CHAPTER 3 EXPERT SYSTEMS

Introduction

Expert systems have been in use for over two decades. However, in the past, they have mainly been used by researchers involved in the field of artificial intelligence (AI). Expert systems have always been considered a form of AI and have been associated with other AI concepts such as neural networks and fuzzy logic.

Expert systems differ from the traditional algorithmic approach to programming. Instead of controlling the flow of logic using program algorithms, the control of logic in expert systems is driven by the information which is stored in a special purpose database, called a knowledge base. Expert systems contain a knowledge base of domain specific knowledge. This knowledge is encapsulated in the form of rules. For this reason expert systems are often referred to as rule based systems.

According to Alty[28], expert systems are well suited to deal with symbolic information. They deal with the manipulation of information using logic. The main languages used for expert systems are Prolog and Lisp. Prolog is really an exercise in programming in logic while Lisp (which is a contraction of List Programming) is a means of programming with lists of information.

This chapter serves to describe the components of an expert system. These systems are used to encapsulate the knowledge of an expert in a particular field in such a way that meaningful information can be extracted from the system.

Basic architecture

Expert systems consist of three components (Browston[29]): *Data memory*, the *set of rules* and the *inference engine*. The set of rules and the inference engine are described later in the chapter. Data memory is the global database which represents facts and assertions about the problem. Data memory stores the current state of knowledge during the problem-solving process by holding symbols that represent facts about the domain world.

The state of the data memory is the main control over the execution of the rule based system. Control also depends upon the problem solving strategies of the inference engine. The inference engine is used to execute, or fire, the rules. It determines which of the rules are relevant and which rule is to be tested next. Browston[29] notes three states that the inference engine must have: *match* rules, *select* rules and *execute* rules. Match rules determines which rules are relevant to the specific problem domain and builds up all the rules into a rule set. The *rule set* (also referred to as a conflict set of rules) contains all the rules that will be executed by the inference engine. Often rules are repeated in the conflict set of rules. The

conflict set of rules is passed to the next state, select rules. This state, using some selection strategy determines which rules will actually be executed. The inference engine transfers to the third state, execute rules, by executing the rules selected. Once the execution is completed, the inference engine cycles back to the first state, rule matching where all the rules that need to be selected and executed (a conflict set of rules) are determined and so the process continues. The rule interpreter is used to execute the rules by interpreting the conditions and actions of the rules.

The expert system uses rules as the basic element of data that is manipulated. The structure of the rule is in the form: *if <condition> then <result>*. The conditional part of the rule is referred to as the left hand side of the rule, while the resultant part of the rule is referred to as the right hand side of the rule. The left hand side of the rule usually contains a boolean combination of clauses, such as AND or NOT. A clause specifies a restriction on the value of some entity that may be represented in data memory. The right hand or action part of the rule is generally a list of modifications to be made to data memory when the rule fires. The actions usually add, modify or delete elements in data memory or even read or write data to and from an input output device. For example given the rule:

IF the car has a battery AND If the battery is dead
THEN the car will not start

Should both of the IF clauses be satisfied, then the THEN part or action part of the rule is satisfied. The action part of the rule sets up the status value which might exist in data memory to, for example, StatusOfCar=STUCK.

Knowledge representation

The knowledge of an expert is captured in the rule base in the form of rules. Rules may be factual, rules of thumb or even non-precise. For example if a car will not start a number of reasons for this may exist:

```
1) IF the car has no engine  THEN the car will not start
2) IF the car has no petrol  THEN the car will not start
3) IF the car has no battery THEN the car will not start
4) IF the battery is dead    THEN the car will not start
```

Figure 1 An Example of Rules in a Rule Base

It may be found that reasons 1,2,3 are false. The car has got all the necessary equipment, i.e. an engine, petrol and a battery, but somehow it will not start. After analysing rule 4 it is found to be true that the car will not start because the battery is dead. This simple example

conflict set of rules is passed to the next state, select rules. This state, using some selection strategy determines which rules will actually be executed. The inference engine transfers to the third state, execute rules, by executing the rules selected. Once the execution is completed, the inference engine cycles back to the first state, rule matching where all the rules that need to be selected and executed (a conflict set of rules) are determined and so the process continues. The rule interpreter is used to execute the rules by interpreting the conditions and actions of the rules.

The expert system uses rules as the basic element of data that is manipulated. The structure of the rule is in the form: *if <condition> then <result>*. The conditional part of the rule is referred to as the left hand side of the rule, while the resultant part of the rule is referred to as the right hand side of the rule. The left hand side of the rule usually contains a boolean combination of clauses, such as AND or NOT. A clause specifies a restriction on the value of some entity that may be represented in data memory. The right hand or action part of the rule is generally a list of modifications to be made to data memory when the rule fires. The actions usually add, modify or delete elements in data memory or even read or write data to and from an input output device. For example given the rule:

IF the car has a battery AND If the battery is dead
THEN the car will not start

Should both of the IF clauses be satisfied, then the THEN part or action part of the rule is satisfied. The action part of the rule sets up the status value which might exist in data memory to, for example, StatusOfCar=STUCK.

Knowledge representation

The knowledge of an expert is captured in the rule base in the form of rules. Rules may be factual, rules of thumb or even non-precise. For example if a car will not start a number of reasons for this may exist:

```
1) IF the car has no engine  THEN the car will not start
2) IF the car has no petrol  THEN the car will not start
3) IF the car has no battery THEN the car will not start
4) IF the battery is dead    THEN the car will not start
```

Figure 1 An Example of Rules in a Rule Base

It may be found that reasons 1,2,3 are false. The car has got all the necessary equipment, i.e. an engine, petrol and a battery, but somehow it will not start. After analysing rule 4 it is found to be true that the car will not start because the battery is dead. This simple example

shows how the expert system analyses a series of rules and so manages to pinpoint a solution. If condition 3 was found to be true, ("the car has no battery") condition number 4 would not have been evaluated at all.

All knowledge captured in the rule base is useless without an inference engine. The rule base is just a collection of knowledge in a rule like form. In the above example the resultant part of the rule is "The car will not start". The conditional part of the rule is one of the bold clauses ,in figure 1, such as "the car has no engine". The expert system needs an inference engine to search for a rule, find the matching rule and infer an additional rule.

Fox[20] states that expert systems form a natural way to express knowledge. They are a compromise between a complete declarative but inefficient representation of knowledge (where knowledge is defined explicitly) and a procedural representation that cannot be manipulated by programs. In an expert system, knowledge is related to other knowledge and is also related to procedures which manipulate this knowledge. For example, if a rule condition is satisfied, then additional facts and assertions on the data in the system may be established and additional rules in the knowledge base become candidates to be checked. The knowledge accumulated so far may be modified by procedures which manipulate this knowledge.

Rules

Lucas[15] notes that expert systems offer a simplicity of control. All facts are stored in a similar form. Since the knowledge base and control is independent of one another, rule based systems can cope with unanticipated situations.

Lucas[15] describes rules as being modular, thus additional rules can be easily added, with few side effects. This simplifies the creation, addition and maintenance of the system. Rules are easy to explain because they represent separate packets of knowledge. Their modularity makes knowledge descriptions easy to update.

It is possible to keep a history of which rules have been selected and which have fired successfully and which have not. The trail of reasoning which leads to a conclusion may be logically followed and understood.

According to Lucas[15], expert systems are advantageous when the rules are unordered and may not be organised as sequenced instructions. They should be used when:

- The knowledge to be programmed naturally occurs in rule form.
- The program's control is extremely complex.
- The program is expected to be significantly modified over time.

Rules may be structured in a variety of ways. A common approach is a tree hierarchy (Lucas[15]). As rule conditions are satisfied, the inference engine works at a higher or lower level of the rule tree, depending on the type of search strategy being used.

Inference

An inference engine is used to manipulate the rule base to extract more meaningful information from it. It reads in a rule and determines whether the conditional part of the rule is satisfied. If it is, it modifies the facts and assertions of the data in the system and from this infers the next rule. If the rule is not satisfied, the result is that the condition is untrue and it looks up another type of rule in the rule structure. It continues to infer rules until a result is found or no result can be found.

The inference engine needs a parser incorporated in it to compile the information from a rule and put it in a format with which the inference engine can work. The parser should be able to deal with rules that are laced with multiple AND and OR conditions and pass each of these conditions that is being tested to the inference engine.

The inference engine performs the following functions:

- Recognises immediately achievable goals
- Expands goals that are not immediately achievable into simpler subgoals
- Takes the necessary action to achieve primitive goals.
- Backups the solutions of the subgoals to yield the solutions of their super goals.
- Maintains a goal tree to store the explanation of the decision making process.

Search strategies

According to Fox[20], Rule-based systems are an extension of a searching strategy. In an expert system rules are organised towards a solution. If a rule condition is satisfied, the solution process is focused to other rules which may then be tested. A solution is finally obtained by finding the rules which lie closer and closer to the solution. Rules may thus be thought of as search operators. After a rule condition is tested, the solution space is reduced. The inference engine continues to test rules which are closer to the final solution. Rather than having a small set of rules which generate a large search space, a large set of rules generate a small search space. A rule's condition is specific enough to identify the problem and generate a search space which is much smaller while still containing a solution.

Generally speaking the narrower the domain of expertise, the fewer rules are needed to

capture this knowledge of the domain. As the domain of expertise gets more detailed it becomes more difficult to encapsulate the expert's knowledge. Expert systems work exceptionally well for simple systems who's expertise can be effectively captured in a knowledge base.

The reasoning strategy employed by the inference engine may be either forward or backward chaining or a combination of both. Both backwards and forwards chaining use various types of chaining strategies to search a tree structure of rules in a knowledge base. According to Powers[15], there are four basic methods that can be used to search the tree: *breadth first search, depth first search, bounding methods* and *heuristic guidance methods*. Depth first searching involves searching each branch to its furthest depth. Breadth first searching involves searching an entire level before moving to lower levels of the tree. With the bounding search method, an evaluation is determined of how well the search is progressing. This solution is used to place bounds on the search space so that certain solutions need not be investigated. Heuristic guidance limits the search space by applying approximate principles or rules of thumb on the present information in the system.

Forward chaining

Forward chaining involves trying to determine a goal by hypothesising on existing facts and assertions in the system to prove the goal. The inference engine compares the facts and assertions in the system with the rules and determines if the result is satisfied. It works downwards from the top of the knowledge tree to the lower levels. The matching of the inference engine has no effect on the state of the data memory. i.e. read only access to data memory. For example from figure 1 the rule states: "if the battery is dead then the car will not start". Forward chaining examines the conditional part of the rule, i.e. "the battery is dead". If the conditional part of the rule is found to be true, then the resultant part of the rule is satisfied, i.e. "the car will not start".

Powers[15] states that an advantage of working forwards is that detailed evaluation is often possible since the preceding states are known. The search can be reduced further by using bounding methods, which has been described earlier. The major limitation is that the search may not lead to goal states and hence effort is lost. Additionally, the initial states may be many and hence a large search space results.

Backward chaining

In *backward chaining* the right hand side or resultant part of the rule is used for matching. The rules that can achieve the goal are sought during the matching process. If the rule conditions are not immediately true, the conditions may be used to establish new subgoals.

Goals are broken down into subgoals until the result is a collection of goals which are attainable. Backward chaining involves trying to prove a result by stating that if the goal is to be satisfied, certain conditions have to be met. Thus it works upwards from the bottom of the knowledge tree. For example, from figure 1, it is known that the car will not start (the solution). Working backwards it is necessary to check all the reasons that the car will not start. Since the car has an engine, petrol and a battery, each of these components need to be tested further. It is then found that the battery of the car is dead. Thus the car will not start because the battery is dead.

Powers[18] mentions that the advantages of working from the goal state to the initial state, using backward chaining are:

- All paths end at the goal state.
- No paths are wasted by generating states that do not end at the goal state.
- It is the only method that can be used when the initial state cannot be defined explicitly.
- The isolation of control knowledge makes the rules easier to understand and explain.

The disadvantages of this approach are that detailed evaluation of the steps preceding the current state are not yet known. Backward chaining has less flexibility than forward chaining systems.

Blackboard systems

A blackboard approach is a system architecture that employs a database of memory that is common to several processes, where each is a knowledge source. The knowledge sources are unaware of one another but they all share the centralised, global data area called a *blackboard*. The blackboard stores the current state of the solution. Typical state data that may exist on the blackboard includes input data, partial and final solutions, hypotheses and control data.

Each knowledge source updates this common database. The knowledge sources contribute information that will lead incrementally to a solution. Each knowledge source is responsible for knowing under which conditions it may contribute a solution. Thus with the combination of all the knowledge sources having a background in a particular area of knowledge, they all combine to solve the problem.

Blackboard architectures, of an expert system, use multiple knowledge sources and multiple lines of reasoning according to Alty[30]. Each knowledge source is a set of rules used by an individual 'expert' to tackle a particular aspect of a problem. When a knowledge source is activated, it creates a new hypothesis which it writes to the blackboard. A scheduler controls

which knowledge source to activate next. If two hypothesis are in conflict the scheduler determines which knowledge source to activate.

Frames

Frames provide a formalism for grouping the properties of individual entities. A frame is a data structure which represents an entity type. A frame's properties are described by *fields* or *slots*. The knowledge is organised hierarchically in a frame hierarchy or frame taxonomy. Frames are organised into an inheritance taxonomy, with the subclasses inheriting from their superclasses.

In a frame approach, the system tries to find values to fill the unfilled slots of a frame. The properties of the slots which are known are used to select a candidate frame. If inappropriate values are found then another candidate frame must be selected. A frame is not found if insufficient knowledge is supplied. In effect the system tries to find a frame which best matches the known properties of an entity. When a sufficient match is found, the system can infer the existence of an entity type represented by the frame.

Slots may be filled by values or by pointers to other frames. When the slots are filled or partially filled, a frame is said to be instantiated. Frames may be used to represent an entity from different points of view. Slots may have the following characteristics:

- Contain generic, i.e. preset values, which occur in every instantiation of that frame.
- Take on default values if no other values are known.
- Have conditions where their values might only lie within specified ranges.

Frames may be linked to other frames in various ways. A slot in one frame may be filled by another frame. Brolio[5] mentions demons or procedures which are executed when data field values are given to slots. These procedures, which are attached to a slot, may be invoked if a certain condition is not satisfied for the slot. For example the procedure may request input from the user for a particular slot value. Frames are used for inference and for checking a body of knowledge for inconsistencies and omissions.

Capabilities

Expert systems offer a number of advantages which make them more attractive compared to conventional programs:

- Provide permanently available human expertise.
- Have greater accessibility to this expertise.

- Offer the possibility for the human expert to get a second opinion.
- Explore different alternatives through the use of 'what if' questions.
- Have the capability to solve problems requiring a great deal of knowledge in situations where data is incomplete and expertise is unavailable
- Offer the opportunity to solve problems that cannot be solved as well as humans or conventional data processing techniques, such as cases where a large number of cases must be explored.

Limitations

Expert systems have serious limitations mainly because they are in the early stages of development. They have a number of problems:

- Knowledge can be difficult to represent.
- It is difficult to establish the degree of confidence in the outcome of facts or conclusions.
- Rules are restrictive as a form of knowledge representation for common sense knowledge and implicit relationships.
- Rules have a surface knowledge only (heuristic knowledge) but do not have a deep knowledge, i.e. a knowledge of principles, structures, or the functions and behaviour of objects.
- It is a laborious and difficult process to extract knowledge from experts.
- Data acquisition is the bottleneck in the building up of the knowledge base.

Conclusion

By capturing the rules that an expert uses and by having a mechanism that can deal with these rules, it is possible to come out with a prognosis which would compare favourably with an expert's. Expert systems should not be thought of as something which might completely replace an expert, but as a tool to assist a less knowledgeable person who has much less experience than the expert.

Alty[30] mentions that the power of an expert system depends on how knowledge is captured in the rule base and how effectively the inference engine can manipulate the data in the rule base. An expert's knowledge and even experience can be captured in a knowledge base. This knowledge can be used by non experts in the field.

The rules within expert systems are not placed in the program code of the application. Rather the application reads the knowledge from a knowledge base external to the program. The knowledge base may be changed at any time, even during program execution. The control of

the system can be varied by changing the logic in the knowledge base (Browston[28]) rather than digging into the actual code itself. Control can also be varied by modifying the way the inference engine looks up rules at run time. This makes expert systems very attractive to work with.

Expert systems capture information in the form of rules which are stored in a knowledge base. There are a number of advantages in structuring the information in the form of rules. Since the rules are not hard wired into the program's internal structure, a flexible system is achieved. Although expert systems are difficult to debug, a programmer with less technical expertise may be capable of doing so. For an algorithmic approach the program needs to be recompiled with every change made. In order to debug a program, it is necessary for the user to have a thorough working knowledge of the programming language that the algorithms were written in. However, to debug a knowledge base, less technical expertise is necessary because the user does not need to become entangled in the programming language. He only needs to understand how the information is structured in the knowledge base and have a good understanding of the information structure of the knowledge base.

Typically, expert systems consist of the current state of the data in the domain, the information structured in the form of rules, and a control mechanism used to read these rules and infer additional rules to be looked up. The control mechanism (or inference engine) matches the rule, selects the rule and then executes the rules. This process continues until the rule has been satisfied, or the rule has been proven to be false.

The expert system needs to evaluate multiple conditions which are logically related to one another, with the AND and OR logical operators. A parser is needed to evaluate a string of multiple AND and OR conditions which are grouped together. This parser is part of the inference engine of the expert system.

Expert systems possess the capability to extract useful information from a system. Expert systems could be used in conjunction with CAD systems to extract meaningful information from the entities in the CAD. The following chapter describes how expert systems can be used together with CAD systems to create an intelligent CAD system.

CHAPTER 4 INTELLIGENT CAD

Introduction

CAD systems possess many remarkable features. These systems could become even more productive by providing them with "intelligence". *Intelligence* enables the CAD system to extract meaningful and useful information from the system. This information can be passed on to the user to inform him of any incongruencies in the design.

Intelligent CAD provides the CAD system with additional features that makes the CAD more user friendly to work with. It is as though the user has an on board assistant to guide him.

An intelligent CAD system can be achieved by giving entities in the CAD additional features and by attaching rules to these entities. These rules determine the constraints and limitations of the entities. In an intelligent CAD system, entities have a whole set of attributes associated with them. Each entity is capable of checking whether it is in an invalid state by virtue of the values that its attributes have.

When an entity in a CAD system is manipulated, the system verifies if the entity has caused an invalid configuration in the design. Each entity must be able to check that all the relationships that it has to other entities are valid. In this way the system can ensure a more robust design using CAD. In typical CAD systems this is the responsibility of the user.

There are a number of factors which need to be traded off for intelligent CAD systems. A strongly coupled system leads to a less maintainable system than a weakly coupled one. A weakly coupled system is slower than a strongly coupled one. A system can be thought of as a collection of black boxes, with each performing a specific function. For a strongly coupled system these black boxes interact a lot with one another and have much interdependence. In order to make a change in a single black box of a strongly coupled system, it is necessary to change other black boxes which have a dependence on the modified black box. Maintainability of the software may need to be traded off for speed and efficiency.

Requirements of Intelligent CAD

Latombe's[16] objective in using an expert system in computer aided design is to blend the designer and the computer into a problem solving team so that design problems can be tackled more efficiently than working alone. The design task is allocated between the designer and the computer according to their capabilities so that each of them draws benefits from the work of the other. This frees the designer of a number of chores in the design of the system and so gives him more time to expend efforts in the problem definition stage of the design process. The computer should perform design automation on a subprocess of the design process that can be efficiently represented by the computer. The difficult aspects of the design

process, which are difficult to program, may be more proficiently performed by designers. CAD systems could be used to bring new kinds of resources into the system and to help to achieve a better working team between the designer and the computer.

CAD and Expert systems are completely different from one another. CAD systems display graphical information upon a screen while expert systems interpret textual information in their knowledge bases to come up with a logical conclusion.

In order to integrate an expert system with a CAD system, it is necessary for the CAD system to share the information about its entities with the expert system. The expert system should be able to verify the information represented in the CAD database. Should the expert system find that an entity causes an invalid configuration, then it should pass a message to the CAD system. The CAD system should be able to deal with the request and may respond by displaying a warning message on the screen, highlighting or even deleting the offending entity.

Each component of the CAD system should work separately and independently from the other parts of the CAD. The drawing part of the system should be completely unaware that it is integrated with a rule base. By separating these functions, the system becomes easier to maintain because a change to a module is not propagated through to other modules. The expert system should not even know that the graphical interface exists. All it needs to know is that it is fed facts, which it analyses and comes up with results which it returns. Some way must be found to integrate the drawing part of the system and the expert system so that they may work independently of one another.

Role of CAD

CAD systems need to become more application specific. It is necessary to store more than the pixel or vector representation of the entities. The CAD system must be more than just a collection of entity names and attributes gathered in a database. Entities should be given meaning. They should have attributes associated with them. These attributes should automatically be determined when the entities are created, manipulated or destroyed.

Entities may have relationships to other entities. These are set up when the entity is created or manipulated or when an entity needs to know of the existence of relationships with other entities. These relationships should be filled in or removed dynamically during the execution of the program. The system should be able to reason with entities which are explicitly or implicitly related to one another. Entities have explicit links if they are related directly to other entities, such as a cable may be *connected* to a transformer. Entities have implicit links by virtue of implied relationships to other entities, for example all cables and transformers which are enclosed by a stand, have a *belongs to* relationship with the stand entity.

In conventional CAD systems, if an entity is in an invalid state, then it is up to the user to

validate that it is not the cause of an error in the design. For example if only one end of a cable is connected and the other end not connected, it is completely up to the designer to pick up the non terminated end of the connection. A cable should really know if it is in an invalid state and should inform the user of it or carry out some predefined action to rectify the situation. The system should be able to check its own integrity.

Both conventional CAD and intelligent CAD consist of entities which contain attributes. A difference between the two is that intelligent CAD also contains a set of relationships to other entities. Thus an entity may have a relationship or a number of relationships to many other entities.

Role of expert systems

Expert systems offer an ideal solution for capturing knowledge about entities in a particular domain. For example cables or transformers may have specific rules attached to them. All th could be captured in a knowledge base. Each entity in the system should have a whole set of rules attached to it. An example of a rule which pertains to both a cable and a stand is: "a high voltage cable may not pass through a residential stand". This rule is satisfied if the two objects which are related to one another are a stand and a cable. The cable has a voltage attribute value of high voltage and the stand has a type attribute of residential. Either the cable entity or the stand entity uses this rule to check if it is in an invalid state.

Capabilities of intelligent CAD

If an algorithmic approach is used, the complexity of the control structure in a program increases dramatically. By having much of the structure of the program encapsulated in a rule base, a more easily maintainable program results. The sequence of events of an application is decided at run time, while the complexity remains constant.

CAD systems can become more evolutionary to use. Knowledge can be represented by a collection of data which can be relatively easily updated.

CAD systems can be useful tools for solving very new problems. The difficulties in describing the knowledge in the system may reveal the designers misunderstandings and may suggest new knowledge to be collected. The designer can spend more time at the problem definition stage of the design process. The description of the problem knowledge allows him to tell the computer what he wants without having to be concerned with the details of attaining it.

The expert system forces the designer work to in a more constrained environment. Thus he

has less flexibility than he would have before, which reduces the margin for error. Errors in a project may be picked up quite early in the design stage, rather than in the construction stage.

Limitations of intelligent CAD

There are a few drawbacks resulting from combining CAD with expert systems. The principle one is that much effort is needed to create new rules into the system and to maintain the rules. However, the structuring of information in a rule like format offers many advantages.

As additional types of entities become incorporated into the CAD system it is necessary to update the knowledge base with these entities and to append additional rules, concerning these entities, to the knowledge base.

Modifications to the rule base are necessary. The cost of maintaining the knowledge base may be high. Rules may need to be modified and edited with changing requirements in time. With the introduction of new entities into the system, additional rules need to be added to cater for this.

The conditions for system validity is captured in the rules. Thus rules which are ambiguous or inaccurate may result in errors.

The time taken to evaluate rules may be large. This would cause the intelligent CAD to execute too slowly to be practical for an on-line, real time system. A way must be found of balancing the processing of the system so that the system is practical to work with in real time, while at the same time able to perform all the necessary system integrity checking. In order to accomplish this it may be necessary to carry out some of the processing in an overnight batch run or on another machine on the network.

Conclusion

It is highly desirable to use expert systems with CAD systems. Very crude intelligent CAD systems can provide great assistance to users. However the price is great in developing and maintaining such systems.

Intelligent CAD systems provide each entity with the facility to validate its own integrity when it is created or manipulated in the CAD system. This enables the designer to locate errors early on in the design process.

In the past much research has gone into finding ways of combining CAD systems with expert systems. The following chapter describes a few approaches that others have taken.

CHAPTER 5 EXAMPLES OF INTELLIGENT CAD SYSTEMS

Introduction

The previous chapter has discussed what an intelligent CAD system is required to do. This chapter examines what other researches have implemented in order to achieve a similar kind of an intelligent CAD system. Relevant issues are brought up such as: entity representation, knowledge representation and the different types of knowledge that can be incorporated into an intelligent CAD system.

Representation of knowledge and entities of the CAD system

For the design of knowledge based systems in CAD, Encarnacao[17] suggests that the following problems need to be solved: entity and knowledge representation, system control and knowledge acquisition.

Entity representation and *knowledge representation* involves the representation of the entities (with their geometry and graphic structure) and the representation of the knowledge about the entities. Knowledge is associated with each individual entity as well as with relationships to other entities.

System control concerns the man machine interface, graphic dialogue, access to the methods manipulating the data in the domain, the methods needed for the problem solution, and implementation of the inference between the knowledge, the problem description, and the related entities .

Knowledge acquisition is concerned with building up the knowledge data base. The data in this knowledge base should be structured so as to minimise redundancy, limit ambiguity and maintain consistency.

Encanacao[17] states that the implementation language and the corresponding graphics interface should be able to interact with one another. This could for example be an interface between Prolog and GKS (Graphics Kernel System). GKS is a graphical standard that many CAD packages use in the display of their graphical output.

A *User Interface Manager* (UIM) handles all the interaction between the user and the application. The knowledge acquisition and User Interface Manager (UIM) needs to be set up so that the User Interface Manager and the dialogue component of the knowledge base may interact with one another. This enables the task of building up the knowledge base to become interactive. It is important to be able to easily and effectively build up the knowledge base, which controls the flow of program execution.

The system comprises a *knowledge base*, *data base* and *methods base*. The knowledge base contains the rules of the system, the database contains the entities in the system and the methods base contains the procedures that use the information in the knowledge base and from that modify the entity information in the database. These databases, with User Interface Management support, need to be integrated so that the system may work with the information in the different databases.

Representation and search of data

Powers[18] maintains that the two basic features of problem solving are *representation* and *search* of the information in the system. His paper discusses the different types of representation and search strategies. A well structured representation results in a small search space. His objective has been to develop an efficient method for looking for the solution in the representation that has been selected. Powers mentions three general representations: *state-space representations*, *decompositions representations*, and *theorem-proving representations*. State-space representation involves searching from an initial state to a goal state. In decomposition the problem is broken down into smaller subproblems so that the original problem reduces to a set of trivial problems. Theorem proving representation makes use of boolean algebra to solve problems. Powers examines each of these representations and discusses their pros and cons.

In state space representation, operators cause a transition from one state to another. Each state is characterised in terms of attributes associated with it. Operators determine if an attribute of a state has the proper value to pass on to the next state. The states making up the state space form a tree like structure. The operators determine if a lower down state in the tree satisfies the present state.

When solving a problem by decomposition, a large problem is broken down into a number of smaller problems. Each of these subproblems is logically tested using the AND and OR logical operators. The principle of "divide and conquer" is implemented by solving the logical conditions of the subproblems that make up the decomposed problem. The solution of the system is the result of the subproblems combined in such a way that they are ANDed and ORed with one another resulting in the solution of the problem.

Predicate calculus can be used as a language for problem solving. It uses boolean algebra to represent information. The AND, NOR, NOT and other boolean operators are used. Set theory is used to group common information together and the UNION and INTERSECTION operations may be applied to the sets of information. This approach offers uniformity in representation and uniformity of searching in the problem space.

Chapter 3 discusses the various types of searching strategies that Powers recommends, i.e. breadth first search, depth first search, bounding methods and heuristic guidance methods.

Graphics Programming for knowledge guided interaction

Tibbert and Bergeron[20] claim that a CAD application ought to incorporate knowledge-based modules into the interaction process of an application as easily as analysis modules have been utilised in conventional software. The success of combining analysis modules with graphic modules corresponds directly to the ability to logically separate those functions.

In their research, a prototype was constructed which determined the configuration of computer rooms and the components which were placed within them. Tibbert and Bergeron found that heavy computation was required by the system which made it impractical to work interactively. Constraints which were not part of the physical world and were thus unlikely to change were built called into the system. Tibbert and Bergeron believed that the intelligence of the entire design would be improved if obvious mistakes were detected in real time by the interaction interface. Two modes of operation were used: an *interactive mode* and a configuration mode. Simple rule conditions such as "disks of type Y must be within 15 feet of CPU X" required little processing and were easily incorporated into the interactive specification module. Rules which were far more complex took longer to solve and so remained in the configuration module. The configuration module contained the list of entities which needed to be validated. This module contained more complex rules which took longer to evaluate and were solved at configuration analysis time. Configuration analysis was a dedicated batch procedure which operated when all the users were out of the system. Each entity which was placed in the configuration module was tested and validated. A report was generated by the batch program, which ran for several hours. This report contained a list of all the entities which were in an invalid configuration.

In order to produce a flexible interactive program that is capable of utilizing a knowledge base, it is necessary to address two major aspects of the design: the *interaction handler* and the *knowledge base*. The interaction handler organises the visible predefined areas on the screen, including menus, prompts, command name, title and sketch area. Figure 2 shows the basic structure of such a system with the knowledge base in between the interaction handler and the database.

The interaction handler is implemented as a *finite state machine*. The transitions to a new state is determined by input events, where input can come from either a user or the knowledge base. Its current state determines what action it is to take. From the information returned from the knowledge base or the user, the interaction handler can transfer to another state. The interaction handler determines the current state the system is in and displays the response or course of action to the user. As has been mentioned earlier in this chapter, Powers[18] uses a state space representation where operators determine if a transition is made to the next state. Finite state organisation was adopted to allow the knowledge base to guide the interaction.

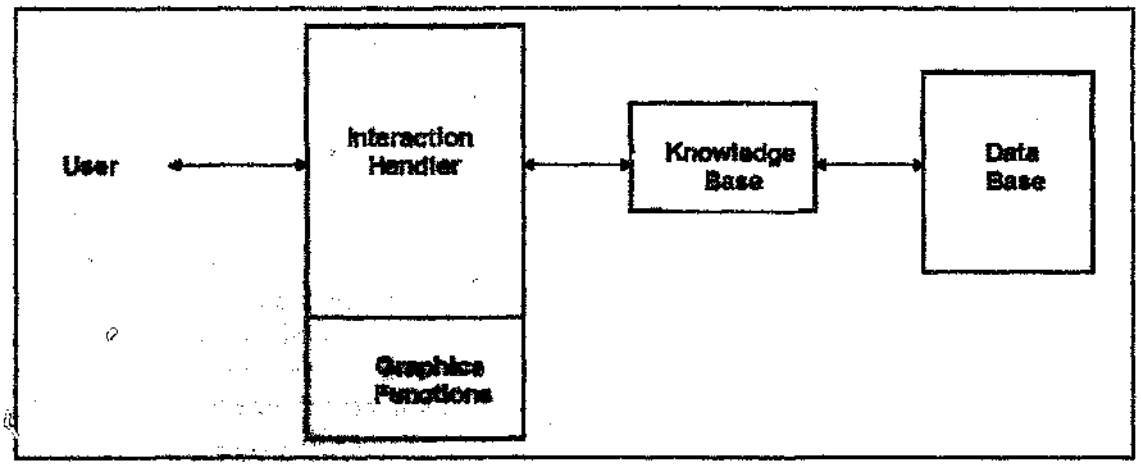


Figure 2 Integrating the Interaction Handler, Knowledge Base and Database (Reproduced from Tibbert and Bergeron[19])

Knowledge base functions succeed or fail. When a function fails, the reason for failure is also returned. The knowledge base consists of four distinct components: a set of *facts about the database*, a set of *facts about the interaction program*, a set of *rules* and an *inference engine*.

The knowledge base must have two kinds of facts about the data base. The knowledge base must be able to access the information in the data base (entities and attribute values applied to entities). It must also have knowledge about the data base information, the kinds of entities in the data base and how to access information about those entities. The knowledge base must have knowledge about the fundamental functions which can be invoked by the interaction handler. It need not know the semantics of these commands and how to invoke them.

The rules provide two basic types of constraints: consistency and completeness. Consistency rules prevent data base updates that cause a rule to provide the incorrect result. Completeness rules guide the user in the information acquisition process. These rules relate the state of the data base to the interaction handlers commands in order to direct the user to enter needed data. Every rule field includes a response field which is returned to the interaction handler in the event of a failure as a result of processing a rule.

The inference engine interprets the rules to determine if consistency or completeness violations have occurred and to indicate to the interaction handler the results of the rule processing (via the response portion of the rule).

The knowledge base does nothing to change the flow of control of the program, beyond reporting errors. The knowledge base searches the data base for the first unsatisfied condition and returns the command which will satisfy that condition as well as the associated response. Adding new rules or revising rules only requires changes to the knowledge base. The

interaction handler or the graphics program do not need to be modified if no new basic objects of program commands are utilised in the new rule.

In typical menu driven systems the prompts, menus and help have a statically defined relationship. Tibbet and Bergeron have attached a knowledge base to the menu system making the menu options contextually available and so the system becomes more friendly for the user.

This intelligent interface requires a knowledge of which pieces of information are important, and needs rules stating how to direct the graphical program to acquire this critical information. For very complex menus it would be useful for a designer to be able to specify the details of interaction importance using traditional interactive dialogue and then to switch to a consultative mode. In this mode the program would guide the interaction and focus the designer's attention on key issues as defined by an expert in the field.

Tibbet and Bergeron's[15] XGRAPH system shows a working example of the use of knowledge engineering techniques with interactive graphics. It provides two major functions: to enforce data base integrity constraints and to provide a knowledge guided interaction dialogue. Database integrity constraints simplifies adding information into the knowledge base since it is possible to view the information that is stored in the database graphically. Any additional rules that are added into the knowledge base are tested for their consistency and validity. This ensures that all information placed into the knowledge base is checked. Knowledge guided interaction dialogue aims at attaching a knowledge base to the user interface so that menus become contextually sensitive. The user interface is greatly improved by having a menu system that can adjust itself automatically to the context that it is in.

The Tropic intelligent CAD system

The Tropic system is an example of a computer aided design system that incorporates an expert system. With his Tropic system, Latombe[16] finds that a general purpose expert system needs *problem solving* and *knowledge representation.*

Problem solving is concerned with finding solutions to problems for which no straightforward solution process is explicitly known before-hand. Knowledge representation is aimed at the representation of facts, laws, actions or judgemental statements so that they may be efficiently used by other components of an intelligent system. Due to the complexity of the problems encountered by the Tropic system, it has been realised that traditional algorithmic techniques that require every particular case to be explicitly described, are inadequate for the complexity of these problems.

Database management should be performed by the computer to represent the model and to access large amounts of data. Database systems may be manipulated by expert systems to

provide common sense reasoning capabilities. The computer may maintain a context for each problem or subproblem which is used to restrict only the relevant information to be accessed. This approach is similar to Tibbert and Bergeron's[19], mentioned earlier in this chapter,who suggest the use of an expert system together with an interaction handler in order to determine a context so that information pertinent to that context may be freely accessed.

Objectives of the Tropic system

In the Tropic system, the user describes the problem by stating the domain in which it lies and the goals that must be achieved in the solution. The task of the system is to solve the problem and report a solution. The steps that lead to the solution can be viewed and analysed. The Tropic system can choose among several alternative steps by correlating their anticipated effects with the goals to be achieved. The system can also correct its bad choices and return to the point at which a wrong approach has been taken.

The Tropic system aims at *design automation*, *man machine interaction* and *database management*. In design automation the computer is used to assist in all possible subprocesses of the design process. The man machine interaction allows the computer and user to carry out the parts that they are most efficient at and thus reduce the work load of the user. Database management concerns the most efficient way of representing information relating to the CAD and the expert system.

The main objective of the Tropic system is to determine what type of knowledge representation is required to solve design problems.

Structure of Tropic

The overall structure of the Tropic system is shown in the data flow diagram of figure 3. It contains a *knowledge selector*, a *design problem solver*, a store of the *relevant knowledge*, the *domain, problem and problem solving knowledge*, the *model data* and the *control data*.

The Tropic design problem solver contains the *structural problem solver* (SPS), the *numerical problem solver* (NPS), the *model data* and the *control data*, as shown in the data flow diagram of figure 4. The structural problem solver and the numeric problem solver produce an analysis graph, from the *MPR's* (model production rules), that they use to analyse the data. This analysis graph is generated in the form of an AND/OR tree called a *model production tree* which is part of the control data maintained by the system. The solved and unsolved decision points, achieved and unachieved constraints and gathered solution means are recorded in the Control Data. A representation of the state of the current solution is maintained. This model produced so far is stored in the Model Data.

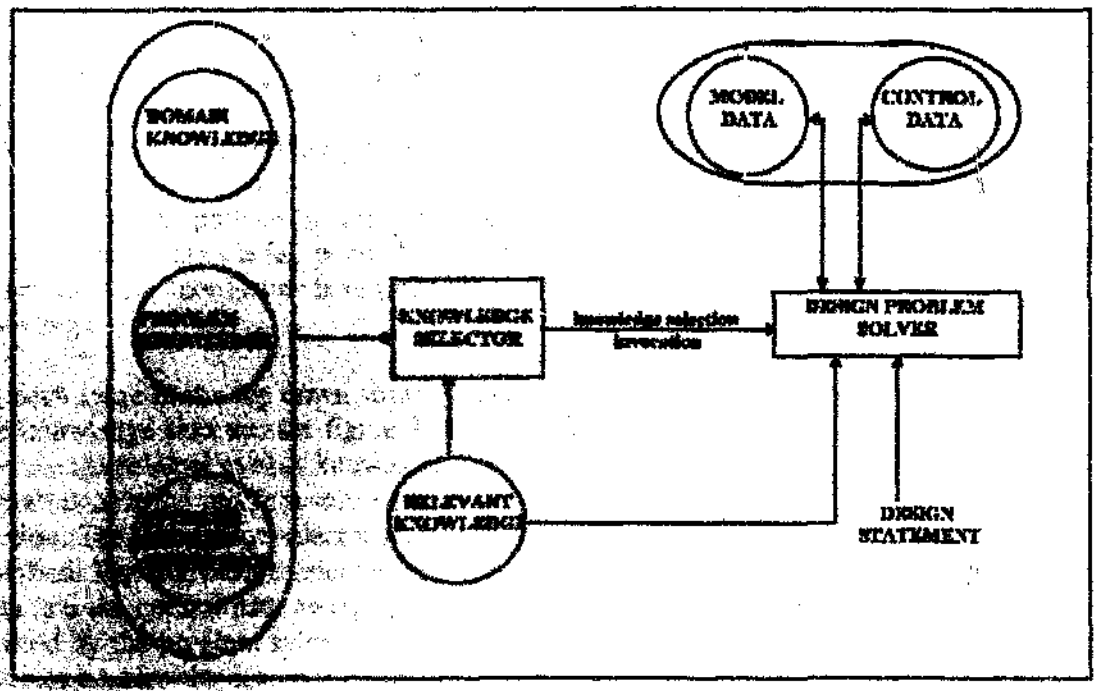


Figure 3 The Tropic System Design
(Reproduced from Latombe[16])

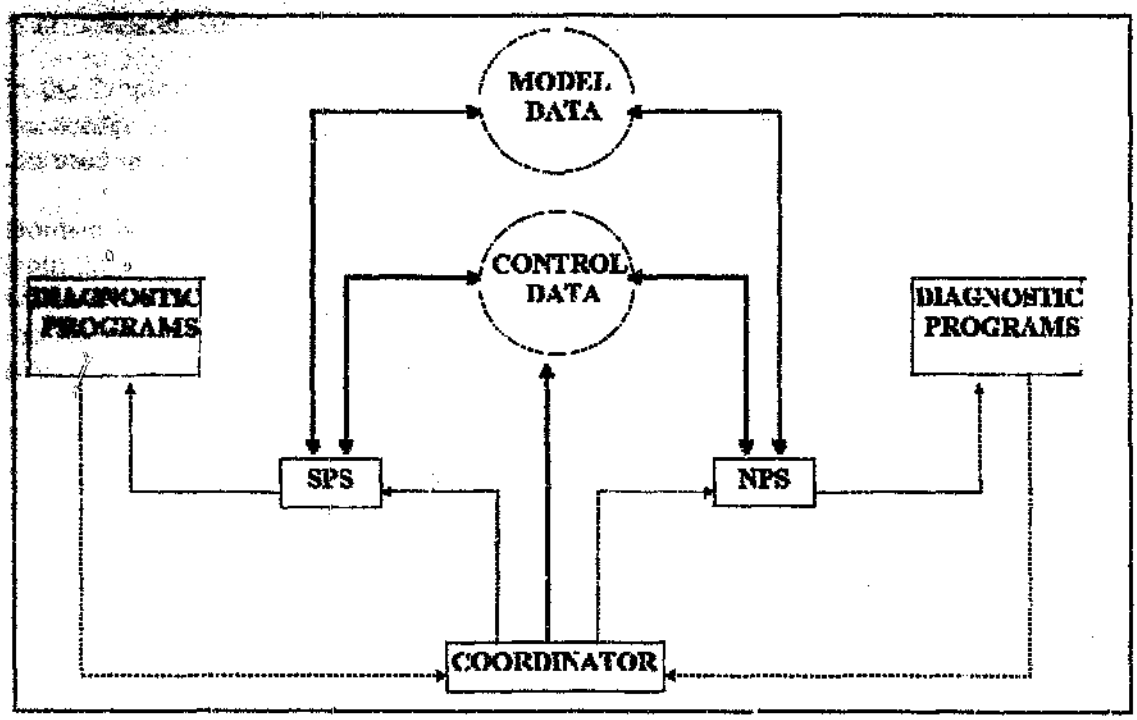


Figure 4 The Tropic Design Problem Solver
(Reproduced from Latombe[16])

In the Tropic design problem solver, activities are coordinated by the *coordinator*. The coordinator calls the structural problem solver for achieving the structural constraints, and then the numeric problem solver for achieving the numerical constraints. When one of these programs encounters a fail point, it calls diagnostic programs (identifies a decision point to return to). The coordinator is responsible for returning to this point and making the necessary updatings.

At each stage in the top down strategy, the *coordinator* (of the design problem solver) calls the knowledge selector, see figure 3 of the diagram of the overall Tropic system. The knowledge selector evokes knowledge from either the domain knowledge, the problem knowledge or the problem solving knowledge. This knowledge is described in the next section. The knowledge selector matches the conditional part of the rule, i.e determines if it is satisfied. The knowledge selector takes the user's description of the knowledge, the model data and the control data as inputs and returns its output into the relevant knowledge, which is used by the problem solver. The independence of the problem solver to the knowledge selector enables the designer to modify his description of knowledge during the problem solving activities of the system.

Knowledge of Tropic

In the Tropic system[16], knowledge is divided into three areas: *domain knowledge*, *problem knowledge* and *problem solving knowledge* (See figure 3). These different knowledge sources are used in conjunction to focus on the knowledge which is pertinent to the problem on hand.

Domain Knowledge determines the solution space of the system. This knowledge is collected from the domain in which the problem lies. It contains details and concepts of the entities which exist in the problem domain. As has been stated by Fox[20], (in chapter 3 of the report) the search space in which the solution lies is reduced by having a large enough number of rules which describe the domain information of the system.

The problem knowledge describes the design constraints of entities in the system. Each entity may have limitations imposed upon it. An entity may be restricted to have values in a certain range or limited to take on specific values. Problem knowledge determines the goals to be achieved and a feasible region of the solution space. This knowledge provides additional rules from a different perspective, which reduce the search space in which the solution lies.

The problem solving knowledge is made up of advice intended to guide the solution process devised by the system. The solution space is divided into a feasible region and an infeasible one. Tropic attempts to find the solution lying in the feasible region. Heuristics, based on experience and intuition, is called "problem solving knowledge". It contains pieces of advice which guide and speed up the search process by reducing the search space of the relevant knowledge even further. Problem solving knowledge contains designer's advice on the way to

attain a solution in that feasible way.

Problem solving knowledge, treated as a set of secondary constraints, enables the expert system to choose paths which are more likely not to lead to a fail point (point at which the system is stuck and unable to come to a conclusion). Once an error has been encountered, the system "learns" to avoid any path that could lead to that fail point and ensures that the same error cannot be generated by the system again.

Problem knowledge, domain knowledge and problem solving knowledge are required to solve the design problems. The Tropic design system should: accept problem, domain and problem solving knowledge described by the user, gather and accept the means of the solution from this description, and devise a user-oriented trace of the system so that the user can easily correlate the steps of the solution process.

System inputs

Items of input to the system are gathered into four sets: *model production rules, analysis schemata, the constraints to the problem knowledge* and the *pieces of advice to the problem solving knowledge*. Model production rules and analysis schemata correspond to domain knowledge. The constraints apply to the problem knowledge. The pieces of advice correspond to the problem solving knowledge.

The purpose of model production rules is to provide the system with data so that it can gather information to generate models. Model Production Rules (MPR's) describe entities in terms of attributes, other entities and links between entities. MPR's produce auxiliary data to be used for evoking new MPR's. MPR's have a context and will only trigger once a condition in the MPR is satisfied. Thus an MPR's condition may only be executed much later on in the execution of the program.

Each entity in the Tropic system is represented by a name, a type and a number of attributes describing the characteristics of the represented entity. The attributes may be structural, having a set of predefined values, or they may be numerical. Entities may be linked hierarchically, inheriting properties from the higher level entities, or associatively, having relationships to other entities which are not hierarchical.

The values of the numerical attributes of the entities in a model are not usually independent. Changing one may result in modifying others. If one entity is related to another then by adjusting an attribute in one entity may have the effect of modifying an attribute of the other entity. The purpose of the analysis schemata is to provide the system with a description of these interactions. Each analysis program represents the procedural means to compute a numerical attribute of the same or other entities. The left side of the schema describes the applicability conditions of the program. The schema and its corresponding algorithm are

invoked when these conditions are satisfied by the model produced so far by the system. This allows for a procedural representation in the system.

The constraints describe the requirements that must be satisfied in the final model. The left side of a constraint describes the conditions under which it must be activated. Its right side represents the restriction applied by the constraint. For example a rule may state "the losses in a transformer must be less than MAXWATTS watts". The left hand side describes the item being a transformer, while the right hand side gives the restriction on the transformer that its losses must be less than MAXWATTS watts.

In the course of solving a problem, the expert system must often choose among several alternatives that may or may not lead to a solution. An exhaustive search would be time consuming, thus a good heuristic is required to focus the search. A piece of advice is a user's judgemental piece of knowledge described in a form similar to a constraint. For example a cable with high rating should only have very thick cables connected to it. These general heuristic rules speed up the search by allowing more likely paths to be searched first. Even if the piece of advice is bad it does not prevent the system from searching for a solution.

Searching the rules

Tropic devises and executes each step of a solution process that produces a solution of the problem which is stated in the knowledge description. Tropic uses a *state space searching* paradigm where the information which may be accessed depends on the current state of the system. The system carries out a number of actions, which transforms the current model from one state into another. The solution process consists of a sequence of these actions. At points in the decision process, *decision points*, Tropic must choose among several alternatives. Often it uses the problem solving knowledge to choose the best path. To get from state to state, numerical and structural constraints must be satisfied.

An expert systems may make errors. The system determines an error when it reaches a point called a *fail safe point*, where some constraints are unachieved while no local alternative is available. In order to recover from an error, it must backtrack to a previous decision point where the solution process can be carried on with another alternative. Backtracking consists of choosing the decision point to go back to and undo the effects of the withdrawn alternative. Backtracking is a time consuming task that is used to undo the actions previously executed which cause the error. To get around this, the backtracker uses heuristics to pick out the most likely point at which an error occurred.

Top down strategy

A top-down strategy is a powerful technique of introducing new levels of detail in stages, branching down in a tree like structure. A top-down strategy also allows decision making programs to operate in more and more detailed abstractions defined by the description of the knowledge, and use the final state of the solution attained in one abstraction as the starting state in the next abstraction. Each successive stage can be tackled by the decision making programs. The system maintains a precise trace of the sequences executed so far. At each stage of the top down strategy, the currently evoked domain knowledge defines an abstraction of the solution space.

The new state attained at each stage of the top down strategy is used to update the set of evoked knowledge to be considered at the next stage. Thus as lower levels of the hierarchy are reached more and more detail is determined about the problem. The final state of the solution attained becomes the new state in the top down strategy. The top down strategy decomposes the problem into simpler problems which can be tackled more easily. Powers[16] was quoted earlier in the chapter as suggesting the use of a decomposition representation. This type of representation reduces the problem into a set of simpler problems which may easily be solved.

In order to determine the outcome of a rule, the Tropic system applies a top down strategy, evokes the knowledge, achieves constraints (structural and numeric) and recovers from errors. For more detail on a top down strategy, refer to Latombe[15].

An Expert system in a pipe-work management system

In Newell's[17] pipe management system, the CAD system comprises three components: a *computer aided drawing system*, a *rule base* and an *inference engine*. The rule base contains the rules of the allowable relationships between entities within the application. The rules are stored in a data base. The inference engine provides a mechanism for looking up rules and returning the results obtained from the rules.

The system contains a *language processor*, *action routines*, *database routines*, and *graphical output routines*. The language processor permits the complex flow of control of the drawing program to be described separately from the program itself. The language processor deciphers and checks for legal user syntax. It controls the system by making calls to the action routines, the database routines, and the graphical output routines.

The application contains a set of action routines that are called by the language processor. The action routines carry out some predefined operation and communicate with one another by sharing common areas of data.

The database and graphical output routines are general purpose while the action routines are application specific. The graphical output routines draw the entities onto the screen. The database routines retrieve information from the data base. The database interface isolates the action routines from the detailed mechanisms of the storage of information. The action routine passes the attributes of the elements to the database routine. The database routine, having a list of allowable attribute values, checks if the request is invalid.

Further work in artificial intelligence, graphics and databases

Stevens[21] describes a sophisticated computer aided instruction program which makes extensive use of graphics to show steam plant operation. Fein[22] describes a project to provide a graphics capability to an automatic authoring program. In dealing with re-planning Nasa space missions, Scar[23], and with configuring computer systems, McDermott[24], show the potential of using graphics with artificial intelligence. Work on graphical design leading to an automatic redimensioning of parts by Goss[25] leads from graphics to artificial intelligence.

Neal[26] reviews recent work integrating data bases and computer graphics. Such systems are beginning to use artificial intelligence techniques to assist in graphical display of data base information. Garett and Foley[27] have developed a system to dynamically update the graphical display resulting from interactive updates to the data base.

Knowledge bases are particularly useful for providing data base protection by:

- Enforcing data base integrity constraints.
- Detecting when a critical piece of information is missing and no sensible default can be assumed.
- Answering the designers question "what next?"

The major difference between a data base and a knowledge base is that a data base contains facts and relations whereas a knowledge base contains facts that are organised towards a solution. The knowledge base contains a representation about what it should contain and how to go about acquiring these facts.

Conclusion

This chapter has identified important concepts for an intelligent CAD system. A good knowledge representation is crucial to the effectiveness of the CAD system. The knowledge base needs to contain data which is organised towards a solution. Different types of knowledge can be represented in the knowledge base. The problem can be solved by using the different types of knowledge in the solution process.

These examples of the work of previous researchers have pioneered the way for further development in intelligent CAD systems. However there are many obstacles that need to be overcome before such systems are practically realised. The next chapter describes the author's proposal for an intelligent CAD system.

CHAPTER 6 A PROPOSAL FOR INTELLIGENT CAD

Introduction

After having analysed previous implementations of intelligent CAD systems in chapter 5, a number of important issues have been raised. This chapter develops some of the ideas mentioned in the previous chapter.

An intelligent CAD prototype is developed. The main objectives of such a prototype are outlined. A brief description is given of the architecture of such a system.

Entities have a major role in intelligent CAD systems. The chapter describes the type of data they encapsulate and the functions that these entities perform.

A major factor in getting an expert system to work is making it possible for the expert system to understand and interpret the structure of the data that it works with. Thus the data must be structured in such a way that it is easy for the expert system to manipulate it.

Objectives of the prototype

Each entity in an intelligent CAD system has a set of rules associated with it. The rules determine the constraints of the entity and the relationships it may not have with other entities.

The prototype has been constructed to solve a specific problem. The problem involves the validity of a relationship between two entities, a cable and a stand. A cable is represented by a straight line and has attributes such as type, rating or voltage. A stand is represented by a rectangle and has a type, i.e. residential or commercial. A rule describing an invalid relationship that exists between a cable and a stand is described in the knowledge base. If this rule is satisfied then the system is in an invalid state. The appropriate action must be taken by the system or the user must be informed of the troublesome area in the design. The rule states "a cable with a voltage rating greater than MAXVOLTS volts may not lie on top of a residential stand". Although this rule is extremely basic it is no trivial task for the computer to solve it. Much processing and logic is necessary. The rule is invoked either when a cable or a stand is created or manipulated within the CAD system. Once an entity is created or modified, the system checks if the rule applies to that entity.

From the prototype it should be possible to establish what the problem areas are for an intelligent CAD system. This can help others find ways of getting around these problems. The prototype looks at dealing with entities which are linked to or have a relationship with other entities. It examines when an entity should check its validity. Certain relationships to other entities need to be determined when the entity is created while others need to be determined

during the testing of a rule condition. Rules must be defined for the entities in the CAD system to ensure that the system can check their integrity.

The prototype aims to find out what and where the problem areas are in intelligent CAD, and how the system reacts to getting stuck and dealing with incomplete information. The system should be able to find out any information that is lacking.

The representation of information is extremely important. The prototype looks at how the entities in the CAD system are represented and how the rules within the knowledge base are structured so that they may easily be interpreted by the inference engine.

The prototype aims at finding a basic framework for intelligent CAD systems and the strengths and weaknesses of such a system. The prototype must be able to locate all the rules relating to an entity and verify that such an entity is in an invalid configuration by testing out each rule condition with the information represented in the CAD. It needs to compare all valid relationships between entities and to check that these attributes in the relationships are satisfied. The prototype also looks at what other technologies can be introduced in order to make such systems more effective, efficient, reliable and practical

System architecture

Intelligent CAD should be composed of independent objects which interact with one another. These objects are the *inference engine*, the *graphical interface*, the *list of shape entities in the CAD* and the *knowledge bases*. These objects should work independently of one another and carry out the necessary actions. For example an inference engine knows how to access the knowledge base and come up with the rule condition or flag that no more rules exist in the knowledge base. The main problem is finding a way to integrate these separate parts so that they can all work independently and at the same time mould together to form the intelligent CAD.

The expert system and CAD system should be completely separate and independent from one another. By changing the implementation of the expert system or the CAD system should have no effect on the other modules. The prototype should be created so that it can become a reusable and maintainable system that can be easily improved by changing only the relevant objects or deriving new objects from existing ones. The components should be able to interact with one another and extract meaningful information from each other. For example, information such as a particular type of relationship between the current entity and other entities in the CAD may be determined.

The CAD system and expert system components in the intelligent CAD system are not very different from a conventional CAD and expert system. The only difference is that the intelligent CAD system allows these components to interact and exchange meaningful

information with one another.

Importance of data representation

Data representation is extremely important because both the CAD and the expert system need to be able to understand and manipulate the information that they are presented. Both systems need access to one another's data and so may be integrated to interact with one another. Knowledge representation is crucial to the effectiveness of an intelligent CAD system. This has been particularly emphasised by Powers[18] and Latombe[16] in chapter 5. The manner in which knowledge is represented determines the extent to which it is communicated and understood. According to Fox[19], a good representation scheme reduces the amount of replication of information, enhances the shareability of the data, and allows basic concepts to be represented in an unambiguous manner.

Knowledge representation concerns *syntax* and *semantics*. Syntax involves structuring the rules in a form which conveys meaning. Semantics involves describing relationship between knowledge to other knowledge (Lucas[15]). The semantics utilise an *is_a* link to define taxonomic relations of knowledge to other knowledge. Knowledge may be represented in a *hierarchical network* forming a *semantic network*. Another approach to representing links between knowledge is by use of *frames* as mentioned in chapter 3. A frame partitions a semantic network into easily identifiable concepts. The semantics of the representation determine the generality of the knowledge.

According to Goodman[2], the choice of a good primitive which is conceptually rich and able to span the set of concepts germane to the application, allows the construction of a shareable knowledge representation. *Generality*, *accessibility* and *extensibility* are also necessary. For generality, the representation should support more than one function such as activity planning, scheduling, question answering, simulation and graphics. For accessibility the representation should be easily perused and altered and easily and unambiguously understood by the user. For extensibility the representation system should easily adapt to representing and organising information in new ways. Ultimately the level of intelligence a system will display may be limited by the representation.

Goodman[2] states that reasoning occurs at many levels of abstraction. The complexity of a problem can be resolved by working at the appropriate level of detail.

The basic unit for representing entities, processes, and ideas etc, is the *schema*, which consists of a schema name and slots associated with it. The schema encapsulates information about the concept. The schema is related to other schemata in the knowledge base.

Knowledge representation

As has been mentioned in chapter 4, the knowledge representation determines how well the expert system is to work and whether it is to work at all. This section describes how knowledge should be represented in order to enable the expert system to understand and work with this information.

As has been mentioned in chapter 5, intelligent CAD must be able to represent explicit and implicit relationships. Explicit relationships between objects are self evident such as a cable is connected to a transformer. Implicit relationships between objects need to be postulated, such as: all the entities enclosed in a stand entity belong to the stand entity. The CAD system needs to be able to handle both of these representation schemes and determine both explicit and implicit relationships.

The knowledge base must be flexible. It must be structured in such a way that the inference engine can [illegible] and infer additional rules. One of the aims of Latombe in his [illegible] and Sargeron[18] in his Xgraph system, is to maintain the integrity and [illegible] represented in the knowledge base. Encarnacao[19] suggests that the rule base should have a self checking mechanism whereby the validity and consistency of [illegible] into the rule base is checked. Rules should be easily and unambiguously [illegible]. The knowledge base should be structured in such a way that new rules can be [illegible] and even dynamically at run time. At the same time the system should validate the consistency of the data as the rules are appended into the rule base. Rules which have given an ambiguity must be highlighted for the users attention. If the result of a rule is [illegible] or inconsistent, this should be recorded in a log. Thus the user may be informed of the rules that are ambiguous or unreliable in the system. A useful feature is the ability to add new rules while the system is running. This allows knowledge to be entered by a domain expert.

The inference engine must be designed so that rules can be read in and additional rules can be inferred from existing rules. It should be able to ascertain if a rule is successful or if it is in an indeterminate state never to be evaluated. The expert system should be able to respond suitably regardless of the outcome of the rule evaluation.

The intelligence of the system is captured in the rule base. It should be carefully structured so that rules can easily be added on or modified. When rules are added into the rule base their validity should be checked with the existing rules in the rule base. A duplicate rule should not exist in the knowledge base and the newly appended rule should not cause an ambiguity or conflict with an existing rule. The added rule should not cause the same rule condition to give conflicting results.

According to Latombe's[16] Tropic system it is worthwhile to record the decisions taken by the expert system so that the logic that has been implemented can be followed.

The inference engine should also contain a description facility, so that an explanation can be displayed on how a particular result was achieved from the information in the rule base. The flow of logic can be followed and understood for a particular result. Thus, once a series of rules have fired, the user should be able to go through a report and follow the reasoning of the inference engine.

Rules involve the relationships between entities which have attributes having specific values. Rules in the prototype should only deal with entities which have a one to one relationship with one another. For example, the system is unable to handle a rule of six or more transformers not being allowed to be connected to a power outlet of a certain type (while five or less transformers may). This type of one to many relationship adds much complexity to the evaluation of rules and is beyond the scope of the prototype.

The rule base is structured in such a way that relevant information may be easily and efficiently [illegible] recommends that the search space be made as small as possible by [illegible] which is relevant to the problem being analysed. The rules describing [illegible] of the entities must be arranged so that only the relevant rules relating to the [illegible] are easily matched.

Object oriented paradigm

The object oriented approach is a powerful technique for structuring information. *Classes* are defined to represent tangible items as well as ideas or concepts. Classes contain *data* relating to them and *functions* which perform operations on their data. Classes offer a high degree of abstraction. An instance of a class is called an *object.*

Three main properties characterise an object oriented philosophy (Booch[1]): *encapsulation, inheritance* and *polymorphism.* Encapsulation encompasses unique data and functions. The functions manipulate the data in the class. Thus a high degree of abstraction is possible making the system highly maintainable and reusable. Being able to work at varying levels of abstraction makes the object oriented approach very attractive.

Inheritance involves deriving child classes from parent classes. The child classes inherit the data and functions of its parent. Over and above this, derived classes are more specialised than parents because they contain additional data members and functions.

As has been mentioned in the discussion of the Tropic system in chapter 5, it is advantageous to decompose the structure of information so that the computer can work at the required level of detail. Inheritance enables the data to be represented as an abstraction at various levels. The attributes are stored at different levels in the inheritance hierarchy. Entities should be able to return specific values of their attributes even if the attributes are at different levels in the inheritance tree. For example the cable class is inherited from the line class which is

inherited from the shape class. The shape class contains the details of the points and lines of the shape, while the cable class contains the attributes of the cable such as voltage, current etc. Thus when an instance of the cable class is asked to return the number of points (the number of points in the cable), the cable looks within the cable object for this attribute. If this attribute is not found, the cable looks for this attribute value in its parent class, class line. If this attribute is not found in the line class, it is searched for in the parent class of the line class, i.e. the shape class. The shape class which contains the number of points attribute returns this value to the cable class which returns this value to the class object requesting this information. Thus the requesting object is capable of accessing information in any level of its hierarchy, which represents information at a different level of detail.

Polymorphism permits the assignment of one name or symbol to an action that is shared up and down the class hierarchy, with each class in the hierarchy implementing the action in a way appropriate to itself. Polymorphism is used to allow objects to manipulate themselves either explicitly or dynamically. Regardless of the entity that is being manipulated, the entity object should be able to apply the manipulation meaningfully to itself.

By using the above mentioned object oriented facilities in combination, classes can be developed in such a way that they can take care of themselves independently of the other parts of the program. This is desirable in the CART system. Object oriented systems are highly maintainable systems therefore it is worthwhile to introduce such a technology in the CART system, which is expected to change much over time.

Message passing

Message passing is the facility that allows objects to interact with one another while at the same time being completely unaware and independent of each other. It is the means of *communication between objects*. In Hirakawa's[4] iconic programming, objects instances modify other objects by sending messages to them to change and update their dimensions. Hirakawa's[4] message passing involves functions and data. If the class instance recognises the messages it receives, the relevant function of the class instance is executed. If the messages are not found then the search continues in the other levels of the inheritance hierarchy. An icon may accept only certain types of messages (acceptable messages). Icons may transfer a message from one icon to another. These are called transmittable messages. When two objects are connected, one is active and the other is passive. Acceptable messages are compared to transmittable messages in the object where the function icon belongs. If common, the relevant program module is implemented. If no message is found, the search continues further along the class hierarchy.

For example in the CART system, one object should be able to retrieve a specific attribute from another object. A message is passed to the object which returns a specific data member value. The object that has been passed the message knows how to find the attribute being

requested, even it exists at a higher level in the inheritance hierarchy.

Shape entities

Bart[6] recommends that when the graphical instance is created, the application supplies a key to identify the application associated with the graphical instance. This approach has been adopted in the CAD prototype. For the CAD prototype, when an entity is to be created, a key which is the entity number is associated with the created entity. That entity can be indexed and so accessed and manipulated by referencing on the entity number.

Bart[6] states that to create an instance of the entity, components are recursively created and composite links are filled in. An entity may be linked to other entities in such a way that by modifying an attribute of one of the other entities changes the value of the current entity. An entity may be made up of a combination of other entities, forming a *composite* entity. When a composite entity whose attributes have dependency links to the attributes of the other entities is created, the dependency lattice is traversed to compute the attributes which need to be displayed. The entity is displayed. To delete an instance of an entity, the inverse process is carried out. Entities are erased from the screen and dependency links are severed. In the Tropic system entities may be linked to other entities hierarchically or associatively. When updating a particular part of the entity, the entities which are linked to that entity are also updated and refreshed.

When an entity is created in the prototype, the system checks if certain types of relationships exist with any of the other entities in the CAD system. Should such a relationship exist between two entities, a description of the relationship and the index to the other entity in the relationship is recorded for each entity.

The existence of relationships between entities may also be determined when the validity of a rule is being checked. Often in the validation of a rule, there may exist incomplete information about a certain type of relationship between entities. In order to determine if these relationships exist, it is necessary whilst testing the current entity, to check whether it has such relationships to the other entities in the CAD.

When creating a graphical entity, the composite structure of the entity and dependencies to other entities must be defined. Thus when an entity is drawn, dependency relationships to other entities are recorded. When the entity is deleted, these dependency links are removed. In the CAD prototype, when an entity is created, the relationship links to other entities are filled in and when an entity is deleted or moved the relationship between the two entities no longer exists and so the relationship links are removed.

Extensions or changes to the attributes of an entity are made by the methods. The entity is modified by sending "change entity" messages to the entities. The entities contain instructions

for updating their own dimensions.

Conclusion

Data representation is crucial to the effectiveness of the system. Entities need to be structured in such a way that they can meaningfully extract useful information from one another. The designer needs to work with data familiar to it so that it too can manipulate and extract meaningful information.

The object oriented approach offers powerful and flexible features which make it appropriate for intelligent CAD. Complex objects can be created by building inheritance tree hierarchies.

By using the facilities of object oriented programming, information can be encapsulated in objects. This makes the system more maintainable because problems are located to specific objects enclosed within the hierarchy of the object. This also ensures that the system will be more maintainable and reusable than conventionally structured programs.

In the object oriented paradigm, each class object encapsulates a concept. By abstracting concepts within the classes, each class is only concerned with the concept that it represents. It does not need to concern itself about matters which it does not understand.

The intelligent CAD system is made up of separate components which work together to achieve a required level of intelligence.

The entities in the CAD need to work at the level of detail depending on the types of rules that are being evaluated. They need to access rules in the rule base to check their integrity. The manner that this it to be achieved is discussed in the next chapter.

CHAPTER 7 IMPLEMENTATION OF THE PROTOTYPE

Introduction

A system which contained a small CAD system and an expert system was written. The C++ programming language was used. Object oriented features were implemented to maximise re-usability. The prototype was developed to test out a number of concepts in order to determine the requirements of an intelligent CAD system. Once a basic intelligent CAD system has been developed, more sophisticated systems can evolve from these.

The prototype attempts to integrate an expert system and a CAD system so that they co-operate to assist the user in the design process. The functionality of the CAD is limited, only being able to perform simple operations such as drawing shapes and storing these shapes in a list. The basic expert system is only able to solve simple rules. The prototype is constructed to pin-point weaknesses in intelligent CAD systems and to highlight concepts which should be incorporated in such systems. By careful determination of the infrastructure, more complex and powerful intelligent CAD systems can be developed.

The prototype consists of four separate but integrated modules: a *graphical user interface*, a *list of shape entities*, a *knowledge base* and an *inference engine*. These separate components interact with one another to form the intelligent system.

Useful ideas have been extracted from the literature to achieve an effective intelligent CAD system. Some additional ideas have been introduced to tie up the loose ends. The project report shows a software architecture which allows the main parts of the system to be combined into an intelligent CAD system.

This chapter explains how the attributes of an entity are determined when the entity is created, manipulated or destroyed. The entity does not exist in isolation and so it must contain the information indicating the links and relationships that it has with other entities. The chapter describes the different types of entities that are used in the prototype and the various attributes that they have.

Mention is made of the importance of entities needing to be aware of the existence of other entities. A class, called the *picture class*, has been created to iterate through all the entities in the CAD. The characteristics of the class and the motivations for creating such a class are discussed. This picture class makes particular use of polymorphism, encapsulation and uses other object-oriented features.

Shape class

The shape class is derived from the gview class. A shape entity has all the characteristics of a view, i.e. it has left, right, top and bottom extremes. It is bounded by a rectangular area. All geometric objects which are represented on the CAD are derived from the shape class and they inherit the attributes and member functions of the shape class.

Shape Entities contain lines and points which describe the dimension of the entity. Shape Entities can create, destroy and modify themselves. Shape entities draw themselves by individually drawing all the lines and points making up the shape. Should the dimensions of an entity be changed in any way, the dimensions of the lines and points making up the shape are also updated. An entity carries out an operation upon itself after receiving a message that is passed to it. The type of operation it performs depends on the message it receives.

A shape entity contains a relationship slot. This data member describes the type of relationship that it has with another shape entity and provides an index to the other shape in the relationship. Each shape in the CAD system has a unique index number.

A shape entity knows how to take care only of itself and is unable to manipulate any other shape entities in the CAD. An entity in the CAD is aware of all its own characteristics and of the existence of relationships to other entities.

Triangle, rectangle and line classes are derived from the shape class. These subclasses are derived further to become transformers, stands and cables respectively. The subclasses inherit all the characteristics of their super-classes. However they do contain additional items in the form of data members and member functions. They are specialisations of their super-classes. The Booch diagram in figure 5 shows the class structure of these classes. A triangle represents a transformer, a rectangle represents a stand and a curve represents a cable.

Cables have a diameter and are made of a certain material, have a maximum rating and have a certain cost/metre associated with them. *Transformers* have a number of outlets, a maximum rating per outlet and have a cost associated with them. *Stands* have dimensions and may be of a certain type, e.g residential or commercial.

Once a shape entity is drawn, certain attributes are automatically determined such as type of entity or rating. Other attributes are later determined when requested by the rule base or the user. The entity itself cannot determine if it related to other shape entities. It can check the state of its relationship slots, which describe any relationships that it may have with other entities. An entity can set and return these values of a relationship.

Picture class

Shape entities are completely unaware that there is a world of other shapes around them. Booch[1] states that a class represents abstractions of objects or concepts in the real world. These objects only know how to take care of themselves and deal with messages which are passed to them. Objects are insulated from the world around them since they may only operate on the information that they encapsulate. All that an entity needs to know is that it exists somewhere in space and time and has a certain set of attribute values. Since a shape entity is completely unaware that it is related to other entities it is necessary to fill in the relationship slot of the shape entity to tell it that it does indeed have a relationship with another entity. The shape entity attaches no meaning to this relationship slot and performs no operations on this slot but may only set and return the relationship slot values.

Shape entities have relationships to other entities. For example an entity may be on top of, be within 10 meters of, be inside another entity. Once an entity is drawn, the explicit relationships to other entities in the CAD are determined. The relationship is filled into the shape entity's relationship slot which tells which shape entity it has a relationship with and the description of the relationship.

Some relationships are created when the entity is created (*explicit relationships*) while other relationships are determined when requested by a rule or the user (*implicit relationships*). For example an on top of relationship might be determined when an entity has just been created (with the newly created entity lying on top of the other entity). Alternatively a relationship may need to be dynamically determined when the rule base requests to find all entities that are within a certain distance of the current entity. This would not have previously been determined and would need to be evaluated.

How is it determined that an entity is related to another entity? The picture object determines the existence of relationships between shape entities. It keeps a list of all entities in the CAD (See figure 6). It contains a set of member functions that determine whether a particular entity has a relationship to any other entities in the CAD system.

The picture object is really a specialised container class. It can iterate through a list of shape entities and compare the attributes of the different entities with one another. For example to determine if one entity overlaps with another entity, the picture object iterates through the list of shape entities and compares each line of each entity with those of the current entity. It then determines for each line of each entity in the list of shape entities, whether it intersects with the current entity. If the lines of two shape entities intersect then obviously one shape lies above or below another. There are two techniques that can be used to determine which entity lies on top of the other entity. The entity that has been created or moved more recently is obviously on top of the other entity. Alternatively, when an entity is created or manipulated, it might be desirable to date and time stamp the entity. Thus the entity that was created earlier would lie below the newly created entity and visa versa.

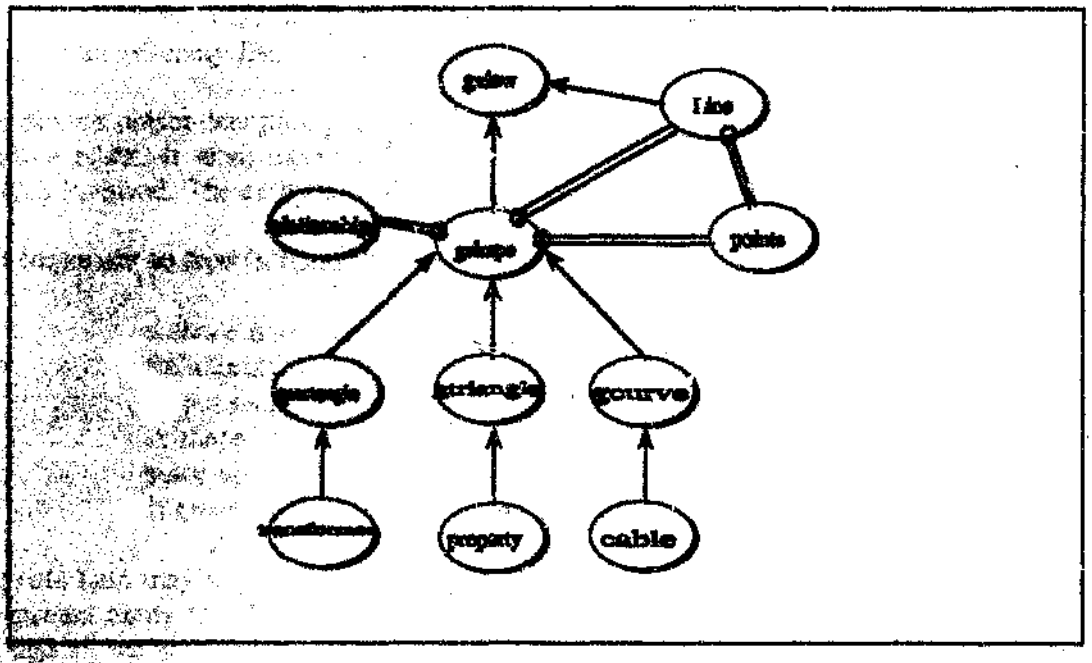


Figure 5 The Shape Objects Within the CAD

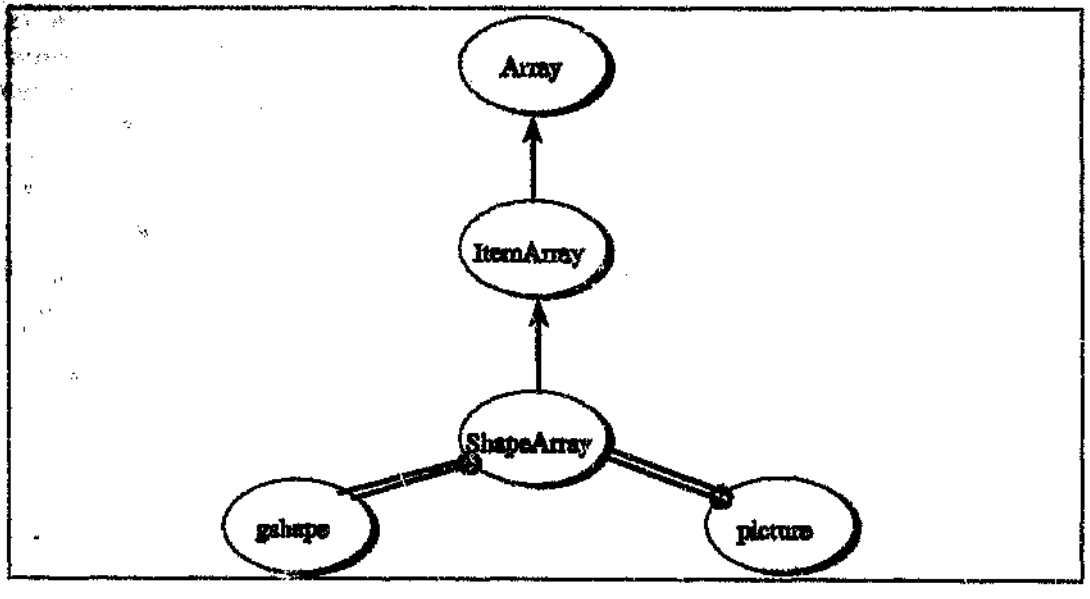


Figure 6 The Picture Object

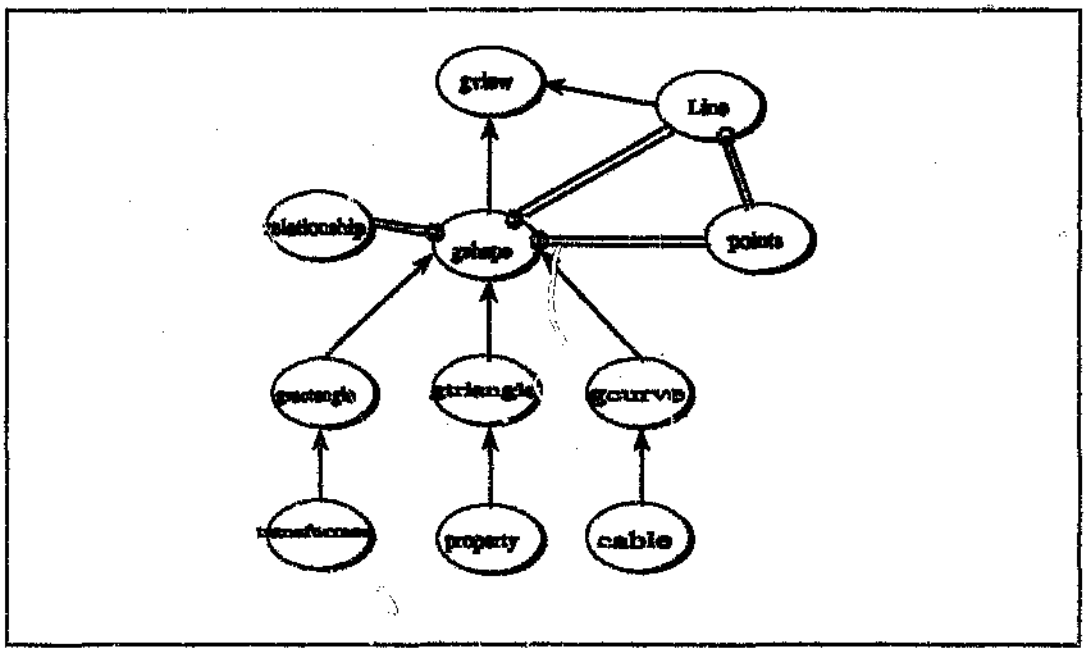


Figure 5 The Shape Objects Within the CAD

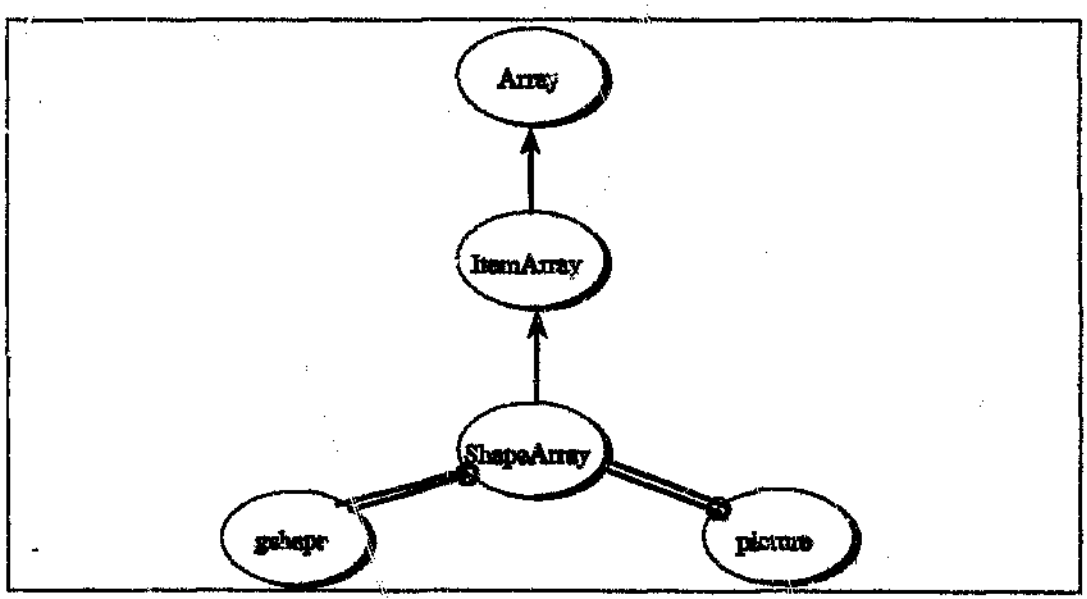


Figure 6 The Picture Object

Once an entity is drawn and the entity has a relationship with another entity, the relationship slots of both entities are filled in. When a shape entity has been created, it is stored in the picture's shape entity list.

The picture object determines inter entity relationships such as on top of, within a certain distance of etc. It sends messages (i.e. the values of the relationship slots) to the shape entities involved. The entity uses this information to set the values of its relationship slots.

It is important to bear in mind that:

- a shape may have multiple relationships with other shapes.
- Relationships may be dynamically added and removed during the manipulation of the shape objects within the CAD.
- A shape entity may not know of all the relationships it has with other shapes, since not all the different kinds of relationships are determined when the shape is created.

The rule base may require the picture object to find a particular type of relationship between the current entity and the other entities in the CAD. The picture object is then used to iterate through all the entities in the CAD and find which entities satisfy the relationship requested by the rule base.

When an entity is modified in any way it is necessary for the picture class to determine if all of its relationships of the entity are still valid. If a relationship is no longer valid for an entity, the relationship slot is removed. For example an entity may have the relationship that it lies on top of another entity but when it is moved, it no longer satisfies this relationship. This means that whenever an existing shape is modified, it is necessary to re-evaluate all the relationships that this shape has with other shapes.

In order to call the functions of the entity at run time, polymorphism is used. Depending on the particular shape entity that is being manipulated, the program dynamically calls the relevant member function to manipulate the entity. Thus even though a number of different shapes may have a member function all with the same name, the correct member function belonging to the shape entity is called. The picture object can select a specific entity in the list of shape entities and by means of polymorphism can call any member function relating to that entity.

Knowledge base

In the prototype three sub knowledge bases make up the rule base of the expert system: The *object knowledge base*, the *relationship knowledge base* and the *attribute knowledge base*. The inheritance hierarchy of these knowledge bases is described.

The knowledge base contains a collection of rules which describes relationships which must or must not exist between entities. For example a typical rule that exists, states: "a high voltage cable may not pass over a residential stand". This rule must be described in such a way that the inference engine can easily interpret and check that the entities in the CAD system do not satisfy this rule. The inference engine must be able to locate all the rules which relate to a particular entity being tested. Thus if cable rules need to be found, all the rules relating to a cable must be indexed and easily retrieved.

The knowledge base contains expertise, in a particular field, which is captured in a rule like format. Figure 7 shows the inheritance hierarchy of the knowledge base class. The FiLookUp object is at the highest level of the hierarchy and is the base class for the RuleList object. The *FiLookup* object can read a record from a file.

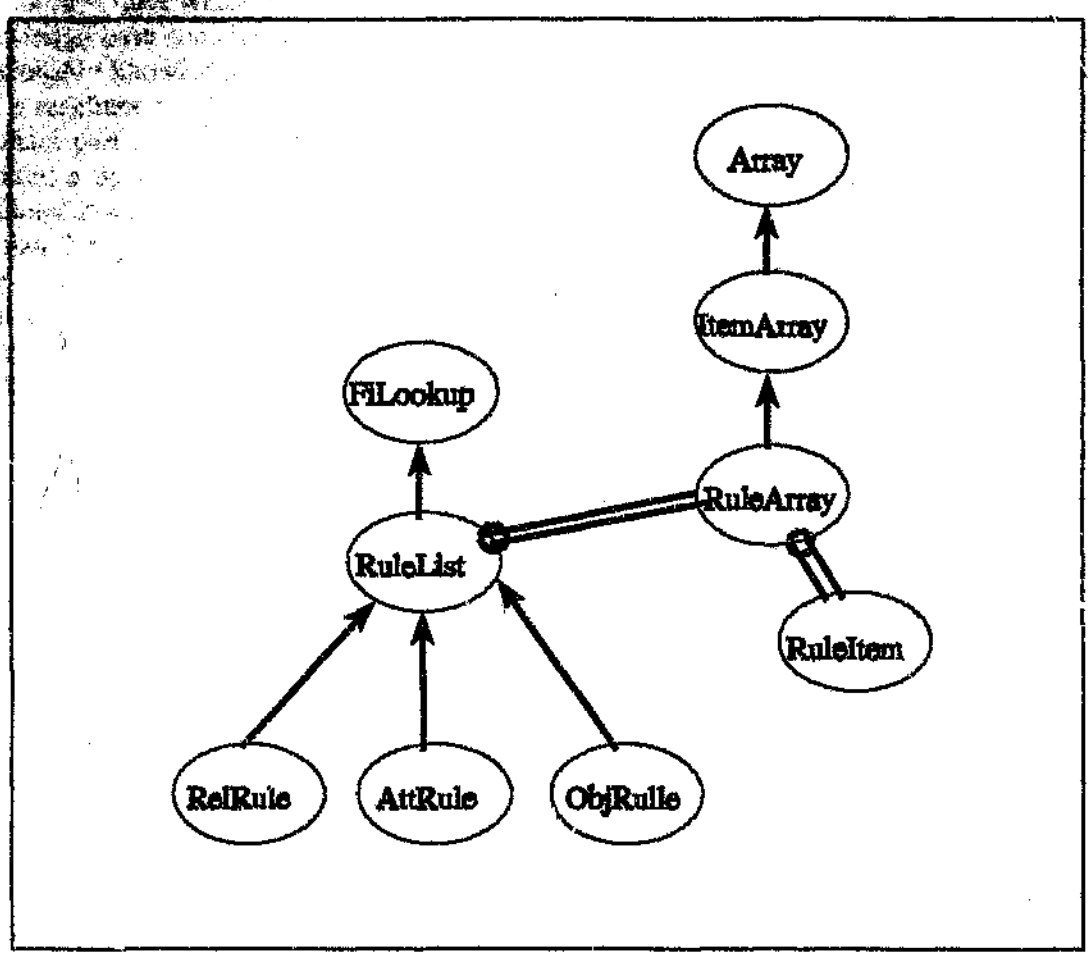


Figure 7 The Inheritance Hierarchy of the Knowledge Bases

The parent class for the knowledge bases is the *RuleList* class. The RuleList can search for, read and find a rule from a knowledge base. It is a specialisation of the FiLookUp class, but it reads a file having the format of a knowledge base file.

The RuleList class is specialised into an object knowledge base class, a relationship knowledge base class and an attribute knowledge base class. Each knowledge base, except for the attribute knowledge base, refer to rules found in one of the other knowledge bases (figure 8 represents how the rules found in one knowledge base relate to those found in another knowledge base).

The object knowledge base contains a list of relationship rules which are found in the relationship knowledge base. The object knowledge base is a dictionary associating a set of relationship rules which pertain to a specific entity in the CAD. In figure 8 the object knowledge base contains an object rule which provides a reference to a rule in the relationship knowledge base, i.e. rule number 6. Each relationship rule has a conditional part and a resultant part. Should the conditional part of the relationship rule be satisfied, then the resultant part of that rule is activated. The resultant part of the relationship knowledge base contains a list of rules which are found in the attribute knowledge base. If in figure 8, relationship rule number 6 is satisfied, then the relationship rule refers to attribute rule number 7, found in the attribute knowledge base. Each rule found in the attribute knowledge base contains a conditional and a resultant part. If the conditional part is satisfied, then the attribute rule is satisfied. In the case of figure 8, if relationship rule number 6 is satisfied and if attribute rule number 7 is satisfied, then the object rule is satisfied and an invalid configuration exists in the CAD.

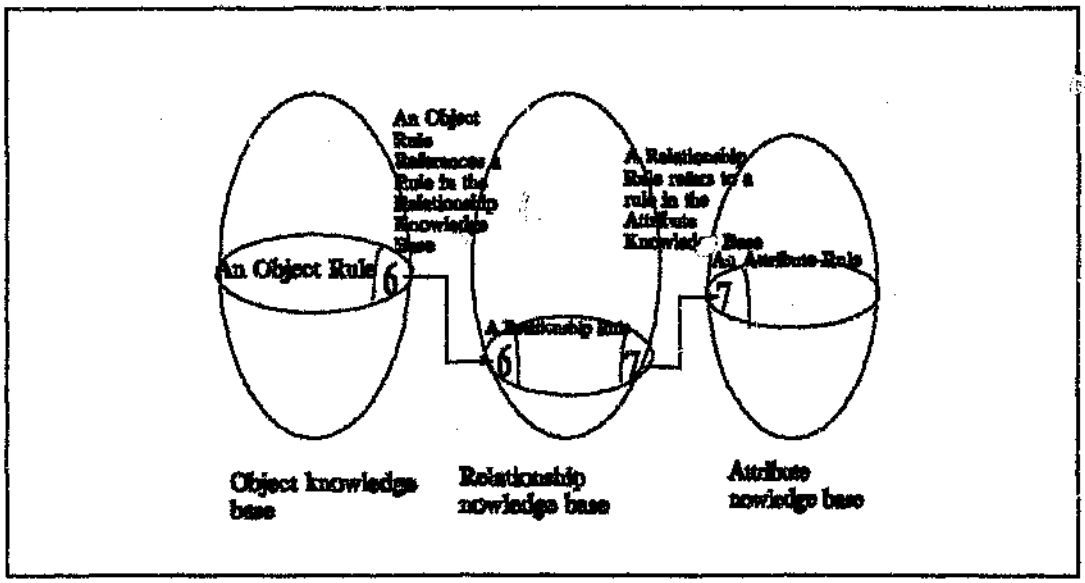


Figure 8 Indexing Rules From the Knowledge Bases

The knowledge base objects do not interact directly with one another in any way. They obtain information from their own knowledge base and keep track of which is the current rule that needs to be checked in the other knowledge base that they relate to. They read in a record and store the relevant data from the knowledge base in their class's data members.

In the Tropic system, a rule may be structured over three levels in a tree like graph. The highest level contains the description of the type of entity and the structural and numeric limitations of the values of the fields of these entities. The next level describes the hierarchical representation of the entity. The lowest level describes inter relationship links between entities. A similar approach has been used in the prototype. However the hierarchy of the entity does not need to be described in a knowledge base because it is explicitly described from its inheritance hierarchy from its parent classes.

The object knowledge base, contains a list of all the entities that are in the CAD. Each entity in the CAD has a list of relationship rules associated with it. The relationship rules are grouped together by the AND and OR logical operators. The AND logical operator means that if a rule is satisfied, the next rule must be checked. However if a rule is found to be invalid, then the AND condition need not be evaluated because the result is false. The OR condition indicates that if any one of the rules is satisfied, then the rule is successful. A parser, as mentioned in chapter 3, is necessary to check the validity of the combination of AND and OR operators. A parser was not developed in the prototype since it would complicate the basic inference engine.

Structure of the knowledge base

The structure of the knowledge base greatly influences how the rules are to be solved. Here the expert system is used to determine if undesirable relationships exists between entities.

Once an entity is manipulated or created in the CAD, it is the purpose of the inference engine to find all the invalid types of relationships that exist between the entity that has just been created (current entity) and other entities in the CAD system.

The following are some examples of object rules represented in the object knowledge base:

Object Name	Relationship Rule No.
cable	17
stand	11&12&101&5

The labels "cable" and "stand" indicate the types of entities that the rules apply to. For example "cable" has a single relationship rule, i.e. rule number 17, while "stand" has a number of relationship rules associated with it. For the stand rule to be satisfied all the relationship rules, i.e. the relationship rules 11,12,101,5 must be satisfied since they all have the logical AND (&) condition associated with them.

Each entity in the object rule base has a list of rules [illegible]ted with it. This list of rules refers to rules that exist in the relationship knowledge base. The relevant relationship rule is referenced in the relationship knowledge base by matching on its rule number. The following are examples of some relationship rules represented in the relationship knowledge base:

Rule	Object1	Relationship	Object2	Attribute Rules
17	cable	ontop	stand	8&22&44
101	stand	ontop	cable	44&33

The number in the rule column is the number of the relationship rule, i.e. 17 or 101. This is the relationship rule referred to from within the object knowledge base. This rule states that a cable entity may not lie on top of a stand entity.

Object1 column refers to the name of the first entity to which the relationship refers, i.e. cable or stand. Relationship refers to the inter relationships between the two entities. Object2 refers to the inter related object, i.e. the stand or cable. The attribute rules refers to the rules that are referenced in the attribute rule base. For this relationship rule to be satisfied, the logical OR and AND operators applied to the attribute rules must also be satisfied.

If a first entity satisfies the given relationship with another entity, then that other entity number is added to the list of entity numbers that satisfy the current relationship. This list of numbers is called the *conflict* set and contains all the entities in the CAD which, so far, satisfy the relationship with the entity currently being checked.

The relationship knowledge base reads in a relationship rule and stores the relevant values into the fields of the relationship object. Each relationship rule has a set of attribute rules associated with it. The relationship knowledge base object contains an index to the current attribute rule which needs to be checked.

For a relationship to be satisfied, each attribute rule needs to be evaluated. The rules in the attribute knowledge base are represented as follows:

Rule	Object Type	Attribute	Operator	Value of Attribute
8	cable	current	>	400
22	cable	voltage	<	400
44	stand	type	=	commercial

The Rule column heading refers to the number of the attribute rule. The attribute rule is indexed from the relationship knowledge base. The object type column refers to the type of entity that the attribute rule refers to. The Attribute column identifies the field that is being tested in the object, such as current, voltage and type. The Operator column refers to the comparison operator on the attribute value of the object. The value of the attribute refers to the value to be used in the test. Attribute rule 8 is satisfied if the entity type is a cable and if the current attribute is greater than 400 units.

It must be borne in mind that the complete list of entities that are candidates for satisfying a relationship are stored in the conflict set. All the entities in the conflict set which are of the object type given in the attribute rule are evaluated. If their attribute value does not satisfy the rule, then their entity numbers are removed from the conflict set.

The relationship is made up of a combination of attribute rules which are ANDed and ORed together. The relationship rule is satisfied when the combination of these attribute rules is satisfied. The next relationship rule indexed from the object knowledge base may then be evaluated. If the conflict set, which contains the list of shapes satisfying a relationship is empty, then the relationship rule has failed.

The attribute knowledge base is passed a message containing the attribute rule to read from its knowledge base. The attribute rule contains a condition which a field in the entity being validated must satisfy. The attribute knowledge base reads a record from its knowledge base and retrieves the values of the attributes into the attribute object. The attribute rule contains

the allowable thresh-hold values that the attributes of an entity may have.

Each knowledge base object contains data members which store the fields of each knowledge base record. These data members store all the information which is relevant to the current rule being evaluated. They contain the conditional and resultant parts of the rule, the rule number and any supplementary information which is relevant to the rule (such as an explanation of the rule, the time the rule was created, who created the rule etc).

Inference engine

The inference engine is the navigator through the rule base. It determines which rules are to be looked up in the rule base, what action is to be taken if a rule fires successfully or fails. Additional information is obtained from the rule base thus determining if further rules need to be looked up.

The inference engine needs to understand the syntax of the rule base so that it can interpret the meaning of the rules. The list of all the necessary rules that need to be looked up is stored in the separate object, relationship and attribute objects. The inference engine determines which type of rule it is to look up next. It contains a scheduler so that it can determine the priority of rules, i.e which rule in which knowledge base is to fire next. A relationship rule is first checked, followed by each attribute rule belonging to the relationship rule. This process continues until the rule fails or all the relationship rules have been tested.

The purpose of the inference engine is to provide the mechanism for scanning the rules. The validity of the rules are tested, additional rules are searched, and additional rules are inferred. The inference engine controls the look up of the rules and determines the next type of rule that is to be looked up, regardless of whether it is an object rule, a relationship rule or an attribute rule. The method of searching for the rules is briefly explained.

Figure 9 shows the control structure for determining the outcome of a rule. The inference engine first locates the entity to be tested from the object knowledge base, looks this up and so gets a list of the relationship rules for this entity. The object knowledge base indexes the first relationship rule to be checked. If the entity is not found in the object knowledge base then the rule checking process ends. The first relationship rule is looked up in the relationship knowledge base. If the relationship rule fails then the next relationship rule number is fetched from the object knowledge base and is looked up in the relationship knowledge base. If no more relationship rules exist in the relationship knowledge base, then no rules have fired successfully and the process ends.

If the relationship rule is satisfied, a list of all the entities that satisfy the relationship are placed in a list, called a *conflict set*, containing the shape index number of each shape entity that satisfies the relationship. All the attribute rules relating to the relationship are in turn

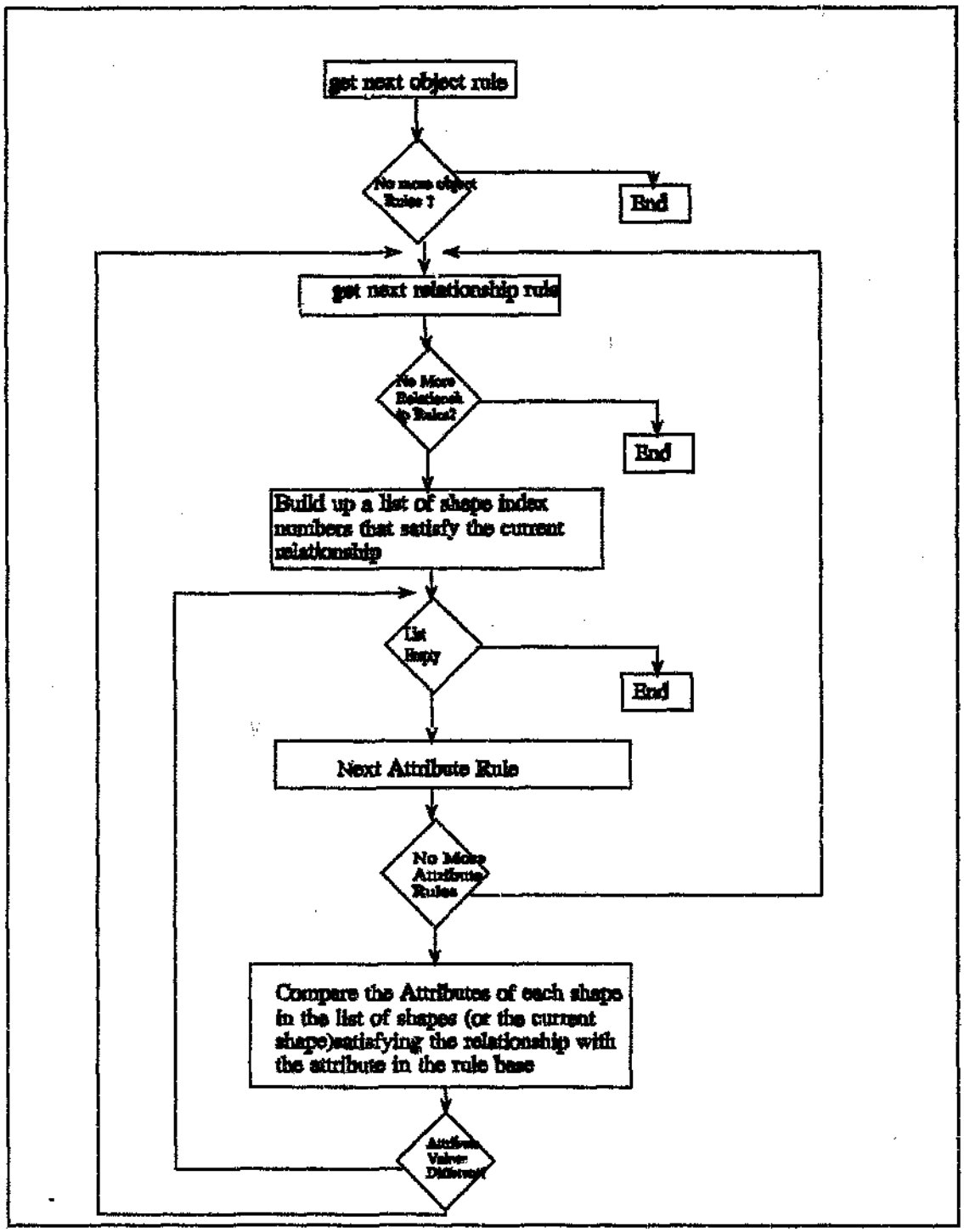


Figure 9 The Control Structure for Determining the Outcome of a Rule

looked up. If no more attribute rules are found and there are still shape index numbers in the conflict set, then the relationship rule is satisfied for the shape numbers in the conflict set. This is recorded and the next relationship rule may be tested. If an attribute rule has failed and the conflict set containing all the shapes satisfying a relationship is empty then the relationship rule has failed and the next relationship rule is tested.

The object knowledge base class instance contains an index to the next relationship rule that is to be looked up. The next relationship rule is indexed, looked up and this process is repeated until no more relationship rules are found. When finished with this rule checking process, all the relationship rules and their corresponding attribute rules that were successful are recorded. Once the inference engine has finished with the process of looking up rules, it is re-initialised and ready to be called upon to evaluate another set of rules.

Structure of the shape controller class

The shape controller (ContShape) class is used to tie all the components together into an intelligent CAD. The requirement for this class is that it should allow all the different parts of the CAD system to communicate and work with one another. The shape controller class, shown in figure 10, contains an inference engine, an object, relationship and attribute rule, a list of numeric indices to shape entities which currently satisfy a relationship rule, a picture object, the current shape entity name and number that is being checked. The ContShape object allows the shape entities in the CAD system to interact with the knowledge base. Thus entities can check their validity to see whether they have caused an invalid configuration in the design.

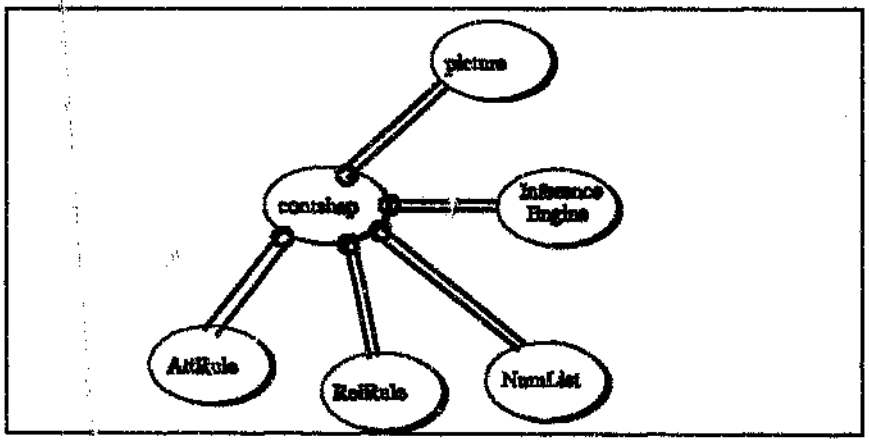


Figure 10 The Shape Controller Class

The inference engine object returns conditions which need to be evaluated. It keeps track of the next rule that has to be looked up in the object, relationship and attribute knowledge bases. The ContShape object requests the inference engine object to look up a rule in any of the knowledge bases.

In order to draw an entity in the CAD system, the user first selects a shape to be drawn. He then goes about placing the vertices in the work area window. When the last point is placed with the mouse, the entity is created and added to the list of entities that currently exist. The entity draws itself on the screen.

The ContShape object passes the name of the entity, which is involved in the relationship, to the inference engine. The inference engine looks up, in the object knowledge base, the relationship rules belonging to this entity. The first relationship rule of the entity is looked up in the relationship knowledge base. This relationship rule is passed from the inference engine object to the ContShape object. The Contshape object sends a message to the picture object to find all the shapes in the CAD which have the specified relationship with the current shape.

In determining a relationship, the ContShape object uses the picture object to iterate through the list of entities in the CAD and determines if the specified relationship exists between the current entity and any of the other entities in the CAD system. Should the picture object determine that there are entities that satisfy the relationship, these shape entity numbers are stored in the *NumList* object. The NumList object contains the conflict set of entities which satisfy a relationship.

The shape controller object checks whether the NumList object is empty. If it is empty, then no relationship rules are satisfied between the current shape being drawn and the other shapes in the CAD. The next relationship rule may be checked.

Should the ContShape object have items in the NumList object, it then goes about evaluating the attribute rules relating to the relationship rule being tested. The ContShape object requests an attribute rule from the inference engine. The expert system then returns the attribute rule that needs to be tested. The attribute values in the attribute rule base are compared to those values of the entities in the NumList list or with the current entity. The picture object iterates through the NumList list of shapes to check whether the attribute rule is satisfied for each item in NumList. If the attribute is not satisfied in the NumList object, the entity's index number is removed from NumList. The attribute values of a rule are also compared to the attributes of the current entity. This process of checking the attribute and relationship rules continues until the rule is satisfied or the rule has failed. The rule is satisfied when a relationship rule with all its attribute rules is satisfied. The entity numbers remaining in the NumList object contain the entities causing an invalid relationship with the current entity. The rule fails when the attribute rules are not satisfied for the current entity, or the NumList object is empty.

Once the result of the comparison on the entity has been completed, the ContShape object

passes a message to the inference engine object to either return the next rule from the specific knowledge base, or to tell it that the rule has failed. If the tested rule condition has failed, the inference engine resets itself.

Should the inference engine find that a relationship rule, with all its attribute rules, has been tested, it advises the ContShape object. The ContShape object informs the user of the error and of the entities causing the error. It may even indicate the necessary action to take to correct the problem. See figure 9 for the logic that the ContShape object uses to look up the different rules in the various knowledge bases.

Interaction between the graphical interface and the user

The user communicates with the application via the graphical interface. The graphical interface records the keystrokes and mouse selections. These parameters are passed as messages to the various parts of the application. The application then carries out the necessary actions to deal with these messages.

The graphical interface also presents the applications output to the user by means of written messages onto the screen. These inform the user of any error conditions, the level of progress and indicate what action he should take or advise the user.

The reason for isolating the graphical interface from the application is to simplify program maintenance. Thus when changing the graphical interface, it is not necessary to modify and update the application. Even though the graphical interface is quite simple, it can easily be enhanced without rewriting or restructuring the existing modules.

Interaction of shape entities

The entities may not interact directly with one another. They interact with one another only through the picture object. The picture object contains the entities represented within the CAD. Thus if an entity wishes to find a shape entity with which it intersects, the list of entities is iterated through to find an entity which intersects with the current shape being modified.

The picture object carries out the following functions:

- Determines if one entity is related to other entities.
- Determine the value of an attribute of an entity.
- Retrieves and sets the values of the relationship of the entity

In order to test relationship between two entities, the picture object, compares the attributes of the two entities to one another and from that determines the relationship between the two entities. For example the picture object determines if one entity lies on top of another, by checking if any two lines of the two entities being tested intersect one another. While one entity has an "on top of" relationship, the other entity has a "below" relationship. In a similar way the picture object can test the attribute values of a particular shape.

Interaction between the entities and the inference engine

The shape entities and the inference engine interact with one another indirectly through the ContShape object. The ContShape object requests the current relationship or attribute rule from the inference engine. Depending on which rule is returned from the inference engine, the ContShape object checks the validity of the rule. Figure 11 shows the interaction of the ContShape object with the inference engine and the rule bases.

If the rule returned from the inference engine is a relationship rule, the ContShape object calls upon the picture object. The picture object builds up a list of all the entity numbers which satisfy the given relationship into a conflict set (the NumList object) with the currently drawn entity in the CAD. If the rule returned from the inference engine is an attribute rule, the ContShape object calls upon the picture object to return the attribute value of each entity in the NumList object, and compare that value with the allowable value advised from the attribute rule. If the attribute rule fails then the entity number is removed from the NumList object, otherwise the entity number remains in the NumList object.

Once the ContShape class has checked the relevant rule, it requests the next relationship or attribute rule from the inference engine. In this way the process is repeated. Thus the ContShape object is the means of interaction between the inference engine and the picture object, and so the entities in the CAD system.

Interaction between the entities and the rule base

As discussed in chapter 5, According to Tibbert and Bergeron[19], the knowledge base should check the validity of the rules and return the results to an interaction handler. The interaction handler is passed a function which it then implements to deal with the invalid situation in the system. Thus it is necessary for the knowledge base to know about the different types of functions that the interaction handler can implement. This approach strongly links the knowledge base to the interaction handler, resulting in a less maintainable system. However, the idea of returning a value from the knowledge base, of an unsuccessful rule, to the interaction handler which then deals with the returned value is an attractive one.

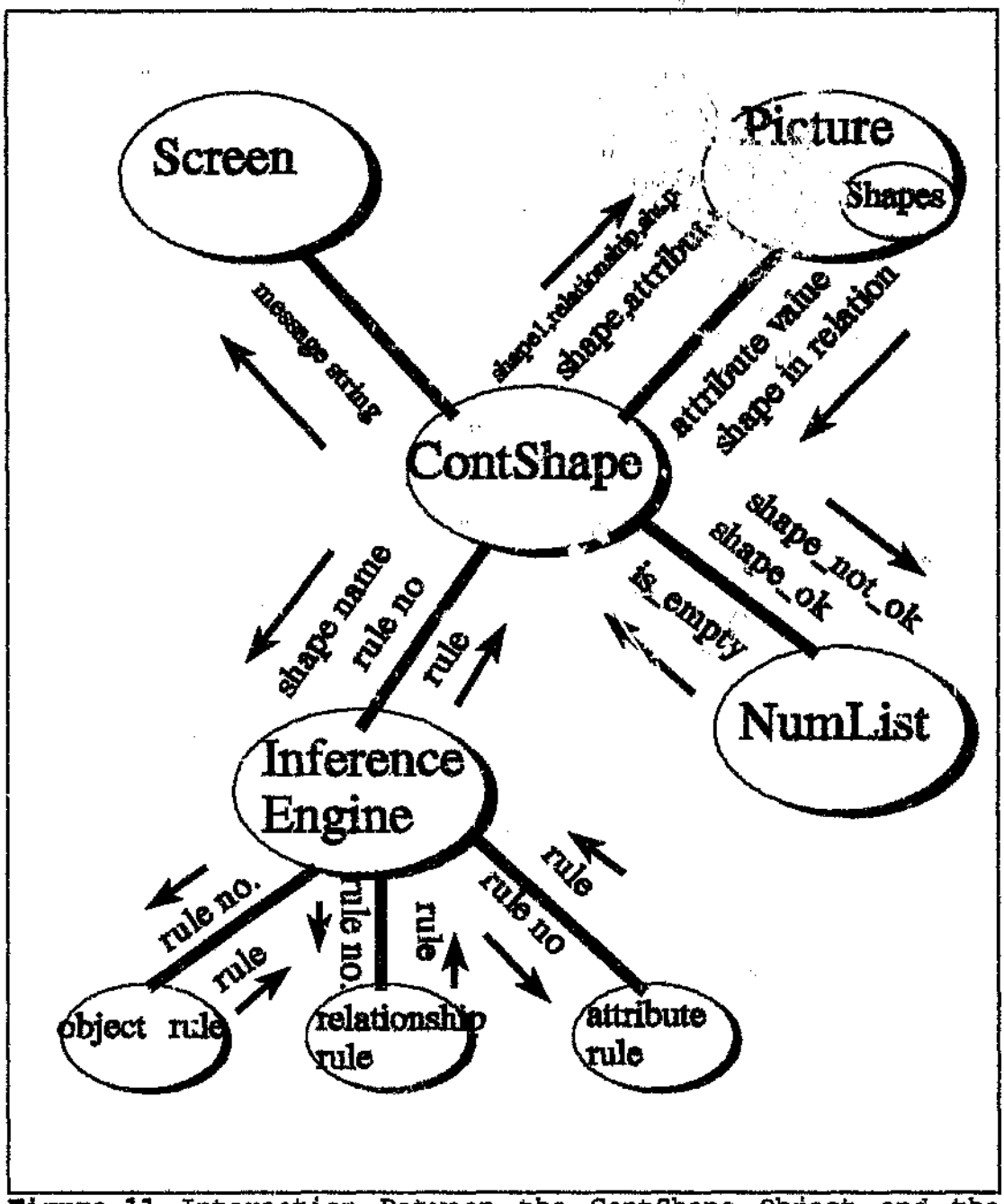


Figure 11 Interaction Between the ContShape Object and the Inference Engine and the Picture Object

The ContShape object carries out a similar function to the interaction handler in Tibbert and Bergeron's[19] system. It is preferable to return a string, instead of a function, to the ContShape object to deal with. This string can be used to place a message on the screen to indicate the error condition of the shape. Figure 11 shows how the ContShape object passes a string, of the message to be displayed, to the Screen object which is responsible for displaying this message. In this way the knowledge description (knowledge base) would not need to know about the internals of the ContShape object. If it were desirable to carry out some user defined function in the ContShape object, then it would be necessary for the knowledge base to know of the different types of functions that the ContShape object can evoke. The required function would be returned from the knowledge base to the ContShape object which would then evoke the function. This results in binding the knowledge base more closely to the ContShape object which makes the system more strongly coupled and less maintainable and less reusable. If at all, only a few general functions should be implemented so that the coupling between the knowledge base and the ContShape object is minimized. Such functions would, for example, be: delete entity, place a message onto the screen etc.

The picture object, list of entities within the CAD, and the rule base never interact directly with each other. The picture object is completely unaware of the existence of the rule base. It only knows how to iterate through all the entities and compare them to one another.

The picture object and the rule base interact indirectly with one another. The inference engine reads the rules from the knowledge bases and returns the rule just read in to the ContShape object. The ContShape object requests from the picture object to find all the entities that satisfy a relationship (if a relationship rule needs to be checked). If an attribute rule needs to be checked, the relevant attribute values of the entities are returned. The ContShape object compares the values of the entities with the values of the rules.

Interaction Between the inference engine and the rule base

The ContShape object decides which knowledge base the inference engine is to access next. The inference engine contains the object, relationship and attribute knowledge bases class objects. Each knowledge base object looks up its own rules and indexes the next rule to be looked up in one of the other rule bases. The inference engine is really a scheduler. It selects the knowledge base, i.e. the type of rule that needs to be tested next. The knowledge base then returns the rule to the inference engine which returns the rule to the ContShape object.

When the inference engine requests a rule to be looked up from one of the knowledge bases, the knowledge base returns the relevant rule but also increments the rule pointer to point to the next rule that needs to be looked up in its knowledge base. Each knowledge base object knows how to look up rules in its specific knowledge base. They do not interact with one another directly, although they index a rule contained in another knowledge base. See figure 11 for the interaction between the inference engine object and the knowledge base objects.

As has been mentioned, the knowledge bases are related to one another. The object knowledge base indexed rules in the relationship knowledge base. The relationship knowledge base references rules which exist in the attribute knowledge base. To look up a rule in the relationship knowledge base, the inference engine retrieves the current relationship rule that needs to be looked up from the object knowledge base. In a similar way an index to an attribute rule, relating to a relationship rule, is found in the relationship knowledge base.

Interacting between entities and the rule base

Tibbert and Bergeron[19] suggest that the knowledge base should interact directly with the entities in the CAD. The knowledge base compares the values of the entities in the database with those in the knowledge base in order to test the relationships and attributes of the entities. The knowledge base needs to have a working knowledge of the information in the database, i.e. its structure and the means of accessing it. The validity of the rule is found by comparing the values in the knowledge base with those in the database. It offers many advantages to allow the knowledge base to have the database in its scope so that it may easily and directly access the information in the database. However, this results in tightly coupling the knowledge base with the entities in the CAD.

The ContShape object interacts with the entities via the picture object. The ContShape object interacts with the knowledge bases via the inference engine object. Thus the entities interact with the rule base indirectly by means of the ContShape class. Figure 11 illustrates this.

By allowing the entities to have indirect communication with the rule base, the entities are decoupled from their rules in the rule base. Thus adding additional features to either the entities or the rule base does not affect how the one interacts with the other.

Conclusion

Although entities need to store information concerning themselves, they also need to store and retrieve details of relationships with other entities (Further detail about their relationships is irrelevant).

A picture class is introduced. It contains a complete list of entities in the CAD system. It enables entity's attributes to be compared to one another and it can pass on messages to the entities to manipulate themselves.

The relationship data member of a shape entity has no meaning to the entity itself. It defines the current state of relationships to other shapes. The shape object can return the existence of a relationship with other shape entities, the type of relationship, and the identification of the

entity that it is related to. As the shapes are created or manipulated the attributes are stored and evaluated.

The system consists of highly independent parts which all work together and form an integrated whole. Each part only needs know how to manipulate itself and how to deal with messages that are passed to it. This is the concept embodied by the object oriented paradigm.

By allowing the objects to interact indirectly with one another, enables the objects to be decoupled and thus makes the system more maintainable and reusable over time. Changes in one area of the code do not affect the other parts of the program.

An object oriented philosophy, using: inheritance, polymorphism and encapsulation enables a powerful maintainable and reusable system to be created. Inheritance allows additional classes to be added onto the class hierarchical tree, without having an effect or changing the implementation of the existing classes in the tree. The new class can be derived from one of the existing classes. For example an underground cable may be added as a new entity in the CAD. This added entity does not cause the implementation of any of the other entities to change in any way. The picture class, a container class, is unaffected because it uses polymorphism to select the desirable member function to be called, so that the relevant entity uses it own member function to manipulate its data.

The chapter has explained how the knowledge base is structured. The knowledge base is decomposed into three knowledge bases which encapsulate different types of information relating to the entities in the CAD. The object knowledge base is really a dictionary of where to find the relationship rules belonging to the entity being looked up. The relationship knowledge base contains a description of the types of relationships one entity may not have with another entity. Should the relationship rules be satisfied, a list of attribute rules which relate to the current relationship rules is given in the relationship rule. The attribute rule contains the values that the entities in the relationship must have in order for the relationship rule to be satisfied.

This structure represents relationships between entities and the allowable values that these entities may have. The advantages of such an approach is that rules can be quickly and easily located, and relationships between entities having certain attribute values can be effectively represented. Rules are used to check relationships between entities or the attribute values of entities. It is easy to add or remove rules because rules represented in one knowledge base are linked to rules represented in other knowledge bases. To delete a rule, it is only necessary to remove the indexes to rules in other knowledge bases and to add a rule it is necessary to add any rules which do not already exist and to index the rules from the different knowledge bases. This structure offers a uniformity of approach so that tools can be created to assist in the maintenance of the knowledge bases. The disadvantages are that to view a single rule it is necessary to view the rules in each of the separate knowledge bases and it is necessary to maintain three rather than a single knowledge base.

Data representation is extremely important for the knowledge bases. It determines whether the system is able to work with the expertise which is encapsulated in the form of rules. The inference engine must be able to read in the rules represented in the knowledge bases and infer additional information from the knowledge bases. The rules must be structured in a familiar format so that the inference engine may effectively manipulate the information contained in the knowledge base. The rules are represented as objects and these are manipulated by the inference engine. The inference engine and its knowledge bases in the prototype have managed to achieve these goals.

CHAPTER 8 CONCLUSION

Importance of Intelligent CAD Systems

CAD systems are important tools which are in widespread use today. What previously could have taken months to accomplish can now be achieved in a fraction of the time. They offer facilities to rapidly create complex entices, from a combination of more simple entities, which can be rapidly retrieved into the CAD system.

However CAD systems have great potential to become more advanced. Many more useful features can be added to enhance the performance of CAD. The functionality of the CAD system can be greatly enhanced by providing it with very basic intelligence.

The major benefit an intelligent CAD system offers is that it uses the computer more efficiently to share the work load of the designer. The computer may perform much of the tedious and monotonous work thus allowing the designer to spend his time more usefully.

Lessons learned from the Prototype

The object oriented approach enables each component of the CAD system to be created and maintained separately from the other components in the CAD system. The components are decoupled from one another. A change to a component does not effect the other components in the system.

The object oriented facilities of inheritance, polymorphism and encapsulation enable a powerful, maintainable and reusable system to be created. Inheritance allows additional classes to be added onto the class hierarchical tree, without having an effect or changing the implementation of the existing classes in the tree. The new class can be derived from one of the existing classes. For example an underground cable may be added as a new entity in the CAD. This added entity does not cause the implementation of any of the other entities to change in any way. The picture class, a container class, is unaffected because it uses polymorphism to select the desirable member function to be called, so that the relevant shape uses it own member function to manipulate its data.

The basic system presented herein may easily be modified to add much more capability. The expert system, rule bases, graphical interface and the picture object may easily be improved and extended without having an effect on the other parts of the program. The objective here has been to create a simple prototype which demonstrates that the basic goals of intelligent CAD can be achieved. Starting with an infrastructure, components which are strongly decoupled from one another, permits a more advanced and powerful intelligent CAD to be derived.

Expert systems isolate the control of the system from the program. Instead of the program control being carried out by algorithms implemented in the program, control is driven by the data that is represented in the knowledge base. The mechanism of control may be modified, by varying the way information is represented. Thus by isolating the knowledge from the internals of the program, the system becomes more maintainable. It is not necessary to dig into the code and spend hours debugging the changes.

Two types of relationships exist in the system: implicit and explicit rules. Explicit relationships need to permanently exist for an entity and are determined when an entity is created or modified. Implicit rules, which are subtle, need to be determined at run time. They are determined when requested by the program. These two types of relationships need to be separately catered for in the intelligent CAD system.

Data representation is extremely important. Each shape entity in the CAD should be able to store all the relevant attributes associated with it. It should be able to store relevant relationships to other entities in the system. The shape entities in the CAD should be structured in such a way that they may easily pass their attribute and relationship values to the objects requesting this information. The representation of the information in a rule-like format is crucial to the efficiency and effectiveness of the system. Effective representation enables the computer to manipulate the data easily, almost trivially, and it facilitates maintainability and extendability.

This system would be efficient with configurations with few entities. However the performance would reduce in proportion with an increase in the number of entities in the CAD. Performance also depends on the time it takes to determine a relationship and the number of different types of relationships that need to be determined. The performance could be optimised by introducing efficient searching strategies and algorithms for evaluating relationships. However depending on the mentioned factors, it might not be feasible to carry out the configuration checking during program execution and a similar approach to Tibbert and Bergeron[19] (in chapter 4) may need to be adopted where the configuration checking is performed overnight in a batch run.

Both a frame based and a blackboard approach (two approaches of implementing an expert system mentioned) to solving problems in an expert system are good techniques for encapsulating expertise and for finding the result to a series of rules. It is not critical which type of approach is implemented, because in either case the expert system may be modified, without affecting any other part of the program. However, a frame based approach generally offers a better representation of the data in the knowledge base and is generally more widely used than a blackboard architecture.

Recommendations for future work

The basic rules used in the prototype cannot deal with one to many relationships. This means that in order for the rule condition to be satisfied, more than one entity must simultaneously satisfy a relationship with the current entity. For example an invalid situation may exist if three transformers lie upon a single residential stand. The prototype requires more sophisticated algorithms to deal with this situation.

A rule dictionary or a relational database needs to be used to optimise the search for rules in the knowledge base. This facility speeds up the rule matching process. Additional information could also be incorporated in a rule such as the date and time the rule was created,the person who created the rule and a help message explaining the relevance of the rule in the context in which it is called.

To overcome the difficulty in capturing the relevant rules to be used by the system, the user should be allowed to dynamically type in the rules and test them during program execution. In this way an expert can capture the rules in the system in an iterative and interactive way.

It is undesirable for the operator to wait for the integrity of the system to be checked. The process of checking the validity of the action of the operator may take considerable time to evaluate. In the absence of adequate computing power, the expert system need not necessarily be a real time system which informs the operator of invalid conditions dynamically. Possibly the entire process of checking for errors could be performed overnight and run in a batch mode. An error report could be sent to a file to be later analysed by the designer. It could point out places where a design decision is flawed and possibly suggest corrections. The results of all the invalid or non optimum solutions could be collected during a session in a file and be viewed later.

Alternatively the non active time of the computer can be used to check the integrity of the system. While the operator is thinking the expert system could be carrying out its evaluation. It would be advantageous to time slice the work of the CAD.

Another approach is to execute the CAD system and the expert system as separate programs. The programs can run on the same machine in a multi-tasking environment or even on different machines over a network. These programs may communicate with one another by sharing common data files. The expert system can be left continuously running and checking the validity of the CAD database. With every update, the CAD can pass the new information onto the concurrently running expert system and request the expert system to check the validity of the newly created entity. Should the expert system find that an invalid configuration exists, it can send back a message of the error in the system to the common data area used by both programs. This error can be retrieved up by the CAD system which can then inform the user of the undesirable design condition in the system. An object oriented approach enables components to be incorporated together within a single program or to be implemented as separate programs quite easily.

By working with the CAD in this type of multi-tasking environment, it could perhaps be feasible to run the system in real time. Each new design decision could be placed at the end of a queue to be evaluated later by the expert system. Thus after an item has been processed on a queue, the next item in the queue would be processed. If the entity on the queue is found to be invalid, the operator is informed of the bad decision. Thus soon after drawing an invalid configuration, a message could inform the designer of what the invalid design decision was, which entities were affected and what action to take.

It is important for the CAD system to be able to communicate effectively with the expert system. Messages that are passed between these two systems should have a common structure. This structure needs to be formally defined so that all the information that is passed between these two systems is complete and fully understood. By allowing the separate and independent components to communicate with one another, enables them to be integrated together.

The situation may arise that the system has insufficient information to fire a rule. It would be useful to have a mechanism whereby the evaluation of a rule could be temporarily paused. Subsequently when sufficient information is determined, the rule may be applied from the point it is paused and verified further. To cater for this situation, it is useful to have a date flag in each entity that indicates that the entity is in the process of being validated in a rule, and that a particular attribute of the entity must be determined in order for the rule to fire. Every time this entity is invoked to perform some operation, it could check if the particular date flag field has a value. If it does then the rule could be triggered to continue from where it left off. This requires a record to be kept of the current steps that have been invoked, i.e. a history of the rules which have been called for each entity. This approach offers many possibilities in dealing with rules that have incomplete information. They can be allowed to fire later on in the future. If they never fire, the user can be informed that a certain rule has not fired due to incomplete information. The user can then check to see why the validity of the shape could not be checked.

Conclusions

Factors such as the growing complexity of engineering systems and the increasing sophistication of user requirements demand the introduction of greater degrees of intelligence into modern systems to cope with the need for higher levels of self checking and analysis. The prototype presented in this project report hints at the potential benefits brought about by increasing the overall intelligence of CAD systems.

Nevertheless expert systems should be used appropriately in intelligent CAD systems. The efficiency and effectiveness of their operation must be based on their usefulness and functionality. It is important to build on small successes and make incremental gains. We must be concerned with automating the tasks that we know how to do well.

The trend may arise in the next decade where CAD systems may have intelligent modules imbedded within them. The integration of knowledge processing capabilities with conventional computing facilities will usher in a new era of computing. The demands and impact that this synthesis will make on computer architectures, hardware and software are not generally well understood. A system approach such as the one adopted in this work will be vital in managing this new generation of intelligent computing.

APPENDIX A CLASS STRUCTURE OF THE PROTOTYPE

The class structure of the prototype is shown in the Booch[1] diagram of figure 12. The arrows indicate inheritance and the lines with open circle indicate membership to a class.

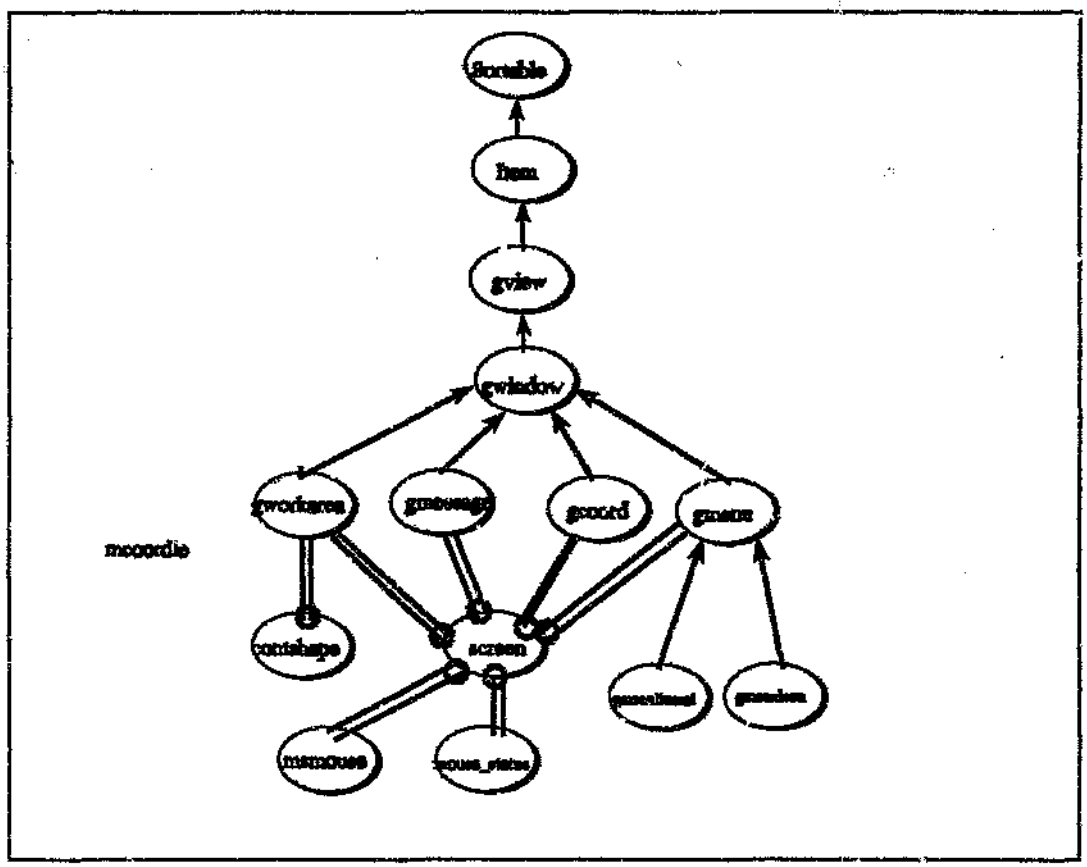


Figure 12 Class Structure of the User Interface for the CAD Prototype

Class *Sortable* is derived from the *Object* class from Borland's C++ container class. A container is a structure that can contain elements of other data types or classes. The sortable base class contains elements which are arranged in a predefined order.

Class *Item*, derived from Sortable, is an individual element which may be arranged in a sorted container.

Class *gview*, derived from Item, is a rectangular area which defines the area a window or a shape entity is enclosed by.

Class *gwindow*, derived from gview, acts as the base class for the various user interface

windows. Four types of windows are derived from gwindow. A *gmessage* window displays messages in its window which indicates the status of the program to the user. A *gcoord* window displays the X and Y coordinates of the mouse. A *gmenu* window contains menu items. A *gmenuliteral* and a *gmenuicon* are a specialised menus. A gmenuliteral describes the menu items with strings while a gmenuicon displays menu options using icons.

A *gworkarea* window, derived from gwindow, is the user window where the user draws the shape entities. The gworkarea class contains the contshapes class.

The *contshape* class controls the interaction of the expert system with the entities in the CAD system. This is where most of processing in the CAD system occurs.

The *screen* class is the interface between the user and the application. It contains the gworkarea, gmessage, gcoord and gmenu classes which are windows which display information on the video terminal. The screen class contains the *msmouse* class which handles all the input into the system and the *mouse_status* class which describes the current status of the mouse.

APPENDIX B HEADER FILES OF THE PROTOTYPE

The header files are included here to describe the structure and approach taken in the prototype. The *.cpp files containing the implementation of the member functions of the header files have been omitted.

```
*****************************************************************************************************
//relationship rule class

#ifndef AttRuleTemplate
#define AttRuleTemplate


#include "rulelist.hpp"

class AttRule : public RuleList
{
   char objName[15];
   char field[15];
   char oper[5];
   char value[15];
   char outcome[15];
public:
                        AttRule() : RuleList("relships.asc") {}
                        AttRule(char *fil) : RuleList(fil) {}
   virtual              ~AttRule()                          {}
   virtual       void   buffFormat(void);
                 char*  objName_(void)                      {return objName;}
                 char*  field_(void)                        {return field;}
                 char*  oper_(void)                         {return oper;}
                 char*  value_(void)                        {return value;}
               AttRule& operator=(const AttRule&);
};


#endif AttRuleTemplate

*****************************************************************************************************
//contshape class
//controls access to the inference engine and the list of shapes

#ifndef ContShapeTemplate
#define ContShapeTemplate

#include "picture.hpp"
#include "infeng.hpp"
#include "numarray.hpp"
#include "relrule.hpp"

#define NOFIELD -5       //different outcomes for a rule
#define OBJFAIL  -4
#define ATTRFAIL -3
#define RELFAIL  -2
#define NOTFOUND -1
#define FAIL      0
#define SATISFIED 1

class ContShape
{
   picture thePict;      // a list of shapes
   InfEng I_E;           //inference engine
   NumArray numlist;     //list of shapes satisfying a relationship
   int curshnum;         //the current shape being manipulated
   char curshname[15];  //name of the shape being manipulated
   RelRule relshRule;   //relationship rule data type
   AttRule attribRule;   //attribute rule data type
   char attribval[15];   //attribute value of an attribute rule
   char oper[3];         //comparison operator of attributes
   char fldval[15];      //the field value returned by the shape
public:
   ContShape() {}

   ~ContShape() {}
   void put(NumItem& anItem); //adds an item onto the list of shape numbers
   void eventhandle(void); //deals with messages passed to this object
   void checkattr(void);//compares the attributes of rule base with that of
                        //the shape using operators >= <= etc
   int compAttr(char*);  //compares attrubutes, returns 0(false) or 1(true)
   void itemRemove(int);//removes an item from the list of items
   int listEmpty(void);  //returns 1 if a list is empty
   void rulesChk(int);   //checks the validity of the shapes drawn with
                         //the rule base
   RelRule&  relatRule(void);
   int attrGet(void);            //gets the attributes from the shape list
   void findRelations(void);     //finds specific relationships which exist in
                                 //in the picture list of shapes
   int chkAttr(void);            //validates the attributes of a shape in a rule
};

#endif ContShapeTemplate

*****************************************************************************************************
```

```
//EVENT class
#if !defined(EVENTx)
#define EVENTx

#include <string.h>

#define ComBlank        0x0000          //macros for commands available to
#define ComQuit         0x0100          //graphical user interface
#define ComDrawTriang   0x0101
#define ComDrawRect     0x0102
#define ComDrawCurve    0x0103
#define ComCreateTriang 0x0104
#define ComCreateRect   0x0105
#define ComCreateLine   0x0106
#define ComMove         0x0120

class EVENT
{
        int xpt;                //x,y values of the mouse
        int ypt;
        unsigned int command;   //command for graphical interface
        char evquote[80];       //message string to graphical interface
        int menustatus;         //menu active or not
  public:
        EVENT() : xpt(0),ypt(0),Command(ComBlank),menustatus(0)   {}
        void point(int x,int y)      { xpt=x;ypt=y;}
        int getxpt(void)             {return xpt;}
        int getypt(void)             {return ypt;}
        char* getquote(void)         {return evquote;}
        void putquote(char* quot)      {strcpy(evquote,quot);}
        void putcommand(unsigned int com)  {Command=com;}
        unsigned int getcommand(void)      {return Command;}
        void setmenustatus(void)       {menustatus=1;}
        void resetmenustatus(void)     {menustatus=0;}
        int getmenustatus(void)        {return menustatus;}

};

#endif

//*************************************************************************************************
// file lookup class
//access a sequential file

#ifndef FlookupTemplate
#define FlookupTemplate

#include <fstream.h>
#include <iostream.h>

class FiLookUp
{
   ifstream theFile;
   char buffer[99];
public:
   FiLookUp()   {}
   FiLookUp(char *fil) : theFile(fil) {if (!theFile) cout << "an error has occurred\n";}
   ~FiLookUp()                  {theFile.close();}
   void firstrec(void)        {theFile.seekg(0,ios::beg);}
   istream& readline(void)    {return (theFile.getline(buffer,44));} //read a line from a file
   int readline(char*);
   char* buff(void)             {return buffer;}
};

#endif FlookupTemplate

//*************************************************************************************************
//class gmenuwind
//Derived from class gwind. A menu is a window
//which when selected becomes highlighted to show it is selected
// and sets a command flag.

#include "window.hpp"
#include "shape.hpp"
#include <string.h>


class gmenuwind : public gwindow
{
   int chosen;       //is always 0 but 1 w  n then menuitem is chosen
   unsigned int mcommand;  //menu command   ero
public:
   gmenuwind() : chosen(0),mcommand(ComBlank) {}
```

```
   gmenuwind(int xl,int yt,int xr,int yb,unsigned int com=0) : gwindow(xl,yt,xr,yb,1),mcommand(com) {}
   unsigned int ischosen(void)        {return chosen;}
   void setchosen(int bool)           {chosen=bool;}
   virtual void select(void) {}                          //select a menu item
   virtual void deselect(void){}                         //deselect a menu item
   void setcommand(unsigned int com)  {mcommand=com;}
   unsigned int getcommand(void)      {return mcommand;}
   virtual void eventhandle(void)  {}                    //after this menu is select, process the event
};
```

```
class gmenulitr : public gmenuwind
{
   char litteral[10];                    //menu command litteral (e.g. move)
public:
   gmenulitr() {}
   gmenulitr(int lef,int top,int ri,int bot,char* lit,unsigned int com)
            : gmenuwind(lef,top,ri,bot,com)
            {
               strcpy(litteral,lit);
               message(litteral);
            }
   virtual void select(void)     {int col=getcolor();
                                  setcolor(RED);
                                  setchosen(1);
                                  message(litteral);
                                  setcolor(col);
                                  test.putcomnd(getcommand());
                                 }
   virtual void deselect(void)   {int col=getcolor();
                                  setcolor(WHITE);
                                  setchosen(0);
                                  message(litteral);
                                  setcolor(col);
                                  test.putcomnd(ComBlank);
                                 }
};
```

```
class gmenuicon : public gmenuwind
{
   gshape* icon;
   unsigned int comname;
public:
   gmenuicon() {}
   gmenuicon(int le,int to,int ri,int bo,gshape& sh,unsigned int com)
             : gmenuwind(le,to,ri,bo,com)
               {
                icon=&sh;
                icon->show();
               }
   virtual void select(void)   {
                                int col=getcolor();
                                setcolor(RED);
                                setchosen(1);
                                icon->select();
                                setcolor(col);
                                test.putcomnd(getcommand());
                               }
   virtual void deselect(void) {
                                int col=getcolor();
                                setcolor(WHITE);
                                setchosen(0);
                                icon->deselect();
                                setcolor(col);
                                test.putcomnd(ComBlank);
                               }
};
```

```
class gmenulist : public gmenuwind
{
   gmenuwind* menu[5];
   int numitems;
       public:
   gmenulist() : numitems(0),gmenuwind(0,0,getmaxx(),30)
         {
          show();
          menu[numitems++]=new gmenulitr(getmaxx()-40,15,getmaxx(),30,"quit",ComQuit);
          menu[numitems++]=new gmenulitr(getmaxx()-80,15,getmaxx()-40,30,"move",ComMove);
          menu[numitems++]=new gmenuicon(0,0,40,30,*new gcurve(38,5,7,25),ComDrawCurve);
          menu[numitems++]=new gmenuicon(40,0,80,30,*new gtriangle(60,5,44,25,76,25),ComDrawTriang);
          menu[numitems++]=new gmenuicon(80,0,120,30,*new greotangle(90,5,110,25),ComDrawRect);
         }
   virtual void select(void)  {}
   virtual void deselect(void){}
```

```
   virtual void eventhandle(void);
};

*****************************************************************************************************
//gview class
// view is a rectangular area in which the object derived from it lies.

#if !defined(GVIEW)
#define GVIEW


#include "item.hpp"
#include "event.hpp"


class gview : public Item
{
   protected:
        int x_left,x_right,y_top,y_bot;          //extremes (edges) of the view
        static EVENT test;                       //present status of keys pressed or menus selected
   public:
        gview() {}
        gview(int,int,int,int);
        int isselected();                        //returns true if the view has been selected
        virtual void eventhandle(void)  {}      //when control is given to this view, it handles the class

};

#endif GVIEW

*****************************************************************************************************
//class for inference engine control
//

#ifndef InfControlTemplate
#define InfControlTemplate

#include "infeng.hpp"

class InfControl : public InfEng
{
   char otherObj[15];                    //other object in the relationship
   char theRelation[15];          //description of the relationship
   char fieldval[15];                    //the field value of the attribute typed in by the user
public:
                        InfControl()     {}
                        ~InfControl()  {}
                void    control();       //controls the looking up of the InfEng
                void    inputRel(void); //type in the relationship and object
                int     compRel(void);  //determines if a relationship is satisfied
                int     attrGet(void);  //gets and tests all the attribute rules
                void    inputAttr(void);//reads in the attribute values from the user
                int     compAttr(void); //compares the rule attributes with those input by the user
};

// inference engine class
//controls the looking up of rules in the knowledge bases

#ifndef InfEngTemplate
#define InfEngTemplate

#include "objrule.hpp"
#include "attrule.hpp"
#include "relrule.hpp"

class InfEng
{
   ObjRule theObject;            //the object knowledge base
   AttRule theAttrib;            //the attribute knowledge base
   RelRule theRelation;          //the relationship knowledge base
public:
                        InfEng() : theObject("rules.asc"),theRelation("relships.asc"),\
                           theAttrib("attrib.asc")
                           {}
                        ~InfEng()  {}
                int     objGet(char *);   //gets the list of relationship rules relevant for an object
                int     relGet(char *);   //gets the list of attribute rules relevant for a relation
                int     attrGet(char *);  //sets up the field for the attribute rule base
                char*   curObjRul(void);  //returns current object rule indicating a relationship rule
                char*   relsh(void);       //return the relation
                char*   obj_2(void);       //returns the other object
                char*   curRelRul(void);  //returns the attribute rule to be looked up
                char*   field(void);       //returns the field of the current attribute rule
                char*   value(void);       //returns the value of the field
                char*   oper(void);        //returns the operator of the current attribute rule
                int     nextObjRul(void); //return the next object rule refering to a relationship
```

```
                int     nextRelRul(void); //return the next relation rule no
               RelRule& retRel(void);      //return the relationship rule
               AttRule& retAttrib(void);  //returns the attribute rule
};



#endif InfEngTemplate


#endif InfContTemplate

*****************************************************************************************************
//item.hpp
//derived from the sortable object. is used as the base class for objects
//belonging to a container object

#ifndef ItemTemplate
#define ItemTemplate


     #include <iostream.h>
     #include "sortable.h"


class Item : public Sortable
{
// ---- Public and Protected Member Functions (i.e. Interface Operations) ----

   public:

    Item ()             {}
    Item(int x)   : value(x) {}
    virtual    ~Item()  {}                  // destructor
    virtual void printOn( ostream& ) const ;
    virtual classType isA() const ;
    v      l int isEqual( const Object& ) const ;
    v      l int isLessThan( const Object& ) const ;
    v      l char *nameOf() const ;
    virtual hashValueType hashValue() const ;
    virtual int isSortable() const;
            int& val(void);
private:
    int value;
};


#endif ItemTemplate

*****************************************************************************************************
//ItemArray class
//Item container class. Contains any number of items or objects derived
//from items

#ifndef ItemArrayTemplate
#define ItemArrayTemplate

//#include <Sortarry.h>
#include <array.h>
#include "item.hpp"


class ItemArray : public Array     //SortedArray
{
public:
                   ItemArray();
                   ItemArray(int,int,int);
   virtual         ~ItemArray();
           Item&   operator[](int);
};

#endif ItemArrayTemplate


//item list class
//contains the item container class
//contains a list of items

#include "item.hpp"
#include "itemarry.hpp"

class ItemList
{
   ItemArray theItems;
public:
```

```
                        ItemList() {}
   virtual              ~ItemList() {}
                void    add(Item& aItem);
                void    stepThrough();
};

*****************************************************************************************************

//points class contains an x and y coordinate
//lines are the basic building blocks of all the shape objects.

#if !defined(LINES)
#define LINES

#include "gview.hpp"
class points
{
        public:
   int xc;
   int yc;
   points()  {xc=0;yc=0;}
   ~points() {}
};


class Line : public gview
{
   points pt[2];
   double ycut;
   double slope;
        public:
   Line() {}
   void setup(points,points);  //set up pair of x and y points to form a line
   int operator==(const Line&); //returns 1 if lines intersect
   int ison(int,int);        //returns true if a point is on a line
   int intersect(Line,Line);
   double getslope(void)    {return slope;}
   void show(void);             //draws a line
   void hide(void);             //hides a line
   void select(void);
   void deselect(void);
};


#endif

*****************************************************************************************************
//a header file that contains the class declaration and support decalarations
//for the microsoft mouse class

#if !defined (MSMOUSE)
#define MSMOUSE

#include <dos.h>
#include <graphics.h>
#include <conio.h>

#define callmouse int86(0x33, &reg,&reg)
#define NOBUTTON     0
#define LEFTBUTTON   1
#define RIGHTBUTTON  2
#define BOTHBUTTON   3
#define vmaxx  0
#define vmaxy  1
#define xsize  2
#define ysize  3

//screen table can be used to convert mouse to real coords
//mousex = (x-1) * screentable[crtmode][xsize]
//mousey = (y-1) * screentable[crtmode][ysize]
//x = (mousex /screentab[crtmode][xsize ) +1
//y = (mousey /screentab[crtmode][ysize ) +1
//this will take into account the difference between TC 1-origin and mouse
//0-origin coordinate systems in text modes. Graphics modes already use
//0-origin coordinates.

extern int screentab[20][4];

typedef struct
{
   int verify;
   int crtmode;
   int cursoron;
   int buttons;
   int numpress;
   int x;               //x and y screen coords
```

```
   int y;
   int minx;              // min-max values are in mouse coords
   int maxx;
   int miny;
   int maxy;
   int crtpage;
} mstat_t;

class mxmouse
{
        union REGS reg;
        struct SREGS seg;
        mstat_t mstat;
        void interrupt ( far *oldx33)(...);
    public:
        void reset();
        mxmouse();
        ~mxmouse();
        int initok()  {return mstat.verify;}
        void showcursor();
        void hidecursor();
        mstat_t *getposition();
        void setposition(int = 0,int = 0);
        mstat_t *getbutton(int);
        mstat_t *releasebutton(int);
        mstat_t *xonex(int,int);
        mstat_t *xoney(int,int);
};

#endif

*****************************************************************************************************
//NumArray class
//Number container class

#ifndef NumArrayTemplate
#define NumArrayTemplate

//#include <array.h>
#include "itemarry.hpp"
#include "numitem.hpp"


class NumArray : public ItemArray
{
public:
                    NumArray();
                    NumArray(NumArray&);     //copy constructor
   virtual          ~NumArray();
          NumItem& operator[](int);
          void     operator=(NumArray&);   //sets one array equal to another
};
#endif NumArrayTemplate

*****************************************************************************************************
//class NumItem - a number item in the numarray container class

#ifndef NumItemTemplate
#define NumItemTemplate

#include "item.hpp"

class NumItem : public Item
{
   int number;
public:
                        NumItem() : number(0) {}
                        NumItem(int theNum) : number(theNum)  {}
   virtual              ~NumItem()          {}
                int&    numb(void)          {return number;}
};

#endif NumItemTemplate

*****************************************************************************************************
//object rules - rules for the object knowledge base

#ifndef ObjListTemplate
#define ObjListTemplate

#include "rulelist.hpp"

class ObjRule : public RuleList
{
public:
                        ObjRule() : RuleList("rules.asc") {}
```

```
                        ObjRule(char *fil) : RuleList(fil) {}
   virtual              ~ObjRule()    {}
   virtual      void    bufFormat(void);
};

#endif ObjListTemplate

*****************************************************************************************************
//picture class
//contains the shape container class

#include "shape.hpp"
#include "shaparry.hpp"
#include "numarray.hpp"

class picture
{
   ShapeArray theShapes;
   int        curShape;                          //the current shape no
public:
                        picture() : curShape(-1) {}
   virtual              ~picture() {}
                void    put(gshape& aShape);
                int     current(void)           {return curShape;}
                void    stepThrough();
                void    add(gshape&);
                int     ontopof(void);      //determines a relationship between 2 shapes
                int*    relMatches(char*,char*);   //returns an array of shape no's
                                                   //that satisfy the relationship
                gshape& getShape(int);             //returns a shape no.
                char*   getAttr(int,char*);    //gets an attribute of a shape
                char*   reciprocal(char*);     //finds the reciprocal of a relationship
};

*****************************************************************************************************
//some power object classes
//class cable
//class stand
//class transformer

#ifndef PowerObjects
#define PowerObjects

#include <string.h>
#include "shape.hpp"

class Cable : public qcurve
{
   int voltage;
   int current;
   int power;
public:
                        Cable(int x1,int y1,int x2,int y2,char* name="cable") :
                                qcurve(x1,y1,x2,y2,name),voltage(300),current(500) {}
   virtual              ~Cable() {}
   virtual      char*   getAttrib(char* theAtt);      //returns the value of a Cable attribute
};

-----------------------------------------------------------------------------------------------------
class  Stand : public qrectangle
{
   char type[15];
public:
                        Stand(int x1,int y1,int x2,int y2,char* name="stand") :
                                qrectangle(x1,y1,x2,y2,name)
                                {strcpy(type,"commercial");}
   virtual              ~Stand()  {}
   virtual      char*   getAttrib(char* theAtt);  //returns the value of a stand attribute
};

-----------------------------------------------------------------------------------------------------
class Transformer : public qtriangle
{
public:
                        Transformer(int x1,int y1,int x2,int y2,int x3,int y3) : \
                                        qtriangle(x1,y1,x2,y2,x3,y3)  {}
   virtual              ~Transformer() {}
   virtual      char*   getAttrib(char* theAtt);  //returns the value of a transformer attribute

};

#endif PowerObjects

*****************************************************************************************************
```

```
//RelshArray class
//container class for relationship rules

#ifndef RelshArrayTemplate
#define RelshArrayTemplate

//#include <array.h>
#include <string.h>
#include "itemarry.hpp"
#include "relship.hpp"


class RelshArray : public ItemArray
{
public:
                    RelshArray();
   virtual          ~RelshArray();
           Relationship&   operator[](int);
};

#endif RelshArrayTemplate

*************************************** ***************************************************************************
//relationship rule class
//describes a relationship between two objects

#ifndef RelRuleTemplate
#define RelRuleTemplate


#include "rulelist.hpp"

class RelRule : public RuleList
{
   char obj1[15];
   char obj2[15];
   char relation[15];
public:
                        RelRule() : RuleList("relships.asc") {}
                        RelRule(char *fil) : RuleList(fil) {}

   virtual              ~RelRule()                          {}
   virtual      void    bufFormat(void);
                char*   rel(void)                           {return relation;}
                char*   thisObj(void)                       {return obj1;}
                char*   othObj(void)                        {return obj2;}
                 RelRule& operator=(const RelRule&);

};

#endif RelRuleTemplate

*****************************************************************************************************************
//class Relationship
//describes a relationship between two objects. Is a data member in
//the shape class

#ifndef RelationshipTemplate
#define RelationshipTemplate

#include "item.hpp"

class Relationship : public Item
{
   int num;              //the no. that it is related to
   char desc[15];        //description of the relationship
public:
                        Relationship() : num(-1) {}
                        Relationship(int val,char* rel)  : num(val) {strcpy(desc,rel);}
   virtual              ~Relationship()         {}
                int&    numb(void)          {return num;}
                char*   descrip(void)       {return desc;}
};

#endif RelationshipTemplate

*****************************************************************************************************************
//RuleArray class
//container class for a rule

#ifndef RulArrayTemplate
#define RulArrayTemplate

#include <array.h>
#include "ruleitem.hpp"
#include "itemarry.hpp"
```

```
class RuleArray : public ItemArray
{
public:
                        RuleArray();
                        RuleArray(int fir,int las,int sz) : ItemArray(fir,las,sz) {}
   virtual              ~RuleArray();
           RuleItem&    operator[](int);
};

#endif RulArrayTemplate

*****************************************************************************************************
//class RuleItem
//base class for different types of rules

#ifndef RuleItemTemplate
#define RuleItemTemplate

#include <string.h>
#include "item.hpp"

class RuleItem : public Item
{
   char num[6];
   char oper[3];
public:
                        RuleItem()  {strcpy(num,"\x0");}
                        RuleItem(char* no,char *op)  {strcpy(num,no);strcpy (oper,op);}
   virtual              ~RuleItem()         {}
                char*   ruleno()             {return num;}
                char*   operat(void)         {return oper;}
};

#endif RuleItemTemplate

*****************************************************************************************************
//class RuleItem
//base class for different types of rules

#ifndef RuleItemTemplate
#define RuleItemTemplate

#include <string.h>
#include "item.hpp"

class RuleItem : public Item
{
   char num[6];
   char oper[3];
public:
                        RuleItem()  {strcpy(num,"\x0");}
                        RuleItem(char* no,char *op)  {strcpy(num,no);strcpy (o    ,op);}
   virtual              ~RuleItem()         {}
                char*   ruleno()             {return num;}
                char*   operat(void)         {return oper;}
};

#endif RuleItemTemplate

*****************************************************************************************************
//ruleList class
//contains the rules container class

#ifndef RuleListTemplate
#define RuleListTemplate

#include "flookup.hpp"
#include "ruleitem.hpp"
#include "rularray.hpp"

class RuleList : public FiLookUp
{
   RuleArray theRules;
   int currentIdx;        //index of current rule
public:
                        RuleList() {}
                        RuleList(char *fil) : FiLookUp(fil),currentIdx(0) {}
   virtual              ~RuleList() {}
                void    put(RuleItem& aRule);
                void    stepThrough();
                int     readLine(char*);
   virtual      void    bufFormat(void);
                void    rulFormat(char*);
                int     current(void);       //returns the index to the current rule
                char*   currentRule(void);   //returns the current rule
```

```
                void    currentReset();       //resets the current index
                int     getRule(char *);      //finds a rule from file
                int     currentInc();          //increments currentIdx
};

#endif RuleListTemplate

*****************************************************************************************************
//screen class which controls the operation of the program
//this class is the graphical user interface

#if !defined(SCREEN)
#define SCREEN

#include "mmouse.hpp"
#include "window.hpp"
#include "gmenu.hpp"
//#include "manuopt.hpp"
#include <stdio.h>
#include <stdlib.h>

class screen : public gview
{
        mstat_t *mstat;
        mmouse *mouse;
        gmenulist wmenu;
        gcoord wcoords;
        gworkarea wuser;
        gwindow wmsg;          //display informative message in window
   public:
        screen();
        ~screen()               {delete mouse;}
        void mouseinit(void);
        void control(void);
        void eventhandle(void);
        void whatmsi(int,int);
};


#endif

*****************************************************************************************************
//ShapeArray class
//Shape container class


//#include <array.h>
#include "itemarry.hpp"
#include "shape.hpp"


class ShapeArray : public ItemArray
{
public:
                ShapeArray();
   virtual      ~ShapeArray();
          gshape&   operator[](int);
};

*****************************************************************************************************
//gshape class
//gcurve class
//gtriangle class
//grectangle class

#if !defined(SHAPE)
#define SHAPE

#include "gview.hpp"
#include "lines.hpp"
#include <string.h>
#include "relarray.hpp"
#include "relship.hpp"


class gshape : public gview
{
      protected:
   int numpts;
   int numlines;
   points* pnt;
   Line *lin;
   char sname[15];
   int snumber;
   static int curpt;
   static char shquote[80];
```

```
   RelshArray relation;
        public:
   gshape()  {}
   gshape(int num) : numpts(num) {}
   gshape(int pts,int line,char* name="shape") : numpts(pts),numlines(line)  {strcpy(sname,name);}
   virtual ~gshape()  {resetrel();}
   void shade(void);
   void unshade(void);
   void putf[illegible]name(void);
   int fra[illegible]verlaps(gshape&);
   Line& getline(int i) {return lin[i];}
   int getnumpts(void)  {return numpts;}
   int getnumlines(void) {return numlines;}
   char* getsname(void) {return sname;}
   void hide(void);
   void addrel(int,char*);
   void resetrel(void);
   int relNo(int i)        {return relation[i].num();}
   char* relDesc(int i)     {return relation[i].descrip();}
   RelshArray& getRel(void) {return relation;}
   virtual void destroy(void);      //deletes the lines and points of a shape
   virtual void eventhandle(void) {}
   virtual void show(void);
   virtual void select(void) {}
   virtual void deselect(void) {}
   virtual int  add(int,int);
   virtual void move(int,int);
   virtual void setup(void) {}
   virtual char* getAttrib(char*);   //returns the value of an attribute for shape
};

-------------------------------------------------------------------------------

class gcurve : public gshape
{
        public:
   gcurve();
   gcurve(int x1,int y1,int x2,int y2,char* ="curve");
   virtual ~gcurve();
   virtual void eventhandle(void) {}
   virtual void show(void)      {gshape::show();}
   virtual void select(void)    {gshape::show();
                                  test.putquote("Click the mouse for the first point of the curve");
                                 }
   virtual void deselect(void) {gshape::show();}
   virtual int add(int,int);
   virtual void move(int,int);
   virtual void setup(void);
   virtual char* getAttrib(char*); //returns the value of an attribute for curve
};

-------------------------------------------------------------------------------

class gtriangle : public gshape
{
        public:
   gtriangle();
   gtriangle(in[illegible]t,int,int,int,int,char* ="triangle");
   virtual ~gtr[illegible]le();
   void fill(void);     //sets up triange for shading in
   virtual void show(void)      {gshape::show();}
   virtual void select(void)    {gshape::shade();
                                  fill();
                                  test.putquote("Click the mouse for the first point of the triangle");
                                 }
   virtual void deselect(void) {gshape::unshade();fill();}
   virtual int add(int,int);
   virtual void move(int,int);
   virtual void setup(void);
   virtual char* getAttrib(char*); //returns the value of an attribute for triangle
};

-------------------------------------------------------------------------------

class grectangle : public gshape
{
        public:
   grectangle();
   grectangle(int,int,int,int,char* ="rectangle");
   virtual ~grectangle();
   void fill(void);
   virtual void show(void)      {gshape::show();}
   virtual void select(void)    {gshape::shade();
                                  fill();
                                  test.putquote("Click the mouse the first edge of the rectangle");
                                 }
   virtual void deselect(void) {gshape::unshade();fill();}
   virtual int add(int,int);
```

```
   virtual void move(int,int);
   virtual void setup(void);
   virtual char* getAttrib(char*); //returns the value of an attribute for rectangle
};

#endif

*****************************************************************************************************
//window.hpp contains the window class and necessary functions to display
//and manipulate a window on the screen.
//also contains gworkarea class and gcoord class

#if !defined(WINDOW)
#define WINDOW

#include "gview.hpp"
//#include "shape.hpp"
//#include "picture.hpp"
//#include "rule.hpp"
#include <graphics.h>


class gwindow :public gview
{
   public:
        gwindow(){}
        gwindow(int xl,int yt,int xr,int yb,int display=0) : gview(xl,yt,xr,yb)
          {
                if (!display) show();
          }
        virtual void eventhandle(void) {}
        void message(char *);
        void wclear(void);
        void hide(void);
        void show(void);

};
-----------------------------------------------------------------------------------------------------
class gcoord : public gwindow
{
   public:
        gcoord() : gwindow(getmaxx()-130,1,getmaxx(),15) {show();}
        gcoord(int xl,int yt,int xr,int yb) : gwindow(xl,yt,xr,yb) {show();}
        void messg(void);
};
-----------------------------------------------------------------------------------------------------
class gworkarea :public gwindow
{
   public:
        gworkarea() : gwindow(0,30,getmaxx(),getmaxy()-20) {show();}
        void eventhandle(void);

};

#endif
*****************************************************************************************************
```

REFERENCES

1. Booch G, Object Oriented Design with Applications, *The Benjamin Cummings Publishing Company Inc, 1991*

2. Goodman A.M, Haralick R.M, and Shapiro L.G, Knowledge - Based Computer Vision, *Computer, December 1989*

3. Eisenstadt M, Domingue J, [illegible]ta E, Visual knowledge Engineering, *IEEE Trans on Software Engineering, Oct 1990, vol16, no 10*

4. Hirakawa M, Tamaka M, An Iconic Programming System - Hi Visual. *IEEE Trans on Software Engineering, Vol 17, No 10, Oct 1990*

5. Brolio J. Draper B.A. Beveridge J.R. and Hanson A.R, ISR: A Database for Symbolic Processing in Computer Vision, *Computer, December 1989*

6. Bart P.S. An Object Oriented Approach to Graphical Interfaces, *ACM Trans of Graphics, Vol 5, No 2, April 1986*

7. Jagadish H.V and O'Goramn L. An Object Model For Image Recognition, Computer, December 1989

8. Moldovan D.I. and Wu C, A Hierarchical Knowledge BAsed System for Airplane Classification, *IEEE Transactions on Software Engineering, vol 14 No 12 December 1988*

9. Orenstein J.A and Manola F.A, Probe Spatial Data Modelling and Query Processing in an Image Database Application, *IEEE Trans on Software Engineering, Vol 14, No 5, May 1988*

10. Joseph T, Cardenas A.F, Picquery, a High Level Query Language For Pictorial Database Management, *IEEE Trans on Software Engineering, Vol 14, No 5, May 1988*

11. Roussopoulos N, Faiatsous C, Sellis T, An Efficient Database System for PSQL, *IEEE Trans on Software Engineering, Vol 14, No 5, May 1988*

12. Chang S, Liu S. Picture indexing and Abstraction Techniques for Pictorial Databases, *IEEE Trans on Pattern Analysis and Machine Intelligence, Vol Pami-6, Vol 4, July 1984*

13. Ketabchi M.A, Valdis B, Modelling and Managing Cad Databases, *Computer, Feb, 1987*

14. Kasturi R, Fernandez R, Amlani M.L, and Feng W, Map Data Processing in Geographic Information Systems, *Computer, December 1990*

15. Lucas P. and Van Der Gaag L. Principles of Expert Systems, Addison-Wessley Publishers Ltd, 1991

16. Latombe J.C, Artificial Intelligence in computer aided design: the Tropic System, *CAD Systems (Proceedings of the IFIP Working Conference on Computer Aided Design, North Holland, February 1976) edited by J.J Allen 1977.*

17. Encarnacao J.L. *Incorporating Knowledge Engineering and Computer Graphics for Efficient and User Friendly Interactive Graphics Applications* Eurographics'85, Elsevier Science Publishers B.V. (North Holland)

18. Powers G.J. *Non Numerical Problem Solving Methods in Computer Aided Design* Proceedings of the IFIP Working Conference on Principles of Computer Aided Design, Eindhoven, the Netherlands, October 1972

19. Tibbert L. and Bergeron R.D. *Graphics programming for Knowledge Guided Interaction* Eurographics'85, Elsevier Science Publishers B.V. (North Holland)

20. Fox M.S. *Knowledge Representation for Decision Support* Proceedings of the IFIP WG 8.3 Working Conference on Knowledge Representation for Decision Support Systems Durham, U.K, 24-26 July, 1984

21. Stevens A. Roberts B. and Stead L. The Use of a Sophisticated Graphics Interface in Computer Aided Instruction, *IEEE CG&A, Mar-Apr 1983, pp 25-30*

22. Scar E. MITRE Co. Personal Communication on work in progress

23. Feiner S. Nagy J. and Van Dam A. An Experimental System for Creating and Presenting Interactive Graphical Documents, *ACM TOG, 1,1 (January 1982), pp 59-81*

24. McDermott J. An Expert in the Computer Systems Domain, *Proc. Nat Conf. Art. Intell. 1, pp 269-271*

25. Gossard D. Panel on Solid Modelling Siggraph '83, *Computer Graphics, July 1982, pp 163-164*

26. Neal D.B. Tutorial on Graphics and Data Bases, *Siggraph '83 Tutorial Notes 20, ACM, New York, N.Y. July 1983*

27. Garret M. and Foley J.T. Graphics Programming using a Database System with Dependency Declarations, *ACM TOG,1,2(April 1982), pp 109-128*

28. Gordon Extensions for Autocad User Manual (versions 217/218/2.5/2.6/release 9)

29. Browston L. Farrel R. Kant E. Martin N. Programming Expert Systems in OPS5 - An Introduction to Rule Base Programming

30. Alty J.L and Coombs M.C. Expert Systems - Concepts and Examples.

BIBLIOGRAPHIES

1. Tello E. Object Oriented Programming for Artificial Intelligence

2. Gorlen K.E. and Plexico P.S. Data Abstraction and Object Oriented Programming, John Wiley and Sons

3. Brodie M.L and Mylopoulos J. On knowledge Based Management Systems - Integrating Artificial Intelligence and Database Technologies, *Springer - Verklag Publishers*

4. Tenenbaum A.M. Langsam Y. Augenstein M.J. Data Structures Using C, *Prentice Hall International Editions.*

5. Zhang, Z.Z Hope, G.S. and Malik O.P. Expert Systems in Electric Power Systems - A bibliographical survey, *IEEE Transactions on Power Systems, Vol 4, No 4, October 1989*

6. Hansen, C. and Henderson, T.C CAGD - Based Computer Vision *Ieee Transactions on Pattern Analysis and Machine Intelligence, Vol 11, No 11, November 1989*

7. Vemuri, K.R, Oh, S. and Miller R.A. Topology - Based Geometry Representation to Support Geometric Reasoning, *IEEE Transactions on Systems, Man and Cybernetics, Vol 19, No 2, March/April 1989*

8. Hoeltzel, D.A. and Chieng, W-H, Knowledge - Based Approaches for the creative synthesis of mechanisms, *Computer Aided Design, Volume 22 number 1 Jan/Feb 1990*

9. Jayaram, S. and Myklebust A, Automatic Generation of Geometry Interfaces Between Application Programs and CAD-CAM Systems, *Computer Aided Design, volume 22 number 1 Jan/Feb 1990*

10. Rosenman M.A, Gero J.S, Hutchinson P.J, Oxman R, Expert Systems Applications in Computer - Aided design, *Computer Aided Design, vol 18 number 10 Dec 1986*

11. Steffik M.J, and De Kleer J, Prospects for Expert Systems in Cad, *Computer Design, April 21 1983*

12. Patel A, Soong N, Wirability Expert Systems, *Computer Aided Design, vol 18, No 9, Nov 1986*

13. Puliti P, Tascini G. Interpreting of Technical Drawings, *Computer Journal, Vol 33 No 5, October 1990*

14. Grosky W.I. and Mehtora R, Image Data Base Management, *Computer Magazine, December 1990*

15. Markowitz V, Makowsky J.A, Identifying Extended Entity Relationship Object Structures in Relational Schemas, *IEEE Trans on Computer Engineering, Vol 16, No 8, Aug 1990*

16. Mannino M.V, Choi J, The Object Oriented Functional Data Language, *IEEE Trans on Software Engineering*, Vol 16, No 11, Nov 1990

17. Kim W and Benerjee J. Operations and Implementations of Complex Objects, *IEEE Trans on Software Engineering, Vol 14, No 7, July 1988*

18. Fagan M.J, Expert Systems Applied to Mechanical Engineering Design - Experience with Bearing Selection and Application, *Computer Aided Design, pg 361, 1987*

19. Bell D.A, Shao J.A, Hull M.E.C, Integrated Deductive Database System Implementation. a Systematic Study, *Computer Journal, Vol 33, No 1, 1990*

20. Taylor M.C, Logical Optimisation of Distributed Knowledge Base Queries, *Computer Journal, Feb 1990*

21. Ketabchi M.A, Berzins V, Mathematical Model of Composite Objects and its Application for Organising Engineering Databases, *IEEE Trans on Computer Engineering, Jan 1988*

22. Deux O. et al, The Story of O2, *IEEE Trans on Knowledge and Data Engineering, Vol 2, No 1, March 1990*

23. Fisher R.B, Orr M.J.L, Geometric Reasoning in a Parallel Network, *International Journal of Robotics Research, Vol 10, No 2, April 1991*

24. Machinlay J, Automating the Design of Graphical Presentations of Relational Information, *ACM Trans on Graphics, Vol 5, No 6, April 1986*

25. Ayala D, Brunet P, Juan R, Navuzo I, Object Representation by means of Nominal Division Quadtrees, *ACM Trans on Graphics, Vol 4, No 1, Jan 1985*

26. Wileden J.C, Clarke I .A, Wolf A.C, A Comparative Evaluation of Object Definition Techniques for Large Prototype Systems, *ACM Trans on Programming Languages and Systems, Vol 12, No 4, Oct 1990*

27. Samet H. Webber R.E, On Encoding Boundaries with Quadtrees, *IEEE Trans on Pattern Analysis and Machine Intelligence, Vol Pami-6, No 3, May 1984*

28. Chock M, Cardenes A.F, Klinger A, Database Structure and Manipulation Capabilities of a Picture Database Management System (PICDEMS), *IEEE Trans on Pattern Analysis and Machine Intelligence, Vol Pami-6, No 4, July 1984*

29. Chang S.K, Yan C.W, Dimitroff O.C, An Intelligent Image Database System (IIDS), *IEEE Trans on Software Engineering, Vol 14, No 5, May 1988*

30. Kasturi R, Alemany J, Information Extraction from Images of paper Based Maps, *IEEE Trans on Software Engineering, Vol 14, No 5, May 1988*

Author: Gritzman Jonathan Larry.
Name of thesis: The use of expert systems in CAD.

PUBLISHER:
University of the Witwatersrand, Johannesburg
©2015

LEGALNOTICES:

Copyright Notice: All materials on the University of the Witwatersrand, Johannesburg Library website are protected by South African copyright law and may not be distributed, transmitted, displayed or otherwise published in any format, without the prior written permission of the copyright owner.

Disclaimer and Terms of Use: Provided that you maintain all copyright and other notices contained therein, you may download material (one machine readable copy and one print copy per page)for your personal and/or educational non-commercial use only.

The University of the Witwatersrand, Johannesburg, is not responsible for any errors or omissions and excludes any and all liability for any errors in or omissions from the information on the Library website.