A RULE-BASED 2D GEOMETRY SYSTEM FOR ENGINEERING DESIGN

By

Chen Yong Hua

September 1990

Thesis submitted in accordance with the requirements of the

University of Liverpool

for the degree of Doctor in Philosophy

The Department of Mechanical Engineering

Faculty of Engineering

The University of Liverpool

Liverpool, England

ABSTRACT

This thesis presents a rule-based 2D geometry system called LOGICAD. The implementation of the system is based on a novel geometric knowledge representation scheme for the purpose of reasoning about geometric elements. This scheme is called geometric element based reasoning (GEBR). The logical inference process in LOGICAD adopts both a *rule-cycle hybrid* method and a *forward reasoning* with *dependency-directed backtracking* method. A novel scheme presented in this thesis and implemented in the LOGICAD software uses coordinate system independent geometric constraints as the object geometry description instead of the conventional vertex coordinates description. A further novel aspect of the new scheme is that it separates and implements the symbolic reasoning processes and the numerical assignment processes in the geometric construction of an object thereby avoiding the usual requirement for the simultaneous numerical solution of nonlinear equations. Moreover, the scheme implemented in the LOGICAD software can be used as the basis for intelligent CAD systems development since geometric constraints can be intermingled with functional constraints.

The thesis discusses the ineffectiveness and inefficiency of existing CAD systems in the design process. Philosophies of engineering design are analysed and some possible approaches to achieve intelligent CAD systems are presented. The author suggests that intelligent CAD system development requires a different approach to geometric modelling systems than for conventional CAD systems, and that this can be based on the geometric constraint description of object shape.

This thesis proposes a geometric knowledge representation scheme which offers the facility for constraint based description of object shape. This scheme is implemented as the LOGICAD software in the logic programming language Prolog. The suitability of Prolog for geometric problem solving is studied with several examples such as edit function support modules, planar graph traversal, 2D polygon clipping and CAD data base management.

The development and elaboration of the geometric element based reasoning scheme has demonstrated significant advantages in automatic inferencing in geometric reasoning problems.

ACKNOWLEDGEMENTS

The work described in this thesis was carried out during the period from August 1987 to September 1990 under the guidance of Dr. A. T. Shenton. On this ocassion, I wish to express my deep gratitude to Dr. Shenton for his support, advice, patience and friendship during the course of this work.

I wish to thank my parents who because of this work I have not seen for so long and my wife who has encouraged me throughout.

Finally, I would like to take this opportunity to express my sincere thanks to the Ministry of Petroleum of the People's Republic of China and the Committee of Vice-Chancellors and Principals of the Universities of the United Kingdom for Sponsoring my work in the University of Liverpool.

Dedicated to my wife Ouyang Jingtao

TABLE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

LIST OF PROGRAMS

CHAPTER 1

INTRODUCTION

Existing CAD systems can assist many areas of design work but designers are not satisfied with what existing CAD systems offer. This chapter identifies aspects of the ineffectiveness and inefficiency of existing CAD systems when used in the design process. It is argued that artificial intelligence (AI) offer many features such as declarative knowledge bases, automated reasoning techniques and intelligent control methods which are suitable for intelligent CAD (ICAD) system research and development. AI-based systems can thus help the designer at a higher level of abstraction in the design process than existing CAD systems. In this chapter, possible approaches to achieve intelligent CAD systems are outlined. The advantages and disadvantages of current programming paradigms for ICAD applications are described. Finally, the role of geometry in engineering design system development is emphasised.

1.1 Development of Computer-Aided Design

Human beings have long been involved in the design process but the importance and the meaning of design is not fully understood even in the computer age. In the early 1970s, leading international management consultants concluded that

> since most of the expenditure in any organization occured in the manufacturing end of the business, and less than 5% at the design end, CAD would contribute little overall improvement and any investment would not be remunerated [Llewelyn 1989].

This naive conclusion may now be understood to be an inverted perception

> with the fact that 85% of the overall expenditure downstream is in reality determined at the design stage. The design stage is seen to be of critical importance to quality control and profitability [Llewelyn 1989].

The design stage has particular importance in an integrated CAD/CAM environment since

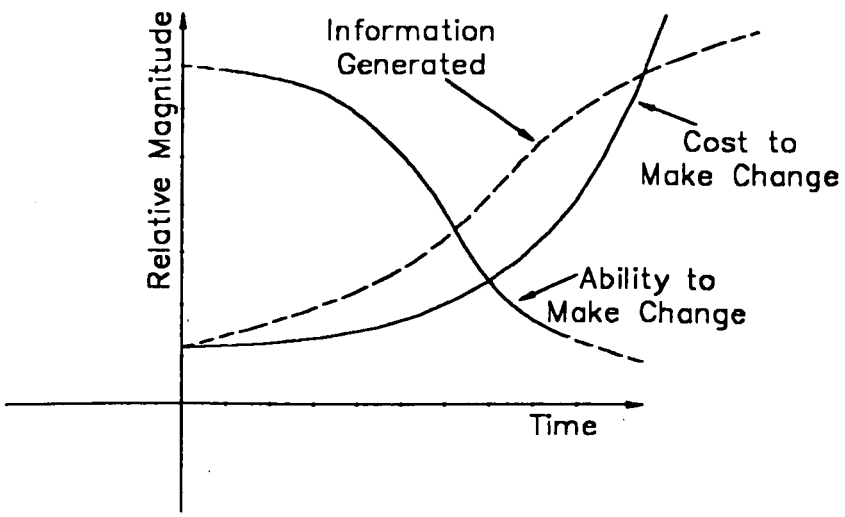
> The comprehensive use of CAD systems are extended over all relevant fields of application, ..., the data established within the CAD systems can be transmitted to other systems for manufacturing (CAM) or calculation (FEM) purpose [Schuster et al 1985].

The effect of design is illustrated in Figure 1.1 [Llewelyn 1989].

Computer-aided design (CAD) refers to the use of computers in the design process. Current CAD systems are of great help in drafting, finite element analysis, design evaluation, etc [Groover and Zimmer 1984]. The design data base can further be used by other application systems such as computer-aided manufacturing (CAM) and management. The productivity improvement resulting from the use of CAD are claimed as being very high, typically ranging from a low end of 3:1 to a high end in excess of 10:1 [Groover and Zimmer 1984].

The development of CAD has undergone three decades, but its wide applications have emerged only in recent years due to the dramatic reduction in hardware cost and the corresponding increased system performance. The

Figure 1.1 Aspects showing the importance of the design stage



progress of CAD development can be described in terms of three periods [Llewelyn 1989].

The first period was dominated by military research and development with Sketchpad [Sutherland 1963] being the first exciting demonstration of the possibilities in the civilian field. In this period, wireframe representation was used which enables systems to produce 2D drawing and model some simple 3D objects. For reasons of high cost and low hardware performance, initial developments were confined to a limited number of universities and research and development teams.

The second period saw the explosive growth in CAD. More advanced representation techniques such as boundary representation (B-rep) and construc-

tive solid geometry (CSG) were being used to unambiguiously model 3D objects. CAD systems could run on mainframes, local area network(LANS), and graphic workstations. Improvements were made in operating systems and computing languages and graphic tools with interfaces to link analysis and application packages were developed. Advanced but specialised CAD/CAM systems emerged for the electronics and microelectronics industries and for manufacturing with NC machines.

The third period emphasises the integration of design and manufacturing. Low cost personal computers are now widely used for CAD/CAM applications. At the same time, the first artificial intelligence and knowledge-based systems have become available for use in industry.

Recently, ICAD (intelligent CAD) has been a major area of CAD research and development. A small US software firm Icad [Hampshire 1988] markets its product *ICAD* which is able to

> capture rules, definitions and relationships, and thus automates a greater part of the design process [George 1988].

The advantage of *ICAD* is that

> it retains a record of the design and engineering process, of the rationale and calculation behind each decision [Hampshire 1988].

These features are made possible through the use of the artificial intelligence language LISP in *ICAD*. Although the author suggests that some of these claims are unjustified, it is nevertheless a significant product because it inte-

grates the CAD functions with expert system technology. More intelligent CAD systems which are under developments include *EDS* [Poppestone 1987], *KAUS* [Ohsuga 1989] and *IIICAD* [Veth 1987] etc.

After 20 years of practice, CAD has been widely accepted by engineers. Its use has covered areas like mechanical engineering, electronics, building construction, facility planing, etc. CAD related hardware and software are now important tools for the engineers.

1.2 Requirement for Intelligent CAD System

Obviously, designers expect the computer to do more work in a more intelligent way but current CAD systems have many deficiencies which discourage potential users. The methodologies and techniques underlying current CAD systems can not be used as the basis for the

> more powerful engineering design support systems required to more closely integrate computer-based support of the engineering design process [Smithers 1989].

Artificial intelligence, however, offers many powerful tools such as *formal* and *expressive* knowledge representation schemes, automated reasoning techniques and intelligent control methods etc [Smithers 1989]. Because of these features, artificial intelligence has received significant attention in engineering design [Smithers 1989].

1.2.1 Deficiencies of Current CAD Systems

Current CAD systems are interactive passive geometric modelling tools, enabling the user to build a precise and accurately scaled geometric model and drawing of the part he is designing. The functionality provided is for the creation, deletion and modification of graphic elements. These functions are useful in the documentation of a design, but of little help for the early stages of design [Shenton 1987 and Ohsuga 1989]. Many applications such as process planning, automatic group classification and manufacturability evaluation, etc can not be automatically implemented through the database of today's CAD systems [Shah and Rogers 1988]. Problems in applying CAD systems to engineering applications are summarized as below.

User Interfaces

Input of shape is one of the most important issues in computer-aided design systems. Many existing CAD systems use representation schemes such as boundary representation and constructive solid geometry or a combination of the two [Requicha and Voelcker 1982]. Practice in using a current CAD system requires the user has to go through rather cumbersome construction procedures to have shapes produced. This process often discourages potential users. As an example, the cubic shape shown in Figure 1.2 requires the explicit description of the connectivity of faces, edges, and the coordinates of each vertices. Table 1.1 and Table 1.2 are the representation required by the well-known *winged-edge* data structure [Baumgart 1965]. These are obviously tedious for the user to input. Although the input may be facilitated through the use of a *sweeping* operation (Figure 1.3) or boolean operation (Figure 1.4)

in CSG, shape input still remains a demanding task for the users since *sweeping* operation can only model limited object shape and boolean operations require the user to specify the precise location and orientation of every primitive object.

Data Base Completeness

CAD systems are used mainly for the definition and manipulation of geometric objects. Information regarding physical constraints, surface finish, tolerances, material properties, surface conditions etc are important aspects for the definition of mechanical parts, but these are not yet incorporated in CAD systems in a meanful way [Shal and Rogers 1988].

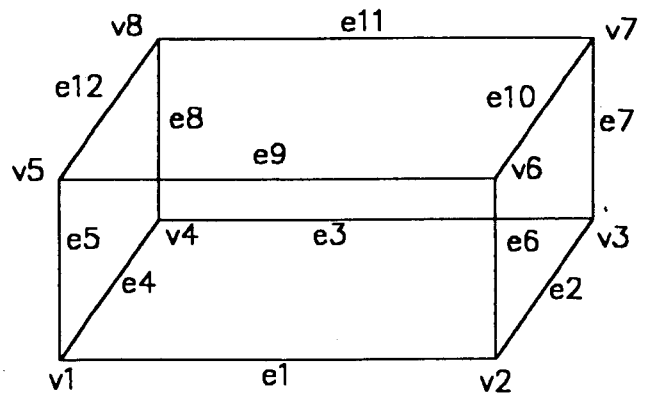
vertex	first-edge	coordinates	face	first-edge
v1	e1	x1 y1 z1	f1	e1
v2	e2	x2 y2 z2	f2	e9
v3	e3	x3 y3 z3	f3	e6
v4	e4	x4 y4 z4	f4	e7
v5	e9	x5 y5 z5	f5	e12
v6	e10	x6 y6 z6	f6	e9
v7	e11	x7 y7 z7	--	--
v8	e12	x8 y8 z8	--	--

Table 1.1 Vertices and face information for Figure 1.2.

Matching of Abstraction Levels

CAD systems store data in terms of low-level entities such as vertices, edges and faces etc or in binary trees that contain primitives and boolean operators.

Figure 1.2 A cubic shape described by winged-edge data structure

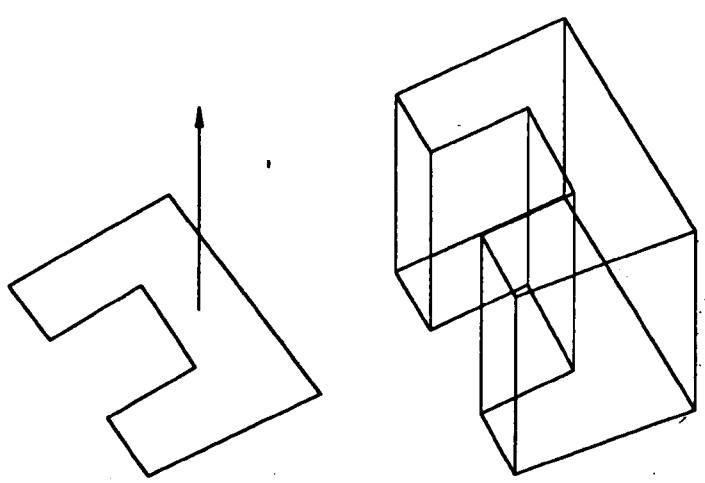


edge	vstart	vend	fcw	fccw	ncw	pcw	nccw	pccw
e1	v1	v2	f1	f2	e2	e4	e5	e6
e2	v2	v3	f1	f3	e3	e1	e6	e7
e3	v3	v4	f1	f4	e4	e7	e7	e8
e4	v4	v1	f1	f5	e1	e3	e8	e5
e5	v1	v5	f2	f5	e9	e1	e4	e12
e6	v2	v6	f3	f2	e10	e2	e1	e9
e7	v3	v7	f4	f3	e11	e3	e2	e10
e8	v4	v8	f5	f4	e12	e4	e3	e11
e9	v5	v6	f2	f6	e6	e5	e12	e10
e10	v6	v7	f3	f6	e7	e6	e9	e11
e11	v7	v8	f4	f6	e8	e7	e10	e12
e12	v8	v5	f5	f6	e5	e8	e11	e9

Table 1.2 Edge information for Figure 1.2: vstart = starting vertex, vend = ending vertex, fcw = clockwise face, fccw = counter-clockwise face, ncw = next clockwise edge, pcw = previous clockwise face, nccw = next counter-clockwise face, pccw = previous counter-clockwise face.

It is difficult to extract the engineering meaning of this data from the data base. For example, it is a non-trivial task to recognize from the CAD data base that a body is rotational and has a stepped through hole. Yet this is the level of information that many application programs operate on. Approaches have been proposed to extract features from the CAD data base [Henderson and Chang 1988, Ansaldi and Falcidieno 1988]. However, there is no algorithm that is universelly applicable.

Figure 1.3 Object description by "sweeping"

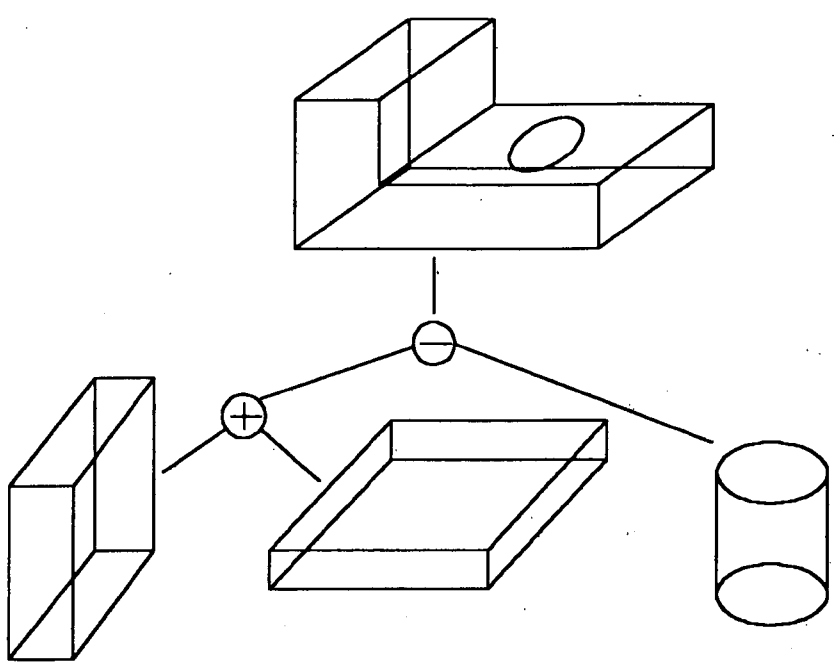


The Design Environment

CAD systems essentially only provide support for the detailed stage of design [Shenton 1987], that is, the user must know the basic design before a CAD system can be used because modifications are awkward and difficult. Sup-

pose a part has a hole on which a groove has been defined. The user wishes to reduce the hole diameter without changing the shape or depth of the groove. The user needs to first *fill* the space of the hole and groove, then to create the primitives for the cylinder and torus with new dimensions, and then to carry out the necessary boolean operations to create the new object. Design is an iterative process and thus requires numerous alterations in the model before a satisfactory solution is obtained. On the other hand, the design process involves a lot of technical information processing and management. Current CAD systems provide no support for such functions.

Figure 1.4 Constructive solid geometry



Despite these limitations, CAD systems are widely used to solve engineering problems, especially when the problems involve complex geometry and numerous computations.

In recent years, it has been identified that there is a strong need for the computer to support more of the design process in a more intelligent way. This has led to attempts to develop new representation schemes which can incorporate different kinds of knowledge and automated reasoning mechanisms. Artificial intelligence techniques are believed to be suitable for these applications [Smithers 1989].

1.2.2 What is Intelligent CAD ?

This is the question constantly asked and argued in the ICAD research community. The definition certainly involves two non-deterministic terms namely *intelligent* and *design*. *Intelligent* is defined by Bond as "involving symbolic representations and inference processes of some kind" [Bond and Soeterman 1987]. This is obviously not a complete definition of *intelligent* because the term *intelligent* is related to human beings. *Intelligent* should include

> feeling, creativity, personality, learning, freedom, intuition, morality, and so on [Haugeland 1985]

The creativity and originality of human beings can not yet be modeled by computers. What is possible at present however is to encode the way the human being solves certain problems.

Although people have long been involved in the practice of design, most of their concerns involve *how to design* [Tomiyama and Yoshikawa 1987]. Thus various design process models have been proposed. Figure 1.5 for example is the design process as described by Shigley [Shigley 1977]. Groover and Zimmer clarify for which design stages the existing CAD systems are of help in the design process based on Shigley's model (Figure 1.6) [Groover and Zimmer 1984]. ICAD systems should support the design process in a higher abstraction level than existing CAD systems since

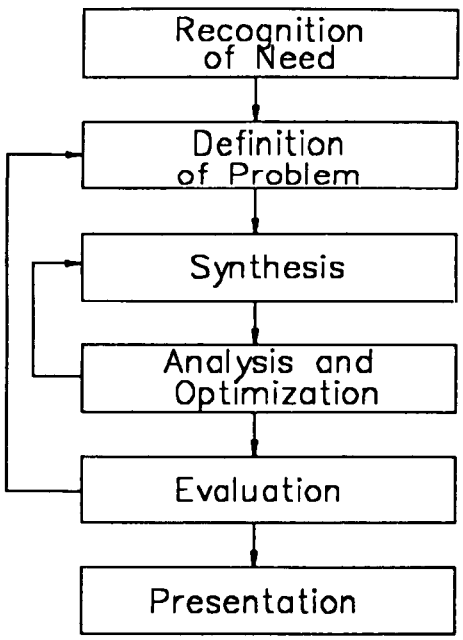
> The functional specification of a design problem is used as a starting point for the design process performed by an ICAD system. A designer supplies a functional specification to the system, and the system translates it to an initial design object model [Veerkamp et al 1990].

Figure 1.7 is the author's view of intelligent CAD systems in the design process based on Shigley's design model.

While the design process has been studied steadily, scientific definition of *what is design* has been almost ignored. Tomiyama and Yoshikawa proposed a general design theory based on a hypothesis [Tomiyama and Yoshikawa 1987]. They start with three axioms which are clarifications of the hypothesis, then deduce a set of 34 theorems. Whether or not this is a scientific theory of engineering design is questioned by some researchers [see discussion after Tomiyama and Yoshikawa 1987].

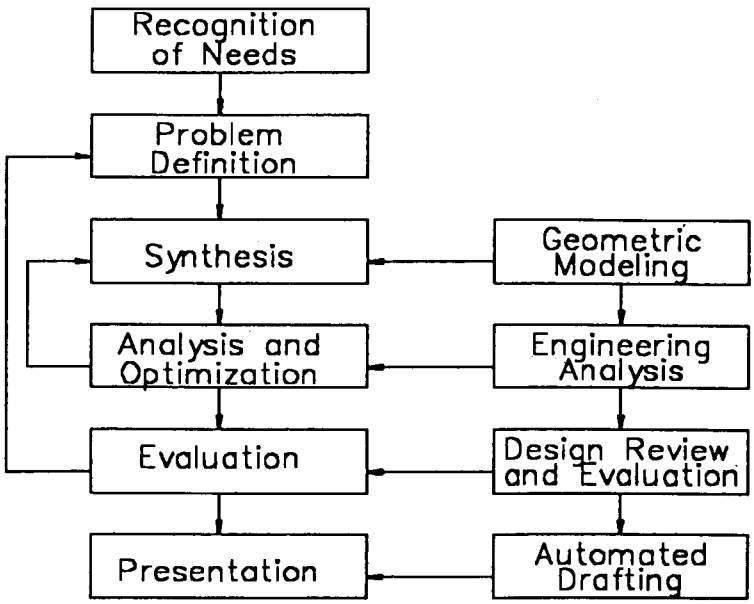
Nevertheless, from the above discussion, we can at least say that the term intelligent CAD is not uniquely defined. Current research in the field has split into the development of user interfaces [Bijl 1987, Ruttkay et al 1987], definitions of design process models [Mostow 1985, Keogel 1987], problem solv-

Figure 1.5 Design Process outlined by Shigley



ing strategies [Shenton and Chen 1990, Brumsen et al 1990] and improvements in geometric modeling systems [Arbab 1987 and Sunde 1987]. An ICAD system should obviously address all these aspects. It should also employ a reasoning capability that addresses design requirements and design specification assignment. Design specification assignment is the process of coding design knowledge in a machine understandable format such that it can be accessed and utilized to progress the design. Scheduling and planning strategies for propagating and satisfying design constraints are of fundamental importance in realizing solutions. Thus current research in intelligent CAD systems involves aspects which would cover a very wide set of natural human problem solving techniques.

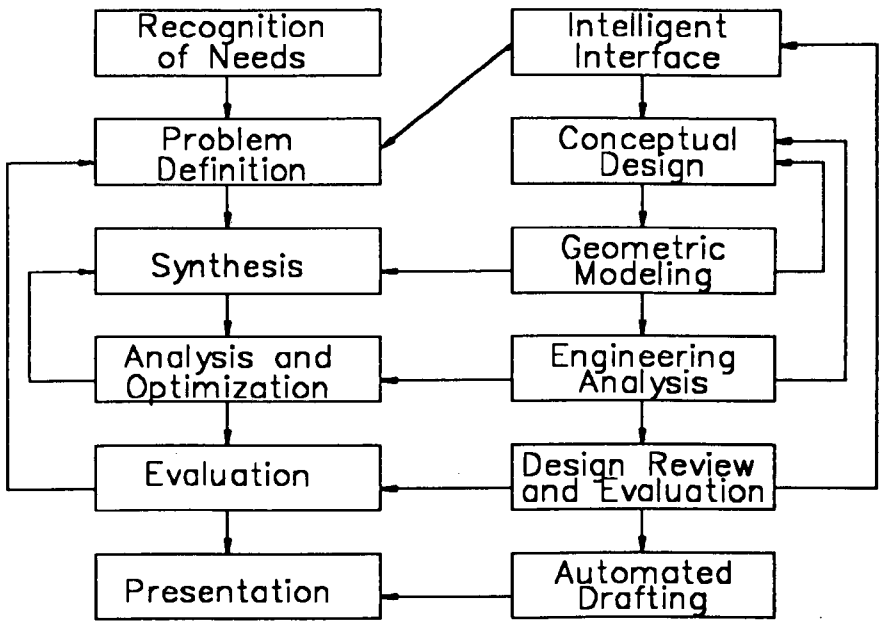
Figure 1.6 Design process extended by Groover and Zimmer



1.2.3 Feasibility of the Implementation of ICAD

Although the term ICAD is relatively new and ill-defined at present, we certainly expect the computer to play a greater part in the design process. Of course, computer systems may not fully realize all our expectations, but they do have the potential to do more of the work in the design process than existing CAD systems. Research in artificial intelligence has offered many tools to apply computers to the designer's needs. These techniques are conveniently categorised by four categories [Smithers 1989].

Figure 1.7 AI-based design based on the auther's view



Knowledge Representation

Engineering knowledge falls into two categories, *domain knowledge* and *task knowledge*. The *domain knowledge* is the knowledge used in the design process while the *task knowledge* is built up as a product is designed. *Formal* and *expressive* knowledge representation schemes have been devised to represent both kinds of knowledge. Here the word *formal* means

> an appropriate mathematical logic to which semantic meaning can be uniquely and consistently attached [Smithers 1989].

and the word *expressive* means that a representation scheme is able to

> represent a wide variety of types of objects and combinations of objects without becoming verbose in its construction [Smithers 1989].

A knowledge representation scheme has particular importance for ICAD system development because it is the basis for further knowledge processing techniques.

Automated Reasoning

There are two underlying kinds of reasoning techniques which are used to support the different types of problem-solving activities engaged in when the space of possible design choices is explored, and alternative designs are considered and filtered. The first are the forward reasoning types of techniques [Smithers 1989] that infer new knowledge from existing knowledge and identify inconsistencies by extending paths in the relational structure maintained in the task knowledge base. The second are the backward-chaining types of inference techniques [Smithers 1989] that are used to build subsystems to advise designers about particular aspects of design, or support the assessment, or analysis of a design.

Consistency Maintenance

As the design process proceeds, and a body of knowledge is built up in the knowledge base, the dependencies between elements of that knowledge and their consistency have to be recorded and maintained. This is so that, as new

knowledge is added or new design decisions and choices are made, the consequences that they have for all existing aspects of the design are considered at an early stage. Thus any constraint violation or inconsistencies introduced into the design description are resolved. The AI techniques currently most suitable for dealing with these type of problems are described as truth maintenance systems [Doyle 1979] and assumption-based truth maintenance systems [Nii 1984].

Intelligent Control

Large systems are often built up from smaller subsystems. The integrated control of both the invocation of and the interaction between a set of intelligent subsystems so as to deliver both flexible and uniform support to engineers engaged in a design process is currently an important field of AI research. There are a number of promising techniques whose extensions to meet these requirements are being investigated, such as blackboard control models [Nii 1986a,1986b] context layering techniques [Worden 1984] and multi-expert systems techniques [David 1987].

1.3 Approaches to ICAD

In the discussion sections of the First Eurographics Workshop on Intelligent CAD systems, there was an argument about the role of CAD systems in the design process. Tomiyama summarized the discussions as the two opposing standpoints:

1. The ICAD system as an intelligent assistant
2. The ICAD system as an automation tool for domain problems

The above division has long been debated in the CAD community and it seems that these views of ICAD surely will influence the development of intelligent CAD systems. Roughly speaking, the proponents of the first point of view believe that the more effective work the system can do in the design process, the more intelligent the system becomes. This has lead to the application of AI techniques to improve or enhance the capability of conventional CAD systems [Hampshire 1988, Bond and Soeterman 1987]. The proponents of the second point of view are trying to develop better design process models. In this case, researchers tend to build systems to support or automate the design tasks [Veerkamp et al 1990, Koegel 1987].

1.3.1 The Pragmatic Development of ICAD from Existing Systems

This approach basically deals with the integration of existing software packages and artificial intelligence techniques. Individual systems are often efffective in solving their specific target problems. Such systems have been constructed to address problems ranging from design simulation, evaluation, documentation, manufacturing process planning, NC code generation, management, etc. A practical approach to building more intelligent computer systems is through the

> meaningful integration of non-geometric and administrative information with geometric information into a knowledge base, incorporating real-world and manufacturing constraints into this knowledge base, and devising schemes to use this collection of knowledge to aid designers [Arbab 1987].

Such systems can be called *intelligent* in the sense that they are more *aware* of the real-world context in which design problems are to be solved.

A common method for implementing this approach is through the incorporation of an artificial intelligence language (eg. Prolog or Lisp) [Hampshire 1988, Popplestone 1986] or expert system shells [Hampshire 1988, David 1987] which are made responsible for knowledge representation, symbolic computation and constraint satisfaction. These systems often start by encoding expert knowledge, and these are then used to improve the entire product design, development and manufacturing process. This knowledge has to be elicited from engineers involved in design and manufacturing, and is usually but not always stored in the system as a set of rules. A collection of such rules forms what is known as a knowledge base. Mechanisms are devised to apply the knowledge base to solve particular types of design problems. Planning and scheduling are increasingly important as the knowledge base grows bigger. The commercial package *ICAD* [Hampshire 1988] and Bond's work [Bond and Soeterman 1987] are based on this approach.

1.3.2 Development of ICAD from Better Models of the Design Process

Apart from the pragmatic approach described above, many researchers argue that the key problem for ICAD system development is to develop better models of the design process [Mostow 1985, Veerkamp et al 1990]. The main effort for these researchers is therefore to develop appropriate design process models and associated computer languages for knowledge representation and processing. The significance of the design process model results from the argument that

A general design model is employed to define a system architecture and to develop language constructs for design representation languages. The model is used to build a framework which quides the design process as it is executed by the system in order to understand the designer's demands [Veerkamp et al 1990].

At present, there is no uniform definition of the design process model. According to Mostow [Mostow 1985], a design process model should address the following aspects of the design process

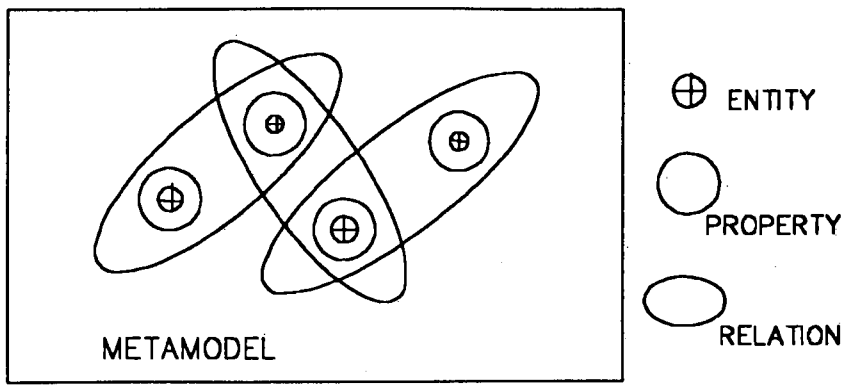
1. The state of the design
2. The goal structure of the design process
3. The design decisions
4. The rationales for design decisions
5. The control of the decision process
6. The role of learning in design.

The design process model is used to build an internal computer model which guides the design process as it is executed by the system. Veerkamp et al [Veerkamp et al 1990] propose a design process model as the mapping from the function space where the design object specification is described in terms of functions, onto the attributes space where the design solutions are described in terms of attributes as shown in Figure 1.8. Veerkamp et al's central description of the design object is by means of a *metamodel* as shown in Figure 1.9 [Veerkamp et al 1990]. A *metamodel* is a "context free description of the design object" which is represented as entities. The *metamodel* should also include the relationships and dependencies among these entities.

Figure 1.8 Design process defined by Tomiyama

The step-wise refinement process of design can be reflected in the changing states of the *metamodel* shown in Figure 1.10. In a certain stage of the design process the *metamodel* Mi-1 is the current, incomplete description of the design object. In order to get a more detailed description an *aspect model* is derived from the *metamodel.* Through this *aspect model* some new information about the design object is obtained. After this refinement the new information from the *aspect model* is merged into the *metamodel* Mi-1. If the merge is successful and fulfills the designer's requirements, then the result of the merge is a new state of the *metamodel* Mi. If for some reason, the *metamodel* Mi is unsuccessful and does not fulfill the designer's requirements, he may decide to redesign from a prior state. In this case, he backtracks to the previous *metamodel.* The process is continued until the design object model is a complete and satisfactory description of the desired artifact. This design process model emphasises the evolution process in design while some other

Figure 1.9 Metamodel



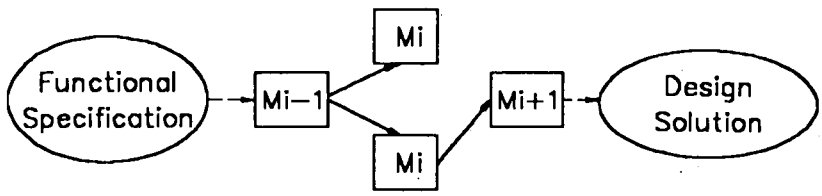
design models emphasise the functional taxomy and the mapping between functions to real world entities [Duhovmik 1987 and Popplestone 1986], or emphasise the "capturing, planing and scheduling of design information" [Hampshire 1988]

1.3.3 Implementation Languages

The appearance of new computing languages is propelled by the ineffectiveness of existing languages in solving difficult problems. For ICAD applications, a computer language should have

> language constructs that are naturally able to describe design processes and objects [Tomiyama 1987].

Figure 1.10 Stepwise refinement of the metamodel



Since conventional computing languages are not flexible and natural, new computer languages which incorporate new concepts such as the object-oriented programming paradigm and logic programming paradigm are considered in the development of ICAD systems. From years of experience, researchers have found that neither object-oriented programming nor logic programming alone are effective enough for ICAD applications [Tomiyama 1987, Arbab 1987, Veerkamp and Akman 1987]. A hybrid of the two are suggested [Tomiyama 1987 and Arbab 1987].

Object-Oriented Programming Paradigm

The object-oriented programming (OOP) paradigm is expected to play an important role in implementing ICAD systems because objects in OOP can be regarded as arbitrary objects in the world that the system is intended to

model. An object can possess both behavior and properties of the real world entity to be simulated. Simulation takes place by sending and receiving messages. Objects with common features are organized in an inheritance hierarchy which dramatically reduces the bookkeeping effort. Although promising, the object-oriented programming paradigm has revealed some fundamental problems as a possible kernel language for ICAD systems. Tomiyama [Tomiyama 1987] summarises the problems as follows :

1. Static relationships can not be expressed well using message passing
2. In order to represent static relationships, imaginary objects have to be created
3. The concept of class instance variables to represent inter- and intra-structural relationships is a problem because this forces objects to know about internal structures of other objects and results in a violation of the principle of information hiding
4. The inheritance of class instance methods and variables does not fit into the principle of information hiding because sometimes superclasses are completely foreign to subclasses

Logic Programming Paradigm

Engineering design is best described as descriptive rather than prescriptive methodologies [Bijl 1987]. The argument for descriptive methodologies in engineering design rests on the belief that design is a highly idiosyncratic human activity, and that consequently CAD systems should not impose definitions of domain objects and tasks upon design. Logic programming is by nature a descriptive paradigm and thus is often adopted for design applications. It is easy to represent the *is-a* and *part-of* or other relationships often required by design systems in a logic programming paradigm. Various inference methods which approximate human deduction can easly be implemented within a logic programming language [Shenton and Chen 1990]. When applied to the de-

velopment of ICAD systems, logic programming languages often suffer from their inability to handle numeric problems and from a lack of data-typing. This suggests a hybrid of the object oriented programming paradigm and logic programming paradigm.

Object-Oriented Logic

In ICAD practice, object-oriented programming and logic programming paradigms may be found to be complimentary [Arbab 1987, Veerkamp and Akman 1987]. A combination of the two may thus overcome most of the problems arising in ICAD system development. The benefits of this combination are claimed as very attractive in IDDL (Integrated Data Description Language) [Veerkamp and Akman 1987] and OAR (Objects and Reasoning) [Arbab 1987]. IDDL is a computer language developed mainly for ICAD application; OAR is developed to meet the demands of geometric reasoning. Both of the these two languages are a coupling of the object-oriented programming paradigm and the logic programming paradigm. The main issue here is to find the most suitable form for the combination. Other applications oriented combinations have also evolved, such as GRAFLOG [Pineda 1989], an augmented logic programming language with a set of geometric and topological functions; and GrafOLog (Graphic Object Logic) [Gloess and Guerin 1990], an interactive graphic system for expressing knowledge in first order logic by drawing objects and logic relations between these objects.

1.4 The Role of Geometry in ICAD

Engineers communicate ideas with both graphic drawings and verbal notes. The drawings are usually imprecise in the first place. But as the design process evolves, concrete shapes can be finalised. Thus the evolution process in design can in part be represented by the change of geometric information. Because of the important role of geometric information in design, a prerequisite for intelligent CAD systems is the more explicit representation and manipulation of geometric knowledge [Arbab 1987].

In order to automate more important features of the design work, it is clearly necessary to understand and to formalise the associated problems including the interdependence of geometric and non-geometric properties of a part, formal specifications of the functions of a part and the derivation of constraints defining its shape from such specifications, automatic verification of a design against its specifications etc. Until more is learned about how functionality affects geometry in general, the interplay of functionality with geometry will perhaps continue to be an activity internal to designers. This implies that exchanging explicit geometric infomation must continue to be an integral part of the interaction between designers and future CAD systems.

The intellectual level of communication between conventional CAD systems and their users is very low. These systems are primarily repositories of geometric information that keep track of the information supplied by a user and perform very specific, well-defined tasks on demand. Clearly CAD systems that support a higher level of user - system interaction are desirable. The current practice in ICAD is to limit the scope of a system to special cases

where the knowledge about how functionality affects geometry can be explicitly encoded [Duhovmik 1987 and Popplestone 1986]. With this knowledge at its disposal, the system can communicate with its users at the level of functionality.

1.5 Organization of the Thesis

This thesis describes the development and implementation of an interactive 2D rule based geometry system called LOGICAD which is entirely written in the logic programming language Prolog. The thesis is organized into 8 chapters. In Chapter 2, current development in ICAD systems are reviewed. The techniques for constraint based geometric modelling are described. A rule based approach for constraint based geometric modelling is proposed and developed. Chapter 3 describes the hardware and software of the implementation and their communication. Applications of Prolog to CAD data base management and some graphics problems including a novel logic programming approach to clipping are discussed and presented in Chapter 4. Chapter 5 is an anotomy of the LOGICAD system where each module is described in detail. The reasoning processes of the system which are based on a novel geometric knowledge representation scheme called geometric element based reasoning (GEBR) are described in Chapter 6. Extensions and potential extensions to the proposed logic programming scheme are discussed in Chapter 7 which include 3D geometric modelling, mechanism simulation and conceptual design. The last chapter is a discussion and conclusion.

CHAPTER 2

CURRENT APPROACHES TO ICAD

Current intelligent CAD system research and development is mainly based on existing CAD systems and artificial intelligence techniques [Popplestone 1986, Myers and Pohl 1990]. However, problems arise since existing CAD systems can not generally not be integrated with the design constraint resolution process in a meaningful way. In other words, as a design evolves, existing CAD systems can not adjust to the evolution process of the design. On the other hand, declarative constraint based geometric modelling systems treat geometric dimensions and relations as constraints. These geometric constraints may be integrated with design constraints. Therefore, the evolution of the design process involves the evolution of geometric constraints. This chapter reviews current intelligent CAD system development with three principal examples. The connection between intelligent CAD systems and constraint based geometric modelling systems is discussed. Current approaches to constraint based geometric modelling systems are surveyed. The often confusing notions of variational geometry and parametric design are compared under their application domain, computational method, computational load and constraint propagation method. Finally, a rule based method for geometric modelling is proposed and developed.

2.1 Intelligent CAD System Review

As discussed previously in Chapter 1, conventional CAD systems work only as a passive geometric modelling tool. Intelligent CAD systems, however, should improve the entire product design, development and manufacturing process. This section surveys three existing intelligent CAD systems. More existing systems are listed in Table 2.1 under five categories.

2.1.1 ICAD

ICAD [George 1988] is a commercial product marketed by Icad Inc. (Cambridge, Mass.) from 1985. Its main purpose is to automate the design and manufacturing of complex objects through knowledge-based expert systems. The first step in using *ICAD* to build a model is to structure the problem into its logic component parts in the form of a tree structure as shown in Figure 2.1. A table is composed of a top, a drawer, and four legs. The system refines the structure using rules and definitions from an initial *kind-of* approximation to one that looks and behaves correctly. The rules or definitions may be refined or new rules added during the design process. The behavior of *ICAD* is rather like a parametric design system. Indeed, *ICAD* works on captured rules, definitions and relationships gained in developing and manufacturing similar products such as pumps, gearbox, hydraulic cylinders and so on. The process of encoding knowledge is not easy even for a small company with simple products. One of the earliest users of the *ICAD* system was reported to be an American heavy engineering company, Hudson Products Inc. [Hampshire 1988]. This company manufactures large air cooled heat

exchangers. This is a very straightforward product but which needs to be customised to each user's individual requirements.

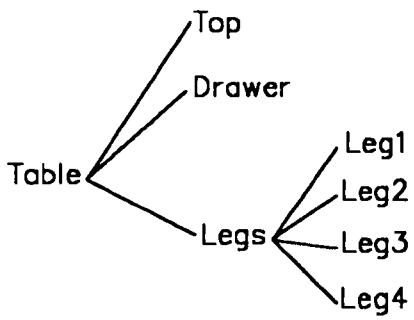
Name	Knowledge rep	Domain knowledge	Inference & Control	Application	Main language
ICAD	DKB,rule relation	Geometry technical	expert system	design& manufacture	Lisp etc
Topology-1	DKB rule,fact	geometry topology	pattern matching	architecture	Prolog
ICADS DEMO 1	rule,fact relation	geometry technical	DBD	architecture	C,ect.
KAUS	DKB,proc. rela.	geometry technical	STR	design	MLL
EDS	DKB,FA rela.	geometry function	forward chaining	design	Prolog Pop-11
Thesys	object rule	geometry	expert system	SAD	Poplog
BiCad	DKB rule	product model	forward chaining	architecture design	Xlisp
Sidesman	rule rela.	geometry technical	forward chaining	VLSI	DRDL

Table 2.1 A list of intelligent CAD systems: Proc. - procedural database, FA - Functional Taxonomy Rela.- relational database, DKB - Declarative Knowledge Base, DBC - Distributed Blackboard Control, SAD - Spatial Arrangement Design,MLL - Multilay Logic, DRDL - Design Rule Description Language

The necessary knowledge-base for the product took the company over three man-years to create and was composed of over 7000 rules governing the design of the heat exchangers which themselves are represented in the database as consisting of over 120,000 CAD objects. The result is reported to be that the time taken to produce a customised design and quotation has dropped from 3 months to less than a week. At the same time the percentage of error in quotations is reported as being virtually eliminated. Further *ICAD* system

users are reported to include some well established companies such as McDonnell Douglas, Kodak and Jaguar.

Figure 2.1 Logical components of a table



The main advantage of the *ICAD* system is its ability to incorporate nongeometric information (such as manufacturing, cost etc.) with geometry. The *ICAD* system can not assist the designer in the early stages of design. Furthermore, using the *ICAD* system, the designer must know the design and its possible variants and document the design in terms of rules. It is sometimes difficult to maintain a consistent set of rules because every part (or feature) of the design must be described by certain rules. When a design model is created, only variants of the design model can be generated. Clearly the use of *ICAD* sometimes may not be justified because of the fast changing market direction and the great effort needed to create a design model in *ICAD*.

2.1.2 ICADS DEMO1

The ICADS DEMO1 [Myers and Pohl 1990] project is based on the definition that the design process is "a reflective conversation with the situation". Thus the role of CAD is basically a "language of designing". With this understanding or view of design in mind, the ICADS DEMO1 system is developed to "understand the language of designs and converse with the designer as an intelligent assistant".

The development of ICADS DEMO1 involves a lot of different software and hardware facilities. An external architectural design database is implemented in a relational database system SQL-RT. A multiple CLIPS expert system shell is used for knowledge base inference. The drawing functions are provided by the MountainTop CAD package. Other coding work is done in the C procedural programming language. The DEMO1 runs as a set of concurrent processes on a network of three IBM-RT workstations running the AIX operating system. Three high resolution color monitors provide the displays featured in the interaction with the designer. A distributed blackboard control system is developed for the overall system control.

DEMO1 is limited to both a *vertical* and a *horizontal* slice of the design universe. The *vertical* level of DEMO1 deals with the development of schematic floor plans which

> lie in the middle of the conceptual design universe, preceded by more abstract work with spaces and functionality.

The *horizontal* level of DEMO1 deals only with structural, thermal, acoustic and daylighting factors of the design problems.

ICADS DEMO1 is still undergoing development. The objects handled and the application domain are limited to schematic floor plans. The only geometric operations supported are adding, stretching, deleting and moving various objects which are only relevant to the creation of object geometry.

2.1.3 Topology-1

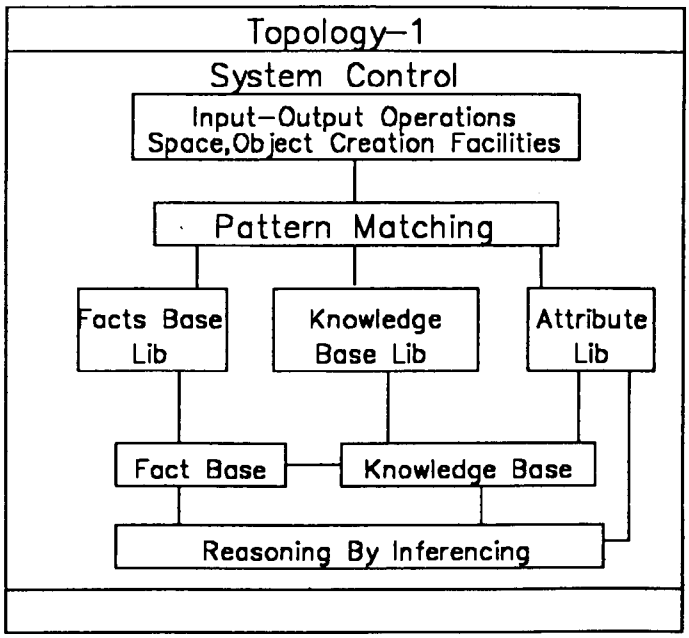
Topology-1 is a knowledge based system for reasoning about objects and spaces in buildings [Akiner 1986]. It encapsulates knowledge about topology, geometry and attributes of objects and spaces into five components which are under a system control facility with the form as shown in Figure 2.2. The inference process is monitored through pattern matching. Knowledge about topology and geometry are represented in an abstract manner such as **adjacent to, above**, and **shorter than** etc. For instance, the **above** rule can be stated as an IF...THEN rule:

IF A is higher_than B &
A overlaps B along z-axis
THEN A is above B

Its Prolog representation is given as

```
r(above,A,B):-
    r(higher_than,A,B),
    r(overlap,A,B,z_axis).
```

Figure 2.2 Structure of Topology-1



The Topology-1 system is virtually a query language about topologies, quantities, paths and names of objects and spaces in architectural design. No geometric modelling facility is provided within the system. The application domain of Topology-1 is very limited. Efficiency regarding the inference process has not been discussed in the development reports.

2.2 Graphical Programming in Prolog

It is obvious that most computer languages need the capability of graphical input and output. This is perhaps evident from the fact that more and more

user-computer interactions are through graphical interfaces and many (perhap most) applications involve graphical presentation and manipulation.

As a logic programming language, Prolog has been widely accepted for symbolic computation. Efforts have been made by some researchers [Krishnamurti et al 1987, Milanese 1988 and Hubner at al 1986] to integrate the GKS [ISO7942 1985] with Prolog. Krishnamurti et al have provided a complete Prolog/GKS binding [Krishnamurti et al 1987] which outlines how abstract function names and data types can be mapped to Prolog predicates and terms. Hubner et al [Hubner et al 1986] introduce a special graphical data structure called "picture" which consists of two terms. The first term is the picture name. The second term is a list of picture components which may be both graphical primitives and geometric transformations in the form of executable goals. Milanese [Milanese 1988] proposes a hierarchical tree structure for picture description based on GKS functions. The leaves of the tree are in the form of GKS output primitives.

Gonzalez et al [Gonzalez et al 1984] apply Prolog to solve some 3D CAD problems. Their main purpose is to evaluate the effectiveness of Prolog for CAD applications. A comparision is made with Pascal to solve the same problems. They conclude that

> Prolog is not only more concise, more readable, and clearer than Pascal, but also more efficient, ... Furthermore, Prolog programmes were developed more quickly and with minimum error.

Graphics applications are traditionally procedure-oriented [Hubner and Markov 1986]. This is also true for most of the existing CAD systems [Arbab

1987]. To construct a geometric model using CAD is like programming using a procedural language, the users first have to know the precise procedures and then a step-by-step method is used to construct the object. The effort required to master and operate a CAD system tends to discourage potential users. In addition to knowing the CAD commands, the user needs to have a good knowledge of geometry. In order to study the suitability of Prolog for CAD applications, various geometric algorithms such as a 3D CAD interface [Gonzalez et al 1984], convex-hull calculations, planar graph traversal, recognition of groupings of objects, boolean combinations of polygons, and cartographic map overlay have been implemented in Prolog [Franklin et al 1986]. More general design systems based on Prolog have also appeared in recent years [Akiner 1986, Popplestone 1986]. Using Prolog, it is easier to solve ill-defined problems (like design) in a knowledge-rich environment and constraint-based programming in Prolog has been proved to be practical [Bruderlin 1986, 1987].

2.3 Constraint-Based Programming

Design is constraint oriented [Serrano and Gossard 1988]. Constraints are continually being added, deleted and modified throughout the design process. Since constraints involved in engineering design are diversified, current constraint programming systems can only solve a small domain of design problems. But constraint based programming systems have some advantages against conventional systems.

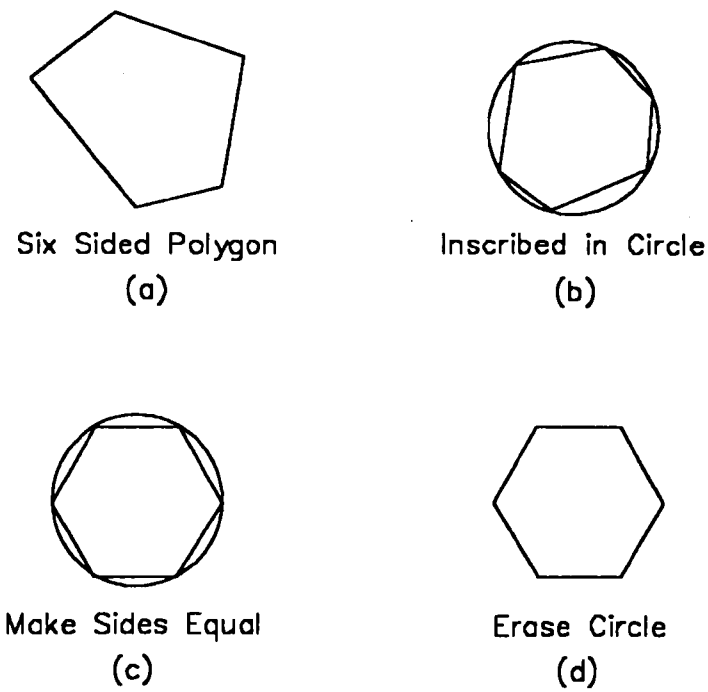
Using procedural languages such as C or Pascal, the effort required to program tends to discourage programming and thus effectively restrict most us-

ers to canned application programs [Leler 1988]. In a constraint-based programming system [Borning 1981, Nelson 1985], however, the user needs only to specify a set of relationships between the composing objects. It is the job of the constraint satisfaction system to find a solution that meets these relations. Since the specific steps used to satisfy the constraints are largely up to the discretion of the constraint satisfaction system, the user can build geometric objects with less regard for the construction sequences. This section presents a brief examination of three of the existing constraint-based systems. It can be seen that all the three examined systems use numerical procedures such as a relaxation method in Sketchpad and ThingLab, and the Newton-Raphson method in Juno. A common problem of numerical methods is stability. For instance, a numerical method may fail to terminate even when the constraints have a solution, or may find one solution arbitrarily for constraints with more than one solution. One possible approach to overcome the numerical problems is to use a rule-based method for constraint satisfaction [Leler 1988]. The scheme implemented in this thesis is a non-numerical rule based approach.

2.3.1 Sketchpad

Sutherland pioneered constraint-based geometry in his original work on Sketchpad [Sutherland 1963], in which nonlinear constraint equations were used to establish the location of geometric entities. Sketchpad's use of constraints, a sophisticated interactive graphical interface, a macro facility, and instancing features were years ahead of its time. Using Sketchpad, the user could draw a complex object by sketching the topology and then adding constraints to it.

Figure 2.3 Draw a hexagon in Sketchpad



For example, steps required to construct a regular hexagon in Sketchpad are shown in Figure 2.3. Sketchpad used a light pen as its location input, and a set of function buttons as its "menu". Firstly, a closed irregular hexagon is sketched (Figure 2.3(a)). The vertices of the polygon are then constrained to lie on the circle and to be of equal length (Figure 2.3(b)). Lastly, the circle is deleted, leaving the desired regular hexagon (Figure 2.3(c)).

The primitive constraints include requiring two lines to be parallel, perpendicular, or of equal length etc. Constraints wrere represented internally as error expressions that evaluate to zero when constraints are satisfied. Sketchpad satisfied constraints by first using the *one pass* method which finds an order by which

the variables of a drawing may be re-evaluated to completely satisfy all the conditions on them in just one pass [Sutherland 1963].

When the *one pass* method failed, the relaxation method was employed.

Sutherland made use of constraints as a design aid in the creation of a part geometry but did not use geometric constraints for the definition of the complete geometry nor in modification of part geometry after it was created. Sutherland also used what was at that time expensive graphics input and display hardware. The use of the Sketchpad system was limited to a small community.

2.3.2 ThingLab

Borning's ThingLab is a constraint-oriented programming language [Borning 1981]. It is an extension to the Smalltalk-76 system which is an object-oriented language environment. The main purpose of ThingLab is to support interactive graphic simulation of experiments in physics and geometry. Constraints in ThingLab are represented as rules and a set of methods that can be invoked to satisfy the constraints. Rules are used by the system to construct a procedural test for whether or not the constraints are satisfied and to construct an error expression that indicates how well the constraints are satisfied. The methods describe alternate ways of satisfying the constraints.

Since constraints are typically multidirectional, it is not always easy to meet the speed requirement for constraint satisfaction [Borning 1981]. To speed up the process, ThingLab again uses the *one pass* method which incrementally analyses constraint interactions and compiles the results of this analysis into

executable code. When this *one pass* method fails, the relaxation method is used. This is reminiscent of the Sketchpad system.

A major limitation of ThingLab is that it requires the user to have the skills of object-oriented modelling. This requirement may discourage potential users. In avoiding this difficulty, Borning expects that classes would be defined "by an experienced user of the system". Using these classes, "a less sophisticated user could then employ them in constructing a simulation". However, class definition may often need modification depending on applications. This obviously presents a difficulty to users. ThingLab also inherits the known defects of numerical methods.

2.3.3 Juno

Juno is a system that integrates a what-you-see-is-what-you-get(WYSIWYG) image editor with programmability [Nelson 1985], that is, when you use the image editor to construct a geometric object, a text programme corresponding to the geometric object is implicitly produced.

The underlying Juno language is relatively simple. Variables representing points, shapes and edges are created as the side effect of procedures parameterized by the data type *point*. There are only four types of constraints available in Juno, they are:

1. **(X,Y) CONG (U,V)**, the distance from X to Y equals the distance from U to V
2. **(X,Y) PARA (U,V)**, the direction from X to Y parallels the direction from U to V

3. **HOR(X,Y)**, the direction from X to Y is horizontal
4. **VER(X,Y)**, the direction from X to Y is vertical.

Since the constraints PARA and CONG introduce quadratic equations, the Juno constraint solver uses Newton-Raphson iteration. But a problem arises, since this method requires an initial guess which must be approximately in the true position. The constraint solver only aligns the geometric elements accuratly, and does not rearrange the configuration drastically. In other word, the user needs to have the approximate image in his mind when using the system. This severely limits the application of Juno. Moreover, the data objects (points) and constraint types that Juno support are clearly very limited. The author suggests that the main contribution of Juno is the idea of automatically constructing a constraint programme from a graphical interface.

2.4.Variational Geometry

Hillyard and Braid [Hillyard 1978, Hillyard and Braid 1978] have proposed a general theory which uses geometric constraints between vertices of a part to study the relationship between variations in dimensions (i.e tolerances) and variations in part geometry. Their approach is to regard a part as an engineering frame structure whose members and joints correspond to the edges and vertices of the part. The members are initially unconstrained in length. Adding dimensions is analogous to adding struts (fixing distances), webs (fixing angles) and plates (fixing planes). They use the rigidity matrix (the matrix of partial derivatives of the constraints equations) to compute the influence on part geometry of variations in dimensions.

In the case of a single dimensional change, often only a subset of the total constraints will be affected. Thus only a subset of the total constraining equations will be required to solve for the changed geometry.

Efforts have been made by several researchers to increase the computational efficiency by isolating the minimum subset of the constraint equations affected by a given dimensional change. Lin et al [Lin et al 1981] use backward and forward propagation of coordinates to derive the sensitive equations and coordinates of a given dimensional change. They also built an interactive CAD system based on variational geometry for 2D and 3D drawing. Light and Gossard [Light and Gossard 1983] use row operation and column permutations to analyse the Jacobian matrix and eventually to detect over-, under and incorrectly constrained situations in a given dimensioning scheme for 2D geometric parts. The problem of computational efficiency is also addressed by them.

As design is an iterative process, the geometry of the designed part should reflect changes as the design evolves. Variational geometry can provide this facility. Users may define the general shape of the part by free-hand sketch input. Then as the dimensional values are refined, the system can compute the exact geometry by solving a set of nonlinear constraint equations.

In the late 1980s, commercial systems based on variational geometry have been introduced by Advanced Graphics System (Tulsa, Okla.), Cognition Inc. (Billerica, Mass.), Iconnex (Pittsburgh, Pa.), MCAE Technology Inc. (San Jose, Calif.), and Premise Inc. (Cambridge, Mass.) [CIME 1989]. These systems may be roughly characterized as optimization tools for conceptual

design and are commonly used for mechanism design, beam deflection analysis, tolerancing, and other tasks. The method proposed in this thesis is actually a rule based approach to variational geometry. Thus it has all the merits offered by variational geometry but is not hindered by the associated numerical problems.

The basis of variational geometry is simple. The geometry of an object is considered to be determined by a set of N characteristic points in 3D space. The complete set of characteristic points is described by a geometry vector **x**, containing the Cartesian coordinates of all the points.

$$\mathbf{x}^T = [x_1, x_2, ..., x_n]. \tag{2.1}$$

Where n is equal to 3N. Dimensions which are treated as constraints limiting the permissible location of these characteristic points are stored in a dimension vector **d**

$$\mathbf{d}^T = [d_1, d_2, ..., d_m]. \tag{2.2}$$

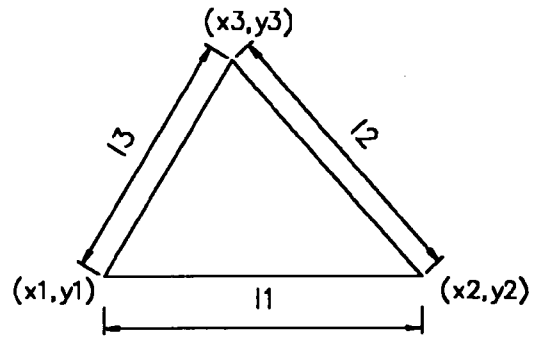
The constraints may be expressed analytically by nonlinear equations of the following form

$$f_i(\mathbf{x}, \mathbf{d}) = 0, \; i = 1,2,...,m \tag{2.3}$$

Where m = the number of constraints. Constraint equations are stored in a function vector **f**

$$\mathbf{f} = [f_1, f_2, ..., f_m]. \tag{2.4}$$

Figure 2.4 Constraint equation for a triangle



f1: x1=0
f2: y1=0
f3: y2−y1=0
f4: x2−x1−l1=0
f5: (X3−X1)**2+(y3−y1)**2−l3=0
f6: (x3−x2)**2+(y3−y2)**2−l2=0

The Jacobian **J** is an (m*n) matrix containing the partial derivatives of each constraint equation with respect to each degree of freedom. It expresses the relationship between variations in dimensions and variations in coordinates.

$$\mathbf{J} = \begin{bmatrix} f_{11} & f_{12} & \cdot & \cdot & f_{1(n-1)} & f_{1n} \\ f_{21} & f_{22} & \cdot & \cdot & f_{2(n-1)} & f_{2n} \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ f_{(m-1)1} & f_{(m-1)2} & \cdot & \cdot & f_{(m-1)(n-1)} & f_{(m-1)n} \\ f_{m1} & f_{m2} & \cdot & \cdot & f_{m(n-1)} & f_{mn} \end{bmatrix} \qquad (2.5)$$

where

$$f_{ij} = \frac{\partial f_i}{\partial x_j} \tag{2.6}$$

When a dimensional change is made, the geometry vector **x** is no longer consistent with the values of the dimensional vector **d**. Specifically, the residuals $\mathbf{f}(\mathbf{x}, \mathbf{d}) \neq 0$. A common approach is to use the Newton-Raphson method. The current geometry vector **x** is used as the initial estimate. The new estimate **x**′ is generated by the equation

$$\mathbf{x}' = \mathbf{x} - \mathbf{J}^{-1}\mathbf{f}(\mathbf{x}, \mathbf{d}). \tag{2.7}$$

The iteration continues until the residuals $\mathbf{f}(\mathbf{x}, \mathbf{d}) \neq 0$ vanish. When this occurs, the geometry **x** is consistent with the values of the modified dimensions **d**, and the part geometry is updated.

An example is given in Figure 2.4. A triangle is constrained by l1, l2 and l3. Suppose that based on the constraints, six constraint equations (f1 to f6) are established. From the constraint equations, it can be seen that a numerical method for the nonlinear equations has to be used. Variational geometry uses the Newton-Raphson method to simultaneously solve the set of nonlinear equations.

Although the Newton-Raphson method is more efficient than other numerical methods for most cases, there are situations where it performs poorly. Divergence from the root and oscillation around a local maximum or minimum may occur with some iterations. The only remedy in these situations

seems to be to have an initial guess that is close to the root. That is, with variational geometry, drastic changes of dimensions bring the danger of divergence in the nonlinear equation solver. This approach also presents a number of other problems. These include large computational requirements, non-uniqueness of the resulting geometry, user difficulties in creating and maintaining a set of consistent constraints and difficulty in 3D applications.

2.5 Parametric Design

While the overall productivity obtained with traditional CAD/CAM systems may vary depending on the applications, some of the highest levels of productivity have been achieved with the aid of special programming languages that automate repetitive tasks [CIME 1989]. These languages allow engineers to create a program or a *macro*, usually called a script, directing the CAD systems to perform a particular sequence of instructions for creating geometry. Designers can then supply dimensions and the script generates the appropriate geometric entities.

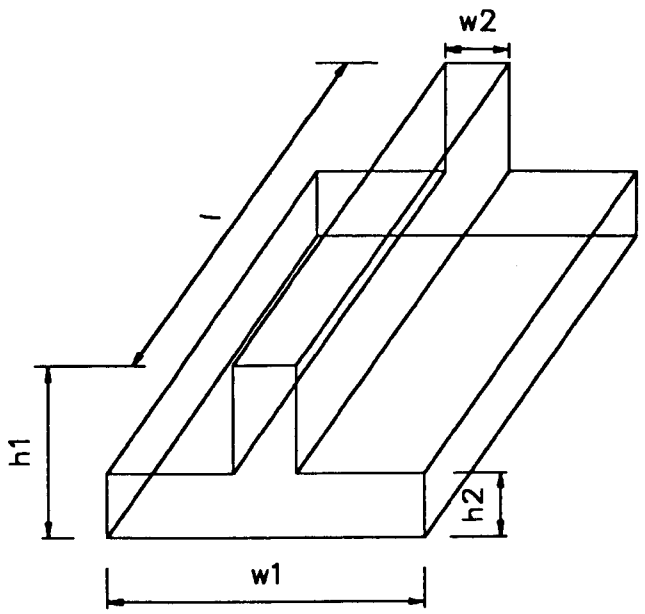
Consider, for example, the parametric object in Figure 2.5 which is defined by the five parameters l, h1, h2, w1 and w2. Its representation may be in the form of the 6-tuple

< tbrick,l,h1,h2,w1,w2 >

where tbrick is the identifier for the part family.

To be a valid representation, the parameters should satisfy the following conditions:

Figure 2.5 Parametric description of a t-brick



$l > 0$

$h1 > 0$

$h1 > h2 > 0$

$w1 > w2 > 0$

$w2 > 0$.

Once the parametric model has been defined, it is easy to change dimensions and relationships. But the creation of a parametric model may prove to be a difficult task. Various attempts have been made to facilitate the creation of parametric models [Cugini et al 1985, Maarten 1986].

Parden and Newell [Parden and Newell 1984] use the variational geometry technique to provide a graphical interface. In their system, a drawing or object is dimensioned in the normal way. The dimensions are interpreted as constraints on the object, and the system then automatically modifies the geometry of the object to satisfy these constraints. This approach is obviously superior to parameterised scripts because it uses the draftman's natural language: dimensioning. However, it is difficult to extend the method to 3D applications. Some numerical problems inherented by variational geometry also limit its application.

An algorithmic language for the definition of parametric objects is proposed by Cugini et al [Cugini et al 1985]. The language has four functional parts which allow the definition of primitives, manipulation of primitives, modularization of objects, correctness checking and implementation of the language. This approach requires the user to learn the language before he can create a parametric model.

To extend parametric systems to wider applications, Kawageo and Managaki [Kawageo and Managaki 1984] propose a parametric object model which has the following features:

* Parameterised dimension manipulation ability
* High-level independency
* Motion representation ability

This proposal is implemented in a geometric modeling system called GEODERM [Krishnan and Patnaik 1986].

The modeling space of the parametric design is entirely determined by the "part library" created. Obviously, it is difficult to incorporate all the necessary part definitions in a limited memory space in this approach.

The determination of a parametric model usually includes the following procedures:

1. The type and position of reference elements. These are the bases for positioning other elements
2. The geometric relationship between the elements (parallelism, angle, tangency, etc)
3. The parameters to be parameterised
4. The relations between the parameters
5. The limits on the parameter values
6. The relations between the dimensions and the parameters (nominal values of dimensions and tolerances)
7. The limits on the nominal values of dimensions

2.6. Comparision of Variational Geometry and Parametric Design

The difference in the meaning of the terms parametric design and variational geometry is often confusing. The confusion may arise because both approaches appear to proceed according to the same principle: each allows the user to create geometry by entering variables (typically dimensions and relationships) and then by invoking operations to update the geometry. That is,

they both allow the user to create and iterate the design and to observe the changes from one iteration to another.

Despite the similarities, they differ substantially in a number of ways, and in particular in the following:

1. The application domain
2. The computational method
3. The computational load
4. The constraint propagation method

2.6.1 The Application Domain

Variational systems allow the user to interactively sketch generic, 2D geometry. For instance, the user can add constraints to an undimensioned or partially dimensioned object, or link geometry to equations governing its behavior. Next, the system determines the dimensions and other variables that satisfy the user's constraints. This allows the application of variational systems to engineering tasks that are not well understood, such as the conceptual design stage.

Parametric design systems usually provide generic geometric entities such as chamfers, holes, slots, etc that can be invoked by entering parameters. The user can then build parametric models involving complicated geometry. However, the user must know which dimensions are dependent on others and in what ways. Parametric systems may therefore be most useful when the design task is well understood

2.6.2 The Computational Method

Variational and parametric systems compute parameter values differently. Parametric systems use a sequential equation solver, which normally requires users to know in advance the steps that must be carried out to create and iterate the geometry and moreover to indicate what comes first, then second, third and so on. On the other hand, variational systems employ a simultaneous equation solver and accept somewhat less structured input.

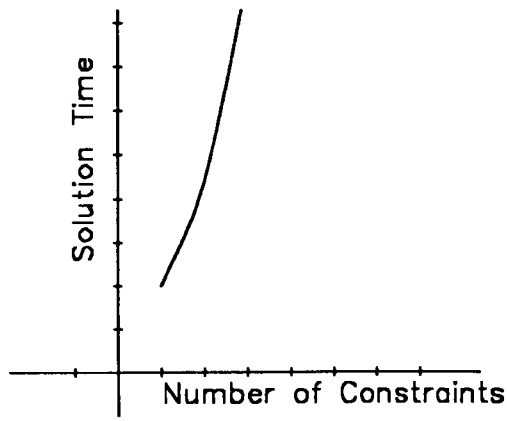
2.6.3 The Computational Load

Computation for a parametric model is less expensive than that of variational geometry. The equations for a parametric model are sequentialy related, and the sequence can be optimised. While for variational systems, as the number of parts grow, the number of constraints can grow even faster. For every constraint, the solution time increases as the square of the number of constraints [CIME 1989] as shown in Figure 2.6.

2.6.4 The Constraint Propagation Method

In parametric design, constraints are unidirected. For instance, the user of a parametric system could establish a parallel constraint governing two lines A and B by indicating that the lines should be separated by a certain distance d and that B should be some angle θ from the known reference line C as shown in Figure 2.7. Accordingly, if the angle θ changes, the parametric system will adjust A so that it remains parallel to B. However, this does not normally hold for the opposite case: should the user change the angle be-

Figure 2.6 Relation between solution time and constraint number for variational geometry

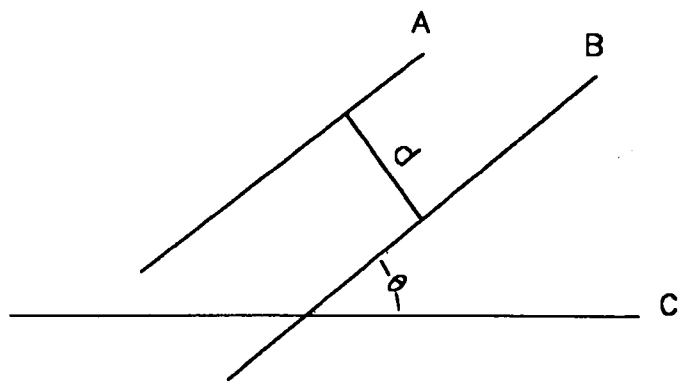


tween A and C, the system will not be able to change the angle between B and C because the parallel constraint goes from B to A. In this way, constraints created with parametric systems typically imply causality.

Reversing the causality of constraints can be complicated and perhaps even impossible for a parametric system. Depending on the situation and the system's capabilities. The user may be required to rewrite the script or to rebuild the model.

Because variational systems use a simultaneous equation solver, they can more easily accommodate changes affecting constraints. For example, Figure 2.8(a) shows a triangle constrained by l, θ and h, If the user later wants to

Figure 2.7 Constraint propagation example



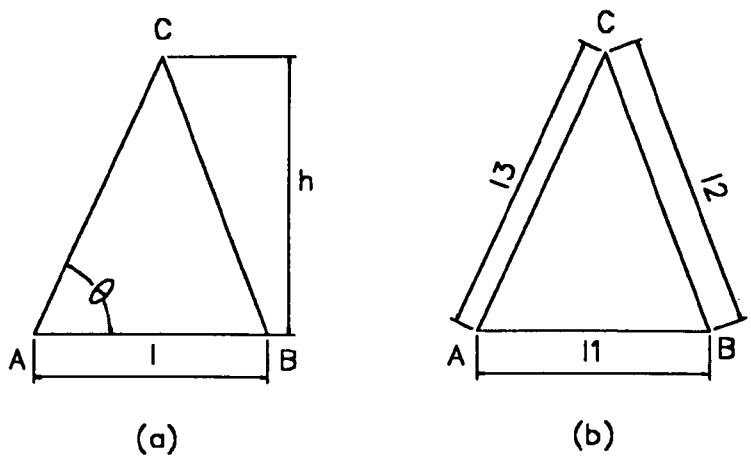
change the constraints to that as shown in Figure 2.8(b), the variational system can automatically adjust to the changes.

2.7 Rule-Based Approaches

As described in section 2.3, problems associated with numerical methods are difficult to overcome. Variational geometry is mainly used for 2D drawing and mechanism analysis. Nowadays, efforts have been made towards a rule based solution to variational geometry [Bruderlin 1986, Aldefeld 1988].

Bruderlin [Bruderlin 1985, 1986] proposes a rule based approach to the geometric construction problem. Based on his approach, geometric constraints are treated as those in variational geometry. Instead of using a numerical

Figure 2.8 Change constraint scheme



method for the simultaneous equation solving, Bruderlin establishes a set of geometric rules to propagate a sequence of constraint satisfaction. The basic ideas of Bruderlin's method are:

1. Objects may be specified by constraints
2. If the specification is written in a suitable formal notation, it may be possible to construct the objects automatically.

In a rule based system, the efficiency of reasoning is very important because of the combinatorial explosion of search space [Stallman an Sussman 1977]. Bruderlin has obviously met this problem but fails to come out with any solution or suggestion. This can be seen from his statements:

An unsolved problem is the time behavior of this approach. Backtracking has exponential time bounds, and so far we do not know of an intelligent way for pruning the backtracking tree.

Apart from the inefficiency of geometric reasoning, Bruderlin's method also has other drawbacks such as the difficulty of building a user interface and the difficulty for the user to maintain a consistent set of constraints. Since Bruderlin's method may generate a great number of variants, he expects the user to rule the useless variants out.

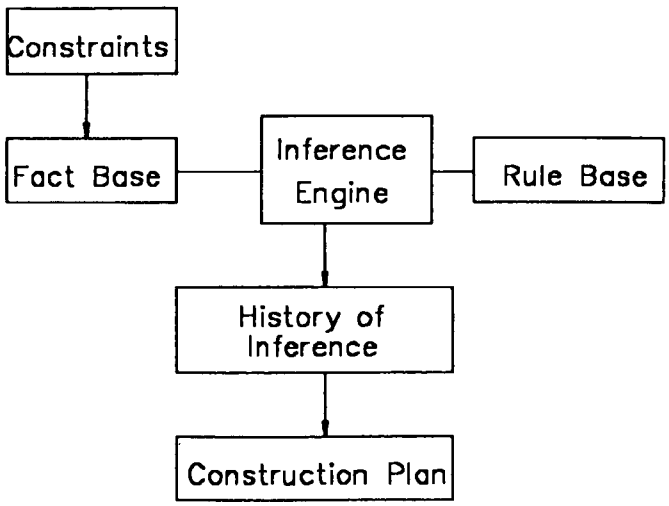
A more detailed description of the approach can be found in Aldefeld's papers [Aldefeld 1986, 1988]. He defines the generic model as a set of points and tracks E, together with a set of constraints C, that is, by the tuple

$$M = < E,C > .$$

An instance of a generic model M is any set of instances of all points and tracks compatible with C. Given the topology and constraints of a generic model, the system starts by working out a "construction plan". Once the construction plan has been derived, the original generic model is only needed for possible future changes. The construction plan now is used to build the precise object geometry. The overall structure of the system is as shown in Figure 2.9. Execution of the construction plan essentially consists of invoking a set of construction procedures (Aldefeld has these defined in Pascal) in the order required by the construction plan and then storing the computed functions.

Aldefeld uses a forward-reasoning method in his implementation. The basic philosophy in Aldefeld's system is to

Figure 2.9 Block diagram of inference components defined by Aldedeld



derive as much explicit knowledge as possible from the current known constraints.

Thus a postprocessor is needed to exclude the data

that are irrelevant to the constraining of the points and tracks of the generic model. The remaining part represents a concise line of reasoning in which all points and tracks are proved constrained [Aldefeld 1988].

From the above description, it can be seen that the geometric reasoning process is inefficient. Besides this drawback, problems such as the completeness of the set of rules, their optimal design and the limitations of the rule based approach in general are important issues but are not addressed in Aldefeld's work.

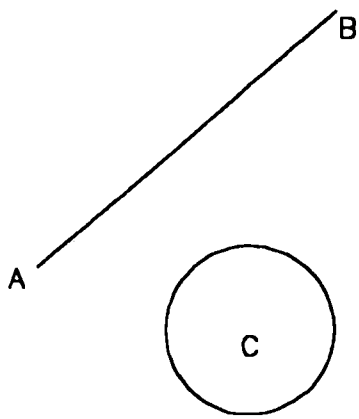
Arbab and Wang [Arbab and Wang 1989] implement a prototype constraint-based design system for the domain of two-dimensional geometry using operational transformation planning (OTP). Object-oriented programming is used in their implementation. As a front-end, the user interface allows the addition of objects and constraints, but each must be given a unique name for reference by the system in its satisfying or maintaining constraints. Since each constraint must be satisfied at data entry, the constraint solver can not resolve constraints as a whole.

Figure 2.10 shows a line AB and a circle C. If a constraint TANGENT is added between AB and C, then there are multiple ways to satisfy the constraint. The system can rotate line AB based on point A or point B, or can move the circle C. This may cause some confusion for the system. Another drawback of the system is that the satisfaction of each constraint is dependent on the previous constraint solutions. Constraint solutions are not influenced by the later added information. This requires the user to plan the input sequence. In addition, once a model is defined, the system does not allow a change of a constraint type.

Declarative specification of shape is more natural for designers than procedural specification [Borning 1981]. The information handled in declarative specifications are dimensions and other geometric constraints, and consequently this entails a naturally higher level of abstraction in dialogues with the system interface.

It is obvious that engineering design generally involves a lot of repetitive geometric manipulation and operations. Parameterised dimension input has

Figure 2.10 Adding a tangential constraint



accordingly been very successful in CAD practice. In this thesis, the principle of a rule-based 2D geometry system called LOGICAD and components of its implementation are described which allows the definition of object geometry in terms of geometric constraints. With LOGICAD, it is possible to relate functional requirements with geometry [Shenton et al 1989] and eventually to reflect how the change in functional specification affects the object geometry. Thus, it is a useful tool for the conceptual design stage.

2.8 Integration of Geometric Modelling and Engineering Functions

From the above discussion, it can be seen that the interplay of function and geometry is one of the crucial issues for intelligent CAD system development. Shenton et al [Shenton et al 1989] propose an expert system shell for the in-

tegrated engineering design of form and function. Their approach is based on the systematic design method [Pahl and Beitz 1988] which views design as a "logical, causal relation between the inputs and outputs of the designed object and the state of the object and the environment" [Shenton et al 1989]. The overall design functions are divided into sub-functions which can match with the catologued form features.

This approach offers a practical way to computerise the routine design work through knowledge based systems. The main defect of this approach is that it can only be used to solve small domain design problem based on existing experience.

2.9 Concluding Remarks

Current ICAD system development has revealed the demand for more explicit representation of geometric information [Arbab 1987]. Existing methods for constraint based geometric modelling are either computationally expensive (variational geometry) or require user interaction (parametric design). Although rule based approaches have been proposed to overcome these problems [Bruderlin 1986, Aldefeld 1988], all of these proposals suffer significant drawbacks such as inefficient reasoning and imcomplete rule bases etc.

In this thesis, a novel geometric knowledge representation scheme called geometric element based reasoning (GEBR) is presented. The method is implemented in a rule based geometric modelling system called LOGICAD. Geometric constraints in LOGICAD are used for the definitions of object shape. The geometric constraints may further be linked with functional con-

straints through engineering equations. As a design evolves, the geometric constraints are refined along with design constraints. In LOGICAD, the methods of dependency-directed backtracking [Stallman and Sussman 1977] and rule-cycle hybrid [Rowe 1988] are used for efficient propagation of geometric constraints. The LOGICAD system has the advantage of declarative specification of geometric constraints and can be used as a basis for sophisticated ICAD system development.

CHAPTER 3

LOGIC PROGRAMMING AND GRAPHICAL FACILITIES

The logic programming language Prolog is novel and none standard computer language. This chapter describes the basic terms of Prolog such as knowledge representation, data structure, and control mechanism. There are many versions of Prolog (such as Wprolog, SD-Prolog, VM-Prolog, etc) on the commercial market, but few provide a full graphical capability. This chapter describes how a graphics environment for logic programming based on the the Tektronix 4100/4200 standard has been provided. Graphical commands in Tektronix 4100/4200 series computers take the form of certain pattern of ASCII code. Encoded graphical commands are reserved as a subset of Prolog predefined predicates. Whenever possible, the graphical predicates obey the convention of GKS (Graphical Kernel System) [ISO7942 1985]. In this chapter, a brief introduction to the Tektronix 4100/4200 terminal programming model [Tektronix manual 1986] is presented. Extended features of Prolog for the development of intelligent CAD appilications and the programming of the LOGICAD system are outlined.

3.1 Introduction to Prolog

Since its inception in the early 1970's, Prolog has evolved to a robust logic programming language. Its merit and contribution have been widely accepted. Although Prolog has its origin in the European AI community, a significant force in Prolog development has arisen from the Japanese Ministry

of International Trade and Industry in 1981 when they announced the creation of a special laboratory to pursue the development of what they called fifth generation computing technology [Moto-Oka 1982]. To the surprise of many computer professionals, logic programming and Prolog in particular was chosen as the fundamental software technology of the fifth generation project. One aim of this project is the production of knowledge information processing systems. Such systems are to be capable of communicating wih users in natural language or graphics, and to be able to assist users in a number of high level domains involving expertise.

3.1.1 Procedural Versus Logic Programming

With programming in a conventional computer language, the programmer needs to specify the sequence of executions. On the other hand, in a logic programming language, the programmer may only need to know what is needed to solve a particular problem. The major difference between the two style of programming languages can be seen from the *pseudo equation*

Program = Algorithm + Data Structure

for procedural languages [Wirth 1976], and

Algorithm = Logic + Control

for logic programming languages [Kowalski 1979]. In an ideal logic programming language, the control aspect of the programming is to be handled by the interpreter (or compiler) of the language. Prolog falls short of this ideal [Gibbins 1988]. One reason is that Prolog "possesses built-in predicates which

have side-effects". Another reason is that Prolog allows the programmer to interfere (by for example **cut**, **assert**, etc) with the control. It can, however, be said that Prolog is an approximation to an implementation of formal predicate logic [Gibbins 1988].

3.1.2 Features of Prolog

The Prolog language is a combination of some powerful ideas, including:

* The use of Horn clauses to represent knowledge
* Descriptive style programming
* Both declarative and procedural semantics
* The ability to intersperse metalevel code with object level code
* Small predefined vocabulary
* Extensibility for new data structures and operators

These features make Prolog a powerful computer language for symbolic computations. These features also allow Prolog to be costomised to the programmer's style. If well written, a Prolog program has readily understandable semantics.

3.1.3 Facts and Rules

A Prolog program consists entirely of a set of clauses, and can be understood as a network of relations that hold between logical terms. A clause is either a fact or a rule. A fact states that a particular relation holds universally and it is written in the form of a name (the predicate) followed by a number of terms (the predicate arguments) inside parentheses. A rule is a predicate

whose truth value depends on the truth values of other clauses or other instances of itself. In other words, a rule is a conditional predicate, that is, it is an implication. Thus a rule can be used to deduce new facts.

Facts provide the simplest form for a Prolog data base. The fact

parallel(line_a, line_b).

states that **line_a** parallels **line_b**. It is up to the programmer to choose the predicate name. The predicate can have any number of arguments.

To retrieve information stored in the data base, a query can be written by prefixing by a question mark. Thus the query

?-parallel(line_a, line_b).

causes the Prolog interpreter to look through the data base for the occurrance or solution of the query, and to respond with with

?-yes

since the queried predicate is found in the database.

If the query includes variables, they are instantiated during the implementation. In Prolog, variables are indicated by having a first character which is upper case. Thus the use of the variables **LINEA** and **LINEB** in the query

?-parallel(LINEA, LINEB).

will produce

LINEA = line_a

LINEB = line_b

Rules can be thought of as conditional facts. They consist of a head and a body with the form **head :- body**. The **body** contains the conditions to be met in order to make the **head** a true fact. Accordingly the :- operator has the function of an **if** operator. Thus the rule

```
parallel(A, C, deduced) :-
        parallel(A, B),
        parallel(B, C).
```

states that if a line **A** parallels line **B** and the line **B** parallels line **C**, then line **A** parallels **C** can be deduced. If the facts

```
parallel(line_a,line_b).
parallel(line_b,line_c).
```

are stored in the data base, a query of the form

```
?-parallel(A, B, XX).
```

results in

A = line_a

B = line_c

XX = deduced.

3.1.4 Conjunction (,) and Disjunction (;)

Conjunction in Prolog is represented as a comma (,). Commas between predicate expressions in a clause are much like logical "and" since all predicate expressions must succeed for the whole clause to succeed. But Prolog commas, unlike logical "and", imply an order of processing. Rearrangement

of the predicate expressions may have significant effect on the performance of a clause. For example, if the following facts are stored in the database

shape(object_one,rectangle).

shape(object_two,rectangle).

color(object_one,white).

color(object_two,black).

The query

?- shape(X,rectangle), not color(X,white).

finds an object whose shape is rectangle and whose color is not white. The answer is

X = object_two

However, if the order of the two predicate expressions is reversed, that is

?- not color(X,white), shape(X,rectangle).

The query does not work because the Prolog predicate **not** does not bind variables.

Disjunction in Prolog is represented by a semicolon (;). When a predicate expression before a semicolon succeeds, all the other expressions to the right that are "or"ed with it are skipped by the Prolog interpreter. When such an expression fails, the next expression to the right of the "or" is tried. If there aren't any more, the whole "or" fails, which results in backtracking to the left.

3.1.5 Data and Control Structure

The only data structure other than the function and parenthesis in Prolog is the list, such as the list containing the seven atoms **a, b, c, d, e, f** and **g**

[a, b, c, d, e, f, g].

Lists can be represented as a head and a tail as

[HEAD|TAIL].

If we equate the **HEAD** and **TAIL** structure to the previous list by the query

?-[HEAD|TAIL] = [a,b,c,d,e,f,g].

The output is

HEAD = a

TAIL = [b,c,d,e,f,g]

The head is always the first element of the list. The tail is the remaining (possibly null) list. A lot of operations on lists can be easily performed in Prolog. The following clause defines the length of a list

```
/*----Boundary condition----*/
length([ ], 0).
 length([H|T], N):-
/*----Tail recursion---------*/
     length(T, N1),
      N is N1 + 1.
```

Program 3.1 Find the length of a list

The length (**ListSize**) of a list (**List**) can be found by invoking the definition clause with the query

?-length(List, ListSize).

The technique of recursion as used above is often used in Prolog programming. Recursion can be used to achieve the same effect as an iterative control mechanism in a procedural language. Any recursive procedure must include at least one each of the following components:

1. A nonrecursive clause that is where the recursion stops.
2. A recursive rule. In the body of this rule, the subgoals generate new argument values from which follows a recursive subgoal utilizing these new argument values.

There are also a number of procedural control predicates like **cut** (!), **assert**, **retract** and some I/O predicates. All these will affect the implementation of a Prolog program. Sometimes the order of the clauses in a program has a significant affect on the implementation since the built-in Prolog control mechanism prefers the earliest occurence in the database and only uses a later occurence when forced to by backtracking.

3.1.6 Applications of Prolog

Prolog has found wide applications in artificial intelligence as well as conventional applications, the main fields include expert systems, natural language processing, relational data bases and graphical applications.

Expert Systems

Expert systems have been built with Prolog in a variety of areas, including equation solving [Bundy and Welham 1981], medicine [Darvas 1980], architecture [Swinson 1980], factory automation and circuit design [Zaumen 1983], financial analysis and decision support [Kriwaczek 1982].

Natural Language Processing

Natural language processing usually includes a parser that can understand the grammer of a language sense [Colmerauer 1978]. Epistle at IBM is a major reseach project in this field [McCord 1985].

Relational Database

Because the relational database model is a logical, well developed formalism, its representation in Prolog is straightforward. Prolog has been proved particularly useful for implementing user interfaces to relational databases. All the major types of query languages have been implemented in Prolog including QBE, SQL and Algebra [Li 1984].

Graphics Manipulation

Traditionally, graphics applications have been procedure-oriented. Since the early 1980's, researchers have investigated the possible applications of Prolog to computer graphics. Gonzalez et al [Gonzalez et al 1984] have compared the effectiveness of Prolog and Pascal in solving graphics problems such as

the construction of a 3-D model from its three orthogonal projections. They conclude that in terms of development time, object code and execution time, Prolog is more effective than Pascal in this application. Franklin et al [Franklin et al 1986] have implemented programs to solve several problems in Prolog, including a subset of GKS (formally not allowed by the GKS standard), convex-hull calculation, planar graph traversal, boolean combination of polygons and cartographic map overlay.

3.2 The Tektronix 4100/4200 Terminal

The Tektronix 4100/4200 terminal offers many facilities for advanced graphical programming applications. This section gives a concise outline of the Tektronix 4100/4200 terminal programming model and its main features in order to have a better understanding of how the Tektronix 4100/4200 terminal's graphical capability is programmed in Prolog.

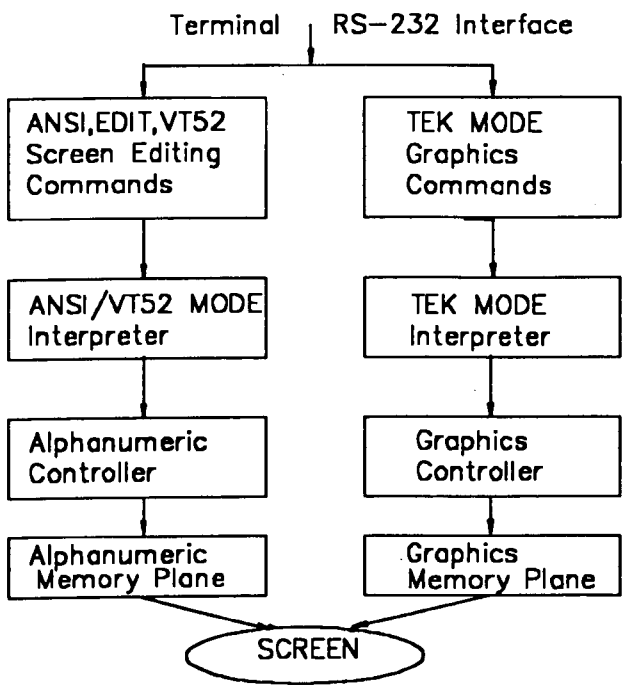
3.2.1 The Tektronix 4100/4200 Terminal's Programming Model

It is convenient to think of the terminal as two different terminals in one package, each having its own set of commands and addressing its own area of memory. These virtual terminals are

* A graphics display terminal
* A text entry and editing terminal

The architecture of the programming model is shown in Figure 3.1. When the terminal functions as a graphics terminal, it processes commands in the following sequence:

Figure 3.1 Programming model of the terminal



1. The program sends TEK 4100/4200-style commands from the host
2. The TEK mode interpreter interprets these commands
3. The TEK mode interpreter invokes the appropriate graphics routines that direct the graphics controller firmware
4. The graphics controller firmware determines how the graphics images will be written to the graphics memory plane

When the terminal functions as a text entry and editing terminal, it processes commands in the sequence:

1. The program sends ANSI or VT52 mode screen editing commands from the host
2. The ANSI/VT52 mode interpreter interprets these commands
3. The ANSI/VT52 mode interpreter invokes the appropriate screen editing routines from the alphanumeric control firmware
4. The alphanumeric controller firmware writes text to the alphanumeric memory plane

Depending on applications, the terminal may frequently switch between the two modes.

3.2.2 Main Features of the Tektronix 4100/4200 Terminal

The terminal offers a wide range of features to support color graphics and screen-editing applications. Briefly these features include:

1. **A Dialog Area** where the user can specify an area of the terminal screen as the display area to display host and user text communication information without interfering with the graphics image.
2. **A Graphics Area** where an area of the terminal screen displays the image created in the color selected.
3. **A Nonvolatile Memory** which allows the user to save certain terminal settings and key definitions in the terminal's permanent memory by which means the user can permanently configure the terminal for a particular application.

4. **The Graphics Input (GIN)** facility which provides an extensive set of graphics input commands for allowing interactive graphics with the host.
5. **The Color Settings** by which the user can specify any one of the three coordinate systems namely HLS (hue, lightness and saturation), RGB (red, green and blue) and CMY (cyan, magenta and yellow). There are 16 color indices for the graphics area and 8 color indices for the dialog area.

These features are perhaps the most important of the many features of the terminal which make it very suitable for building serious CAD application software.

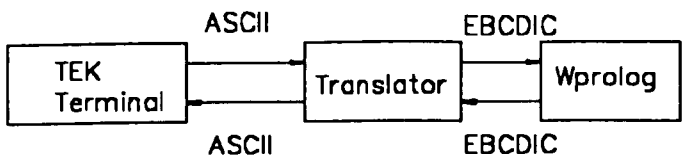
3.3 Terminal and Wprolog Communication

The logic programming language Wprolog is a product of Intralogic Inc. [Intralogic Inc. 1985]. Wprolog is based on the *core syntax* described by Clocksin and Mellish [Clocksin and Mellish 1981]. Some built-in predicates described by Clocksin and Mellish have been enhanced to provide more flexibility. Wprolog also provides several additional built-in predicates such as **ancestor** and **real**.

In programming the Tektronix 4100/4200 terminals with Wprolog, attention must be paid to the different codes they each use for their communications. Wprolog only supports EBCDIC code input and output while the Tektronix 4100/4200 terminals use ASCII code for its command sequences. Thus a *Translator* is needed for the communication between Wprolog and Tektronix 4100/4200 terminals as shown in Figure 3.2. Another important point is that

the Tektronix terminal only sends and receives ASCII code in certain bit patterns, and thus it is the task of the Wprolog to adapt its input and output style to that of the Tektronix terminal.

Figure 3.2 Wprolog-Terminal communication



Whereas in general computing practice most communication software is written in low-level procedural codes to provide an efficient binding between the high-level programming system and the graphical interface, computational experiments with the established hardware have indicated significant speed advantage in avoiding multiple language interfaces. This situation arises largely because the Wprolog system is interpretive in form whereas traditionally accepted communication code is compiled. The system described in this thesis has consequently required the development of a novel communication encoding program written unsually in the high-level interpretive logic

programming language Prolog. The full encoding program is listed in appendix A1.

3.4 Some Extended Features of Wprolog

3.4.1 Arithmetic Operators

Unlike *core* Prolog [Clocksin and Mellish 1981], the Wprolog interpreter supports floating point number calculations. The operators provided are addition (+), subtraction (-), multiplication (*) and division (/). These numerical facilities are not enough for graphics applications. Thus additional predicates such as **sin, cos, tangent** and **power** are implemented in the author's Prolog code by means of power series and recursive functions. The program is given in Appendix 2

3.4.2 Other Extensions

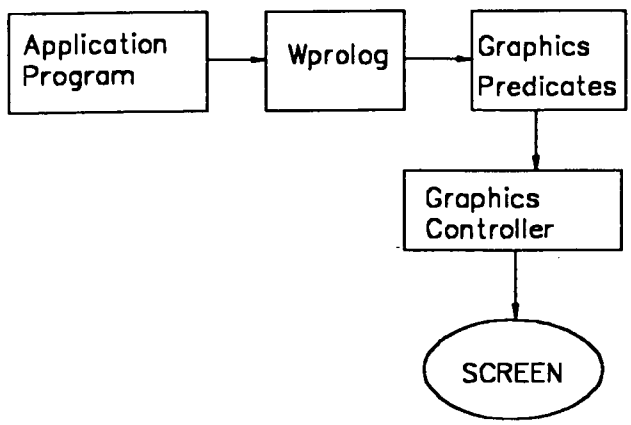
Although Wprolog has a limited predefined vocabulary, the user, can however define more operations and operators on top of Prolog. List operations such as **intersection, union**, and **subtraction** etc are among the extended predicates installed by the author.

3.5 Prolog Environment for Graphical Programming

In order for Prolog to solve graphical problems, an interface between Prolog and the Tektronix 4100/4200 graphic system has been built (Figure 3.3). The interface consists of a set of Prolog predicates which correspond exactly to the

graphic functions supported by the Tektronix 4100/4200 series terminals. These functions are used to generate and control picture displays.

Figure 3.3 Graphics-Prolog Interface



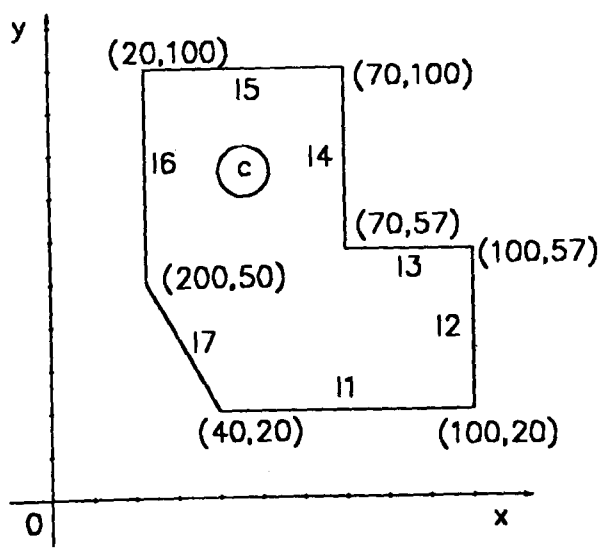
With the binding of Prolog/Graphics, it is accordingly easy to build pictures directly in Prolog. For example, a data structure **geometry** has been defined to store geometric components in a hierarchical structure. The components in the list of the **geometry** structure may be executed to display the picture. Figure 3.4 is a part geometry drawn by program 3.2.

```
geometry(part,[l1,l2,l3,l4,l5,l6,l7,c]).

    l1:- polyline([40:20,100:20]).

    l2:- polyline([100:20,100:57]).
```

```
l3:- polyline([100:57,70:57]).

l4:- polyline([70:57,70:100]).

l5:- polyline([70:100,20:100]).

l6:- polyline([20:100,20:50]).

l7:- polyline([20:50,40:20]).

c :- circle(45:75,6).
```

Program 3.2 Graphical programming in Prolog

Figure 3.4 Graphical programming in Prolog



3.6 Concluding Remarks

Graphical programming is traditionally procedurally oriented. Logic programming, however, adopts a different methodology for computer programming, which is, declarative rather than procedural. Because of the increasing popularity of the logic programming language Prolog, researchers have attempted to apply Prolog to solve geometric problems. They conclude that Prolog is more effective than procedural programming languages in many aspects of graphical applications [Gonzalez et al 1984]. Therefore, it is important for Prolog to have the capability of graphical representation and manipulation. This chapter has presented a binding of Prolog and the Tektronix 4100/4200 series terminal's low level graphical system codes through an original communication encoding program. An example of Prolog for graphical programming is presented which shows the advantage of organizing geometric components in a hierarchical structure.

CHAPTER 4

GEOMETRIC PROBLEMS IN PROLOG

The logic programming paradigm appears to offer the potential for implementing automatic reasoning techniques which can combine the various viewpoints characteristic of engineering design. In this thesis, an implemented ICAD prototype system named LOGICAD is described which presents implementations of key aspects and features of the geometric sub-system of a possible ICAD system. The implementation is achieved through the development of several novel logic programming techniques. The motivation for geometric algorithm implementation in a logic programming language comes from the observation that current ICAD research and development often uses an existing CAD system mainly for the purpose of graphical presentations [Myers and Pohl 1990, Popplestone 1986]. However, geometric constraints are really part of the design constraints. Geometric constraints are refined as a design evolves. The inclusion of geometric algorithms in a logic programming language has the advantage that intelligent deduction about geometric constraints in combination with other constraints (such as manufacturing constraints, functional constraints etc) can be performed intimately within a single logic system. This chapter demonstrates that the proposed Prolog implementation of the geometric algorithms are short, concise and readable.

4.1 Representations of Geometric Data

The simplest geometric object is the point. In the case of 2D geometry, a point is uniquely defined by its x and y coordinates. The internal representation of a point in LOGICAD is given by

vertex(V,X,Y,COUNTER).

where **V** is the point identifier and **X** and **Y** are the coordinates. The attribute **COUNTER** is a record of how many lines the point **V** belongs to. Initially **COUNTER** is set to be zero, its value is incremented by 1 each time a new line is created which emanates from this point.

The other important geometric objects are line segments including straight lines, circular lines, etc. It is convenient to represent a straight line in terms of its two end points as in

line(ID,V1,V2).

ID is the name of the line segment. **V1** and **V2** are the two end points. For a circle, its representation is

circle(ID,Xo,Yo,R).

As before, **ID** is the name of the circle. **Xo** and **Yo** are the centre coordinates. **R** is the radius. Other geometric primitive objects such as arcs and ellipses are also supported as listed in Appendix 1 but are not discussed here.

4.2 Output Functions

The output primitives defined by the international graphical standard GKS (Graphics Kernel System) [ISO 7942 1985] are POLYLINE, POLYMARKER, TEXT, FILL AREA, CELL ARRAY and GENERALISED DRAWING PRIMITIVES (GDP). The appearance of their output is defined by their output attributes such as line type and line index etc. With these facilities, a wide range of object shapes can be modelled. Where possible, the implementations of geometric functions in the LOGICAD system obeys the convention of GKS.

As an example, the POLYLINE implementation is presented. It is perhaps the most often used output function in graphical programming. The specification in LOGICAD is given as

gs_polyline(POINTS).

where **POINTS** is a list of coordinate pairs in the form [X1:Y1,...,Xn:Yn]. The implementation of POLYLINE in Prolog is shown in Program 4.1.

```
gs_polyline(POINTS):-
/*---enter vector drawing mode---*/
    entervector,
    gs_polyline1(POINTS), !.
gs_polyline([]):-
/*---when completed, enter alpha mode---*/
    enteralpha.
gs_polyline1([X:Y|T]):-
/*---output x and y coordinate code---*/
```

```
    ibmpack(X,Y),
/*---tail recursion---*/
gs_polyline1(T).
```

Program 4.1 Draw polyline

4.3 Edit Functions

In defining an object geometry, edit functions such as deletion and transformation of geometric entities are often needed. This section shows the implementations of the two frequently used edit functions. The first is the deletion of a line and the second is the translation of a geometric object.

4.3.1 Deletion

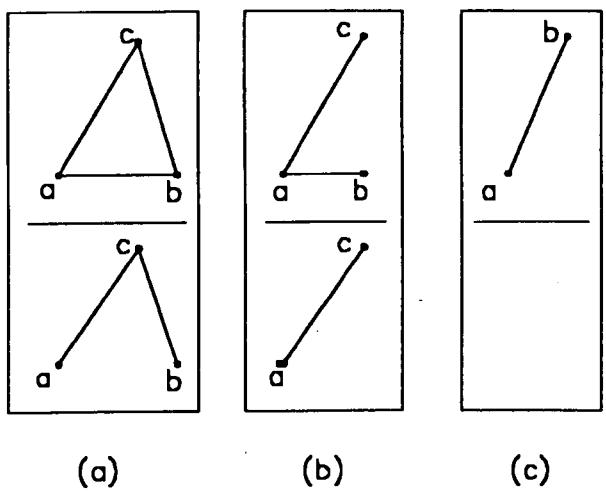
Line deletion is often used in the creation of a picture. In the implementation of the line deletion, three possibilities should be considered.

1. The line has no free end, and after line deletion, the counter of the two end vertices are decreased by 1 as in the case of Figure 4.1(a)
2. The line has a free end as in Figure 4.1(b). Line ab is deleted, the dangling vertex b must also be deleted
3. Both end vertices of a line are free ends as in Figure 4.1(c), delete the line and its end vertices

The implementation of line deletion is given by Program 4.2

```
delete_line(ID):-
```

Figure 4.1 Delete a line




```
        line(ID,V1,V2),
        retract(line(ID,_,_)),
        delete_vertex(V1),
        delete_vertex(V2).
delete_vertex(V):-
        vertex(V,X,Y,C),
        ((C = 1, retract(vertex(V,_,_,_)));
         (C1 is C - 1,
          retract(vertex(V,_,_,_)),
          assert(vertex(V,X,Y,C1)))).
```

Program 4.2 Delete a line

4.3.2 Translation

In 2D geometry, translation of an object is defined by the translation vector **T** = (Dx,Dy). A single vertex is translated by adding the translation vector to the vector representing the vertex. For instance, the vertex vector **V** = (Vi,Xi,Yi,_) is translated by the equation

$$t(\mathbf{V}) = \mathbf{V} + \mathbf{T}$$

or $t(Vi) = (Vi, Xi + Dx, Yi + Dy, C)$.

To translate a geometric object requires translating all vertices of the object that are stored in the data file. This is implemented in Program 4.3.

```
    /*---translate a geometric object---*/
translate(VIEW,Dx,Dy):-
    /*---open the file and read an item---*/
see(VIEW), read(ITEM),
    /*---if end of file then stop---*/
((ITEM = eof,seen);
    /*---ITEM is vertex information--*/
(ITEM = vertex(V,X,Y,C),
    /*---calculate new x and y values---*/
X1 is Dx + X,Y1 is Y + Dy,
wwrite(vertex(V,X1,Y1,C)),
translate(VIEW,Dx,Dy));
translate(VIEW,Dx,Dy)).
```

Program 4.3 Geometric translation

4.4 The Prolog Search Mechanism

The Prolog interpreter adopts depth-first as its implementation control strategy [Clocksin and Mellish 1981]. It is consequently easy to implement depth-first in a Prolog program. The following is a list of the Prolog program for depth-first search

```
/*---A is the initial state, Z is the goal---*/
/*---SOL is the solution path from A to Z----*/
search(A,Z,SOL):-
    search(A,Z,[A],SOL).
search(A,Z,LIST,SOL):-
    arc(A,B), not member(B,LIST),
    write(A- > B), tab(2),
    search(B,Z,[B|LIST],SOL).
search(A,Z,List,SOL):-
    arc(A,Z), write(A- > Z),
    List1 = [Z|List],
    reverse(List1,SOL).
```

Progrom 4.4 Depth-first search

With Program 4.4, a query

```
?-search(A,Z,SOL).
```

will find a solution path **SOL** from the starting state **A** to the goal state **Z**. The **member** predicate is used to check whether a node has been visited or not in order to prevent an infinite loop. During the implementation, the history of the search is displayed. A solution path is stored in **SOL**.

To illustrate how the program works, consider the route planning graph problem in Figure 4.2. Each node represents a graph junction, A query

?-search(a,f,SOL).

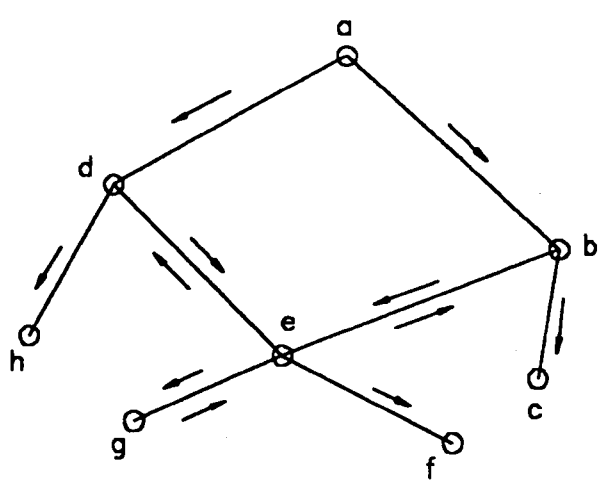
will cause

a->b b->c b->e e->d d->h d->e e->g e->f

SOL = [a,b,e,f]

to be displayed.

Figure 4.2 Route planning problem



From the example, we can see that the Prolog interpreter is well suited to solving problems involving symbolic computation. Obviously depth-first

search may not be efficient for all problems. But with Prolog, we can easily implement breadth-first and other more advanced search strategies [Shenton and Chen 1990].

4.5 Planar Graph Traversal

To construct a 3-D object from its 2-D projections, connectivity information about vertices are needed [Gonzalez 1984]. Figure 4.3(a) shows a projection which is internally treated as in Figure 4.3(b). When an edge is shared by two faces, it is represented as a pair, bearing the same vertex and edge identities, only with the edges in different directions. The representation of Figure 4.3(a) is

line(e1,v1,v2). **line(e6,v5,v6).**

line(e2,v2,v3). **line(e7,v6,v7).**

line(e3,v3,v4). **line(e8,v7,v1).**

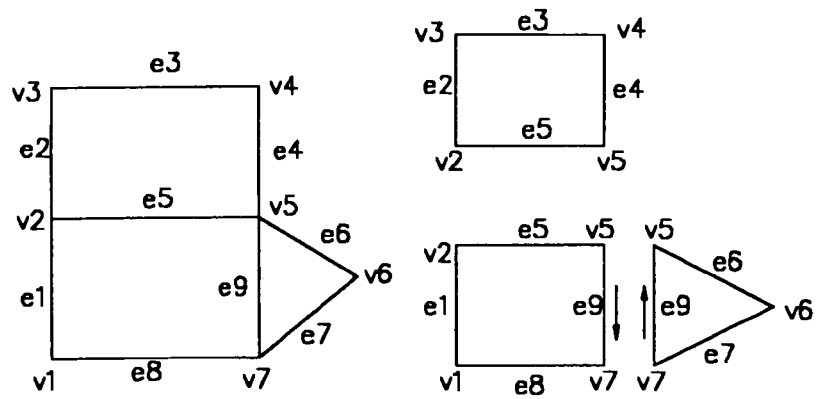
line(e4,v4,v5). **line(e9,v7,v5).**

line(e5,v5,v2). **line(e9,v5,v7).**

line(e5,v2,v5).

Obviously the purpose of the graph traversal in this application is to find closed loops. Each loop represents a face. But there are several cases which are invalid loops. In Figure 4.4 (a), a dangling edge appears. This happens because e5 is shared by two faces and the traversal program starts from v5. In Figure 4.4(b), line e2 is shared by two faces and thus the single line again makes a loop. Figure 4.4(c) shows that a loop [V1,V2,V3,V4,V5] contains multiple faces and thus is invalid. These problems are prevented by the predicate expression **no_edge** which is defined as

Figure 4.3 Graph traversal problem

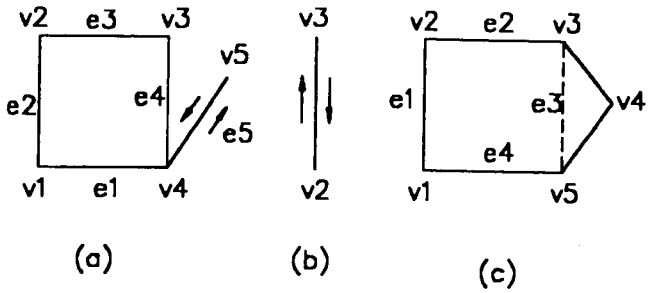



```
no_edge(V,[ ]).
no_edge(V,[V1|T]):-
   not line(_,V1,V); not line(_,V,V1),
   no_edge(V,T).
```

The following is a list of the implemented Prolog program that finds only the valid loops.

```
/*---planar graph traversal---*/
find_loops(LOOPS):- find_loops([],LOOPS).
find_loops(L,LOOPS):- find_loop(LOOP),
   find_loops([LOOP|L],LOOPS), !.
find_loops(LOOPS,LOOPS):- not find_loop(LOOP).
```

Figure 4.4 Invalid loops for graph trversal problems




```
    delete(X,[X|T],T).
delete(X,[X|T],[X|T1]):-
    delete(X,T,T1), delete_loop([V]).
delete_loop([V1,V2|T]):-
    line(E,V1,V2), retract(line(E,V1,V2)),
    delete_loop([V2|T]).
find_loop(LOOP):- assertz(line(0,0,0)),
    line(E,V1,V2,_), not(E = 0),
    find_loop(V2,[V1],[E],LOOP),
    delete_loop(LOOP),!.
find_loop(V1,V,E,LOOP):-
```

```
    line(E1,V1,V2), not(E1 = 0), not member(V2,V),
    no_edge(V2,V),
    find_loop(V2,[V1|V],[E1|E],LOOP).
find_loop(V1,V,E,LOOP):-
    line(E1,V1,V2), not(E1 = 0),
    no_edge(V2,V), last(V2,V),
    not member(E1,E),
    reverse([V2,V1|V],LOOP).
last(V1,V):- reverse(V,[V1|T]).
no_edge(V,[V V]).
no_edge(V,[V1|T]):- not line(E,V1,V),
    no_edge(V,T).
reverse(LIST,R):-  reverse(LIST,[ ],R).
reverse([ ],R,R).
reverse([H|T],M,R):- reverse(T,[H|M],R).
length([ ],0).
length([H|T],N):- length(T,N1), N is N1 + 1.
```

Program 4.5 Planar graph traversal

Based on the data base for Figure 4.3(a), the query

```
?-find_loops(LOOPS).
```

gives the result

```
LOOPS = [[v5,v6,v7,v5],[v2,v3,v4,v5,v2],[v1,v2,v5,v7,v1]]
```

4.6 Clipping

Clipping is the process of extracting a portion of a geometric data base. It is fundamental to several aspects of computer graphics. In viewing a transformation, we often need only to display the information that falls into a specific region. In some more advanced graphics algorithms like hidden-line and hidden-surface removal, shadow and texture display, clipping plays an important part. In addition, a clipping algorithm can also be easily extended to perform boolean operations on polygons. This section presents a novel logic programming implementation of the Weiler-Atherton clipping algorithm [Weiler and Atherton 1977] in Prolog.

While most clipping algorithms require a convex clipping region (e.g. Sutherland-Hodgeman algorithm [the Sutherland and Hodgeman 1974]), the Weiler-Atherton algorithm is capable of clipping a concave polygon with interior holes to the boundary of another concave polygon also with interior holes. The polygon to be clipped is the subject polygon. The clipping region is the clipping polygon. The exterior boundaries of the polygons are denoted by clockwise vertices, and the interior boundaries or holes by counterclockwise vertices. When traversing the list of vertices, this convention ensures that the inside of the polygon is always to the right. The boundary lines of the subject polygon and clip polygon may or may not intersect. If they intersect, then the intersections occur in pairs. One of the intersections occurs when a subject polygon edge enters the inside of the clip polygon, and one when it leaves.

The formal statement of the Weiler-Atherton algorithm is:

1. Determine the intersections of the subject and clip polygons
2. Process non-intersecting polygons
3. Create two intersection lists
4. Perform the actual clipping.

Fundamentally, the algorithm starts at an entering intersection and follows the exterior boundary of the subject polygon clockwise until an intersection with the clip polygon is found. At the intersection a right turn is made, and the exterior boundary of the clip polygon is followed clockwise until an intersection with the subject polygon is found. Again, at the intersection, a right turn is made, with the subject polygon now being followed. The process is continued until the starting point is reached. Interior boundaries of the polygons are followed counter-clock- wise. Figure 4.5 shows an example with one resultant polygon.

To start with, the intersections between the subject and clip polygon edges must be found. A line segment from point P_1 to P_2 is represented parametrically as

$$P(t) = P_1 + (P_2 - P_1)t \quad 0 \leq t \leq 1 \tag{4.1}$$

or in the form

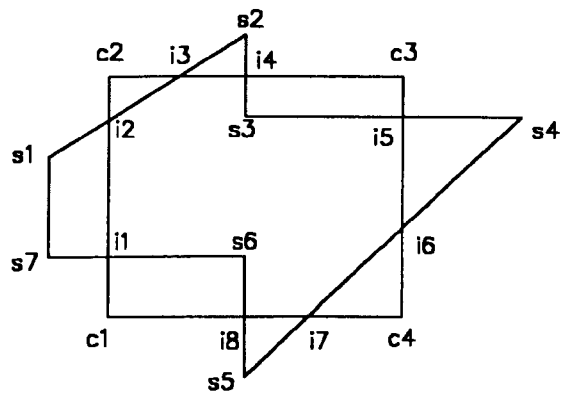
$$P(t) = [X_1 Y_1] + [X_{21} Y_{21}]t \tag{4.2}$$

where $X_{21} = X_2 - X_1$, $Y_{21} = Y_2 - Y_1$

In the proposed Prolog representation, the geometric information is stored as

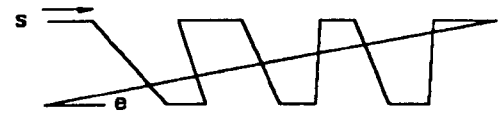
vert(VERT,X,Y,_).

Figure 4.5 Polygon clipping with one resultant polygon



(a)

[s1,[i2,i3],s2,[i4,s3,i5],s4,[i6,i7],s5,[i8,s6,i1],s7,s1]



(b)

[c1,[i1,i2],c2,[i3,i4],c3,[i5,i6],c4,[i7,i8],c1]

line(LINE,V1,V2).

eqn(LINE,X1,Y1,X21,Y21).

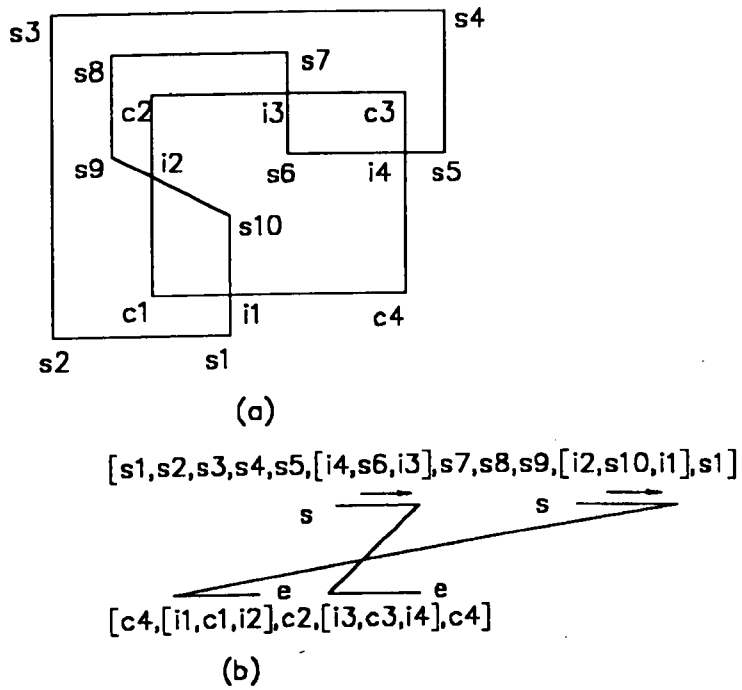
for vertex, line segment and parametric line equation respectively.

If two line segments have end points P_1, P_2 and P_3, P_4 respectively, they can be represented as

$$P(t_1) = [X_1 Y_1] + [X_{21} Y_{21}]t1 \quad 0 \leq t1 \leq 1 \tag{4.3}$$

$$P(t_2) = [X_3 Y_3] + [X_{43} Y_{43}]t2 \quad 0 \leq t2 \leq 1 \tag{4.4}$$

Figure 4.6 Polygon clipping with multiple resultant polygons



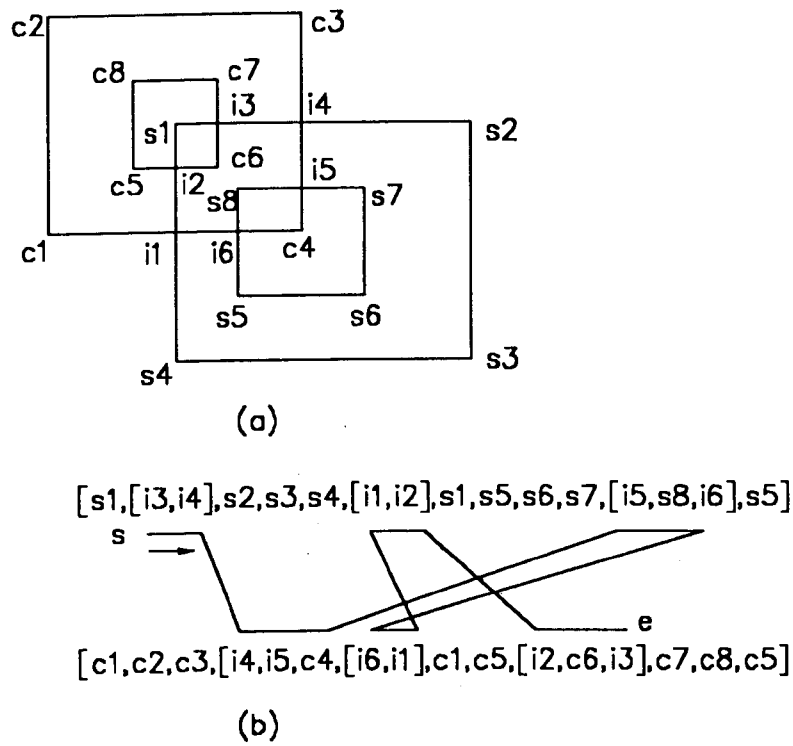
by equating the X and Y terms, we can find the parameter values for t_1 and t_2 from the following two equations:

$$X_1 + X_{21}t_1 = X_3 + X_{43}t_2 \tag{4.5}$$

$$Y_1 + Y_{21}t_1 = Y_3 + Y_{43}t_2 \tag{4.6}$$

The two equations can be solved simultanously. If there is no solution, then the lines are parallel or colinear. If either t_1 or t_2 is outside the required range [0,1], the two line segments do not intersect. If both t_1 and t_2 are within the range [0,1], the two line segments intersect and the intersection can be calculated.

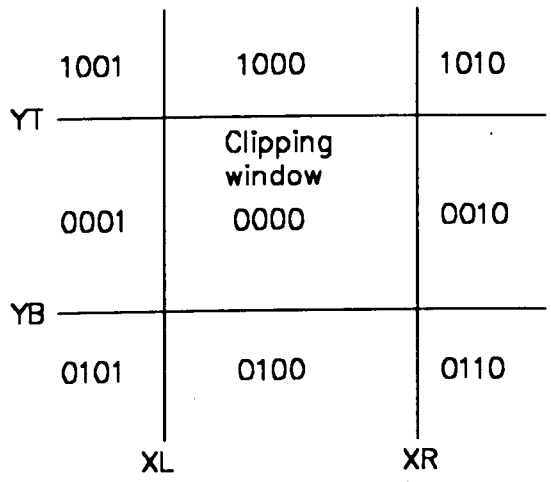
Figure 4.7 Polygon clipping with holes



By clipping each edge of the subject polygon against the clip polygon, all the intersections may be found. Since a picture with a large number of lines may be clipped against a clipping polygon and the edge intersection calculation is computationally expensive [Rogers 1985], the efficiency of clipping algorithms is of particular interest. In many cases, the large majority of lines are either interior to or exterior to the clipping polygon. Therefore, it is important to be able to quickly identify wether a line is totally visible, partially visible or totally invisible. Only the lines that are partially visible are used for the edge intersection calculation.

Cohen and Sutherland [Rogers 1985] uses a four-bit code method to identify totally visible and totally invisible lines. Based on their method, the clipping

Figure 4.8 codes for line and point regions



plane is divided into nine regions as shown in Figure 4.8. Each region is assigned a four-bit code. The rightmost bit is the first one. The bits are set to 1 based on the following scheme :

First-bit ----- if the end point is to the left of the window

Second-bit ----- if the end point is to the right of the window

Third-bit ----- if the end point is below the window

Fourth-bit ----- if the end point is above the window

Otherwise, the bits are set to zero. The Prolog program for the vertex coding scheme is implemented as

```
vertex_code(VERT_ID,CODE):-
    vertexCode(VERT_ID,CODE), !.
```

```
vertex_code(VERT_ID,CODE):-
vert(VERT_ID,X,Y),
clip_window(XL,XR,YT,YB),
(( X = < XL, CODE_1 = 1);
  CODE_1 = 0),
(( X > = XR, CODE_2 = 1);
  CODE_2 = 0),
(( Y = < YB, CODE_3 = 1);
  CODE_3 = 0),
(( Y > = YT, CODE_4 = 1);
  CODE_4 = 0),
CODE = [CODE_1,CODE_2,CODE_3,CODE_4],
assert(vertexCode(VERT_ID, CODE)).
```

Program 4.6 Code for vertex

From Figure 4.8, it is obvious that, if both end point codes are zero, then both ends of the line lie inside the clipping window and the line is totally visible. Therefore it must be displayed.

The end point codes can also be used to trivially reject totally invisible lines. To do this, a bit-by-bit logical ***and*** operation of the two end point codes is performed. If the combined code is not zero, then the line is totally invisible and may be rejected. The code combination of the two end points of a line is performed in the Prolog program

```
line_code(LINE_ID,CODE):-
        lineCode(line_ID,CODE), !.
line_code(LINE_ID,CODE):-
```

```
        line(LINE_ID,VERT_1,VERT_2),
        vertexCode(VERT_1,CODE_1),
        vertexCode(VERT_2,CODE_2),
        combine(CODE_1,CODE_2,CODE),
        assert(lineCode(LINE_ID,CODE)).
combine([C11,C12,C13,C14],[C21,C22,C23,C24],[C1,C2,C3,C4]):-
        compare(C11,C21,C1),
        compare(C12,C22,C2),
        compare(C13,C23,C3),
        compare(C14,C24,C4).
    compare(C1,C2,C):-
        ((C1 = C2, C = C1); C = 0).
```

Program 4.7 Code for line segment

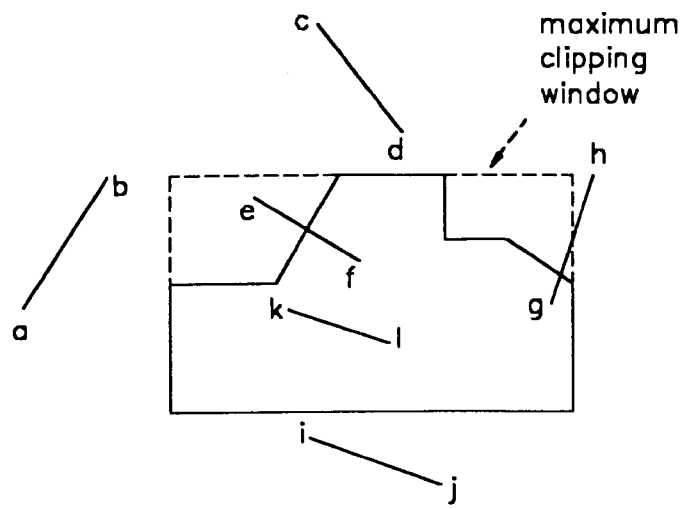
If the clipping region is not a rectangular area, a maximum clipping window can be formulated such as the dashed line shown in Figure 4.9. The coding method can then be used. Since the codes for line ab, cd and ij are not zero, they are invisible in the maximum clipping window and thus can be rejected.

After all the intersections have been found, the process then begins to traverse the boundaries of the subject and clip polygons. When an intersection which enters the other polygon is found, a sublist is formed. The traversal proceeds until the starting vertex is reached.

To find the polygons inside the clip polygon, the following method is used

1. Find a sublist from the subject polygon, mark the first element

Figure 4.9 Identify invisible lines based on the codes of lines



2. Jump to the clip polygon vertex list, find a sublist with the first element being the last element of the last subject polygon sublist, mark the ending element
3. Repeat until the starting element is reached again
4. Test if there is any sublist in the subject polygon vertex list, and if not, end the clipping process, otherwise repeat from (a)

Polygons outside the clipping polygon are found using the same method, except that the initial intersection vertex is obtained from the leaving list and the clip polygon vertex list is followed in the reverse direction. The polygon lists are copied to the outside holding list.

Various circumstances are illustrated as examples. In Figure 4.5, the result is a simple polygon; Figure 4.6 has two clipped polygons as a result; Figure 4.7 clips two polygons both with holes. Their corresponding vertex traversals are shown in Figure 4.5(b), Figure 4.6(b) and Figure 4.7(b).

A list of the program which performs the actual clipping is given as

```
/*------- Program for polygon clipping -------*
clip(POLY,CLIP,OUT):-
      clip(POLY,CLIP,[],OUT).
clip(POLY,CLIP,OUT,OUT):-
      not find_sub(POLY,_).
clip(POLY,CLIP,LIST,OUT):-
      find_sub(POLY,[HP|TP]),
      find_poly([HP|TP],POLY,CLIP,POLY1,CLIP1,[HP],ONE,last),
      append(ONE,LIST,LIST1),
      clip(POLY1,CLIP1,LIST1,OUT).
find_poly([HP|TP],POLY,CLIP,POLY2,CLIP2,LIST,ONE,LAST):-
      last([HP|TP],LP),
      find_sub(CLIP,[LP|TC]),
      last([LP|TC],LC),
      assert_sub(TP,LIST,LIST1),
      assert_sub(TC,LIST1,LIST2),
      delete_sub([HP|TP],POLY,POLY1),
      delete_sub([LP|TC],CLIP,CLIP1),
      ((find_sub(POLY1,[LC|TPP]),
find_poly([LC|TPP],POLY1,CLIP1,POLY2,CLIP2,LIST2,ONE,LC));
```

```
    find_poly([LP|TC],POLY1,CLIP1,POLY2,CLIP2,LIST2,ONE,LC)).
find_poly([HP|TP],POLY,CLIP,POLY,CLIP,LIST,ONE,LAST):-
    last(LIST,LAST),reverse(LIST,ONE).
delete_sub(SUB,[SUB|T],T).
delete_sub(SUB,[H|T],[H|T1]):-
    delete_sub(SUB,T,T1).
find_sub([H|T],SUB):-
    functor(H,.,V),SUB = H.
find_sub([H|T],SUB):-
    functor(H,.,V),not SUB = H,
    find_sub(T,SUB).
find_sub([],_):-fail.
find_sub([H|T],SUB):- not functor(H,.,V),
    find_sub(T,SUB).
/* assert the elements of a list to a list*/
assert_sub([],OUT,OUT).
assert_sub([H|T],LIST,OUT):-
    append(H,LIST,LIST1),
    assert_sub(T,LIST1,OUT).
/* find the last element of a list */
last([],empty).
last([E],E).
last([H|T],E):- last(T,E).
append(A,[],[A]).
append(A,[H|T],[A,H|T]).
reverse(LIST,R):- reverse(LIST,[],R).
```

```
reverse([],R,R).

reverse([H|T],M,R):- reverse(T,[H|M],R).
```

Program 4.8 Polygon clipping

4.7 Prolog for CAD Data Structures

In current CAD systems, boundary representation (B-rep) is the most widely used data scheme because it supports all the numerical algorithms including hidden line removal, ray tracing, illumination, and tool-path generation etc [Requicha and Voelcker 1982]. An important requirement for a boundary data structure is quick and easy access to topological information about objects [Woo 1985]. Since objects can be described in terms of three abstract entities: Vertices(V), Edges(E), and Faces(F), there may be nine topological relationships among the three entities [Woo 1985] as listed below:

V-V V-E V-F

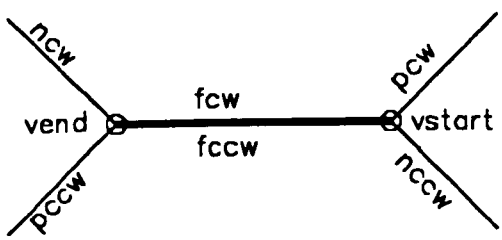
E-V E-E E-F

F-V F-E F-F

Because of the trade-off between access time and storage space, not all such relationships are explicitly stored in existing CAD practice. Thus an efficient data structure for B-rep is needed. The winged-edge data structure [Baumgart 1975] is one of the most widely used data schemes by CAD researchers and developers.

The winged-edge data structure maintains the adjacency information with pointers to the two neighbouring faces fcw (clockwise face) and fccw (counterclockwise face), the two end vertices vstart (the starting vertex) and

Figure 4.10 Winged-edge data structure



vend (the ending vertex), and four of the adjacent edges ncw (next clockwise edge), pcw (previous clockwise edge), nccw (next counterclockwise edge) and pccw (previous counterclockwise edge). Figure 4.10 shows once more the meaning of the various data items. From the winged-edge data structure, we can directly query an edge to find four of its adjacent edges (ncw, nccw, pcw, pccw). We can also directly query the data structure to ascertain the two faces (fcw, fccw) sharing the edge as well as the two adjacent vertices (vstart, vend) that bind the edge. These queries are easy to implement since the necessary data is explicitly stored in the data base. For instance, to find the two faces that share an edge, the predicate expression **ei_faces** which is implemented as

```
ei_faces(Edge,FACES):-
        edge_inf(Edge,Fcw,Fccw,_,_,_,_),
        FACES = [Fcw,Fccw].
```

can be used.

Other adjacency relationships are not explicitly stored and must therefore be determined algorithmically by traversing the graph data structure. For example, to extract the edges which form a face, it is possible to follow the ncw and nccw pointers. Figure 4.11 shows a rectangular prism. The data required to describe the prism is given in Table 1.1 and Table 1.2 in Chapter 1. To find the boundary edges of face f2, edge e9 is first found explicitly from the data base. The next edge is given by ncw(e9) = e6. Because e6 requires the counter clockwise traversal (which can be seen from the face f2), the next edge is now given by nccw(e6) = e1. Similarly we get nccw(e1) = e5, and ncw(e5) = e9. Since e9 is the starting edge, we know all edges have been extracted. Thus face f2 is bound by the edges [e9,e6,e1,e5]. This traversal process is implemented in a predicate expression **fi_edges**

```
fi_edges(FACE,EDGES):-
    f_first_edge(FACE,Fedge),
    fi_edges(FACE,Fedge,[Fedge],EDGES),
    asserta(fedges(FACE,EDGES)).
fi_edges(FACE,Fedge,EM,EDGES):-
    (edge_inf(Fedge,FACE,_,Nedge,_,_,_);
    edge_inf(Fedge,_,FACE,_,_,Nedge,_)),
    (member(Nedge,EM),
     EDGES = EM);
```

fi_edges(FACE,Nedge,[Nedge|EM],EDGES)).

In order to formalise all the information retrieval operations, the following terms are introduced :

Vi--- a particular vertex

Ei--- a particular edge

Fi--- a particular face

{V}-- a list of vertices

{E}-- a list of edges

{F}-- a list of faces

|V|-- the length of list {V}

|E|-- the length of list {E}

|F|-- the length of list {F}

Based on the terms introduced, all the nine possible operations are presented in this section as a novel logic program implementation in Prolog. The program is listed as Program 4.9. Table 4.2 lists all the relevant information about the nine operations

```
/*---Prolog for Winged-Edge Data Structure-------*/
/*---vert(V,X,Y,Z), line(ID,Vs,Ve)----------------*/
/*---edge-inf(Edge,Fcw,Fccw,Ncw,Pcw,Nccw,Pccw)---*/
/*---v_first_edge(V,First_Edge)------------------*/
/*---f_first_edge(Face,First_Edge)---------------*/
ei_vertices(EDGE,Vlist):-
        line(EDGE,Vs,Ve),
        Vlist = [Vs,Ve].
```

Operations	Prolog Predicates	Descriptions	Others
Vi-{V}	**vi_vertices**	Find {V} connected to Vi	-
Vi-{E}	**vi_edges**	Find {E} sharing Vi	-
Vi-{F}	**vi_faces**	Find {F} around Vi	-
Ei-{V}	**ei_vertices**	Find {V} connected by Ei	|V| = 2
Ei-{E}	**ei_edges**	Find {E} connected to Ei	|E| = 4
Ei-{F}	**ei_faces**	Find {F} sharing Ei	|F| = 2
Fi-{V}	**fi_vertices**	Find {V} around Fi	-
Fi-{E}	**fi_edges**	Find {E} around Fi	-
Fi-{F}	**fi_faces**	Find {F} around Fi	-

Table 4.1 Prolog predicates and information retrieval operations

```
/*---Find The Four Edges Connected by EDGE-------*/
ei_edges(EDGE,Elist):-
        edge_inf(EDGE,_,_,Ncw,Pcw,Nccw,Pccw),
        Elist = [Ncw,Pcw,Nccw,Pccw].
/*---Find The Two Edges sharing EDGE------------*/
ei_faces(EDGE,Flist):-
        edge_inf(EDGE,Fcw,Fccw,_,_,_,_),
        Flist = [Fcw,Fccw].
/*---Find All Edges around FACE----------------*/
fi_edges(FACE,Elist):-
```

```
        f_first_edge(FACE,Fedge),
        fi_edges(FACE,Fedge,[Fedge],Elist),
        asserta(fi_e(FACE,Elist)).
fi_edges(FACE,Fedge,EM,Elist):-
        (edge_inf(Fedge,FACE,_,Nedge,_,_,_);
         edge_inf(Fedge,_,FACE,_,_,Nedge,_)),
        ((member(Nedge,EM),
         Elist = EM);
        fi_edges(FACE,Nedge,[Nedge|EM],Elist)).
/*---Find All Vertices around FACE----------*/
fi_vertices(FACE,Vlist):-
        fi_edges(FACE,Elist),
        fi_vertices(Elist,[],Vlist).
fi_vertices([],EM,Vlist):-
        no_double(EM,Vlist).
fi_vertices([Edge|T],EM,Vlist):-
        line(Edge,Vs,Ve),
        fi_vertices(T,[Vs,Ve|EM],Vlist).
/*---Find All Faces Around FACE-------------*/
fi_faces(FACE,Flist):-
        fi_edges(FACE,Elist),
        fi_faces(FACE,Elist,[],Flist).
fi_faces(FACE,[],EM,Flist):-
        delete(FACE,EM,Flist).
fi_faces(FACE,[Edge|T],EM,Flist):-
        edge_inf(Edge,Fcw,Fccw,_,_,_,_),
```

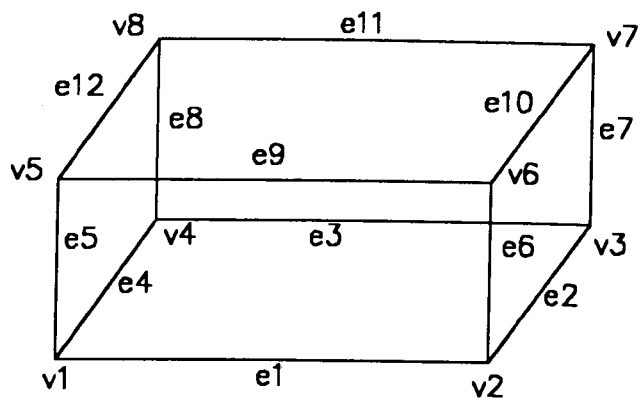
```
        fi_faces(FACE,T,[Fcw,Fccw|EM],Flist).
/*---Find All Edges emanating from V-------*/
vi_edges(V,Elist):-
        vi_edges(V,[],Elist),
        asserta(vi(V,Elist)),!.
vi_edges(V,EM,Elist):-
        line(Edge,Vs,Ve),
        retract(line(Edge,Vs,Ve)),
        assertz(line(Edge,Vs,Ve)),
        ((Edge = end,Elist = EM);
        ((Vs = V;Ve = V),
        vi_edges(V,[Edge|EM],Elist));
        vi_edges(V,EM,Elist)).
/*---Find All Vertices connected to V-----*/
vi_vertices(V,Vlist):-
        vi_edges(V,Elist),
        vi_vertices(V,Elist,[],Vlist),!.
vi_vertices(V,[],Vlist,Vlist).
vi_vertices(V,[Edge|T],EM,Vlist):-
        (line(Edge,V,VC);
        line(Edge,VC,V)),
        vi_vertices(V,T,[VC|EM],Vlist).
/*---Find All Faces around V----------*/
vi_faces(V,Flist):-
        vi_edges(V,Elist),
        vi_faces(V,Elist,[],Flist),!.
```

```
vi_faces(V,[Edge|T],EM,Flist):-
        ei_faces(Edge,[F1,F2]),
        vi_faces(V,T,[F1,F2|EM],Flist).
vi_faces(V,[],EM,Flist):-
        no_double(EM,Flist).
member(EL,[EL|T]):-!.
member(EL,[H|T]):-
        member(EL,T),!.
/*---Delete all occurences of X---------*/
delete(X,[X|Xs],Ys):-
        delete(X,Xs,Ys).
delete(X,[Y|Xs],[Y|Ys]):-
        delete(X,Xs,Ys).
delete(X,[],[]).
/*---Delete Duplicates------------------*/
no_double([X|T],[X|Ts]):-
        member(X,T),
        delete(X,T,TT),
        no_double(TT,Ts).
no_double([],[]).
```

Program 4.9 Topological information retrieval for winged-edge data structure

```
/*--Prolog database for Figure 4.10 ---------*/
line(e1,v1,v2).     line(e8,v4,v8).
line(e2,v2,v3).     line(e9,v5,v6).
```

Figure 4.11 Example for winged-edge data structure



line(e3,v3,v4). **line(e10,v6,v7).**

line(e4,v4,v1). **line(e11,v7,v8).**

line(e5,v1,v5). **line(e12,v8,v5).**

line(e6,v2,v6). **line(end,_,_).**

line(e7,v3,v7).

vert(v1,x1,y1,z1). **vert(v5,x5,y5,z5).**

vert(v2,x2,y2,z2). **vert(v6,x6,y6,z6).**

vert(v3,x3,y3,z3). **vert(v7,x7,y7,z7).**

vert(v4,x4,y4,z4). **vert(v8,x8,y8,z8).**

edge_inf(e1,f1,f2,e2,e4,e5,e6).

edge_inf(e2,f1,f3,e3,e1,e6,e7).

```
edge_inf(e3,f1,f4,e4,e2,e7,e8).
edge_inf(e4,f1,f5,e1,e3,e8,e5).
edge_inf(e5,f2,f5,e9,e1,e4,e12).
edge_inf(e6,f3,f2,e10,e2,e1,e9).
edge_inf(e7,f4,f3,e11,e3,e2,e10).
edge_inf(e8,f5,f4,e12,e4,e3,e11).
edge_inf(e9,f2,f6,e6,e5,e12,e10).
edge_inf(e10,f3,f6,e7,e6,e9,e11).
edge_inf(e11,f4,f6,e8,e7,e10,e12).
edge_inf(e12,f5,f6,e5,e8,e11,e9).

v_first_edge(v1,e1).        f_first_edge(f1,e1).
v_first_edge(v2,e2).        f_first_edge(f2,e9).
v_first_edge(v3,e3).        f_first_edge(f3,e6).
v_first_edge(v4,e4).        f_first_edge(f4,e7).
v_first_edge(v5,e9).        f_first_edge(f5,e12).
v_first_edge(v6,e10).       f_first_edge(f6,e9).
v_first_edge(v7,e11).
v_first_edge(v8,e12).

/------End of database ----------------*/
```

Based on Program 4.9 and Figure 4.11, the following types of queries and responses can be made and obtained

```
?-ei_v(e1,V).
  V = [v1,v2].
?-ei_e(e1,E).
  E = [e2,e4,e5,e6].
?-ei_f(e1,F).
```

```
F = [f1,f2].

?-fi_e(f1,E).

E = [e4,e3,e2,e1].

?-fi_v(f1,V).

V = [v1,v2,v3,v4].

?-fi_f(f1,F).

F = [f2,f3,f4,f5].

?-vi_e(v1,E).

E = [e5,e4,e1].

?-vi_v(v1,V).

V = [v2,v4,v5].

?-vi_f(v1,F).

F = [f1,f2,f5].
```

4.8 Concluding Remarks

The logic programming language Prolog has many features such as unification, backtracking and recursive programming which have made it extensively used for symbolic computations. In this chapter, Prolog has been applied to CAD problem solving. The example problems include edit functions, planar graph traversal, clipping and CAD database management. A common feature of these problems is that they involve few numerical calculations. The implementation of these problems in Prolog has shown that the corresponding Prolog programs are not only concise and readable, but also can be developed quickly with minimum error.

In the case of complex numerical calculations, the author's opinion is that it would be most efficient to build a fast interface between Prolog and a procedural programming language such as Fortran or Pascal. The numerical problems would then be programmed as procedures or functions in the procedural language. The procedures and functions can then be used as predicate expressions in Prolog. A point of difficulty for such an interface would be that the procedural language and Prolog may support integer and real numbers with different precisions.

CHAPTER 5

COMPONENTS OF LOGICAD

The development of the LOGICAD system follows the principle of modularity. In modular programming, changes to a program have only a localized effect. Related to the localization is the principle of *information hiding* [Parnas 1972]. This principle of information hiding may be exemplified by a black box which has certain behavior and effects on the environment but we do not know how these effects are accomplished. Unfortunately in core Prolog niether information hiding nor modularity are directly supported by programming facilities. In the system presented these effects are achieved by careful programming. The LOGICAD system currently consists of five modules. Each module receives data over its input links, processes the data in some way, and then sends it on to other modules over its output links. The interactions between module-module or user-module may also be realized through user commands. This chapter describes all the modules and their interconnections in the LOGICAD system.

5.1 The Architecture of LOGICAD

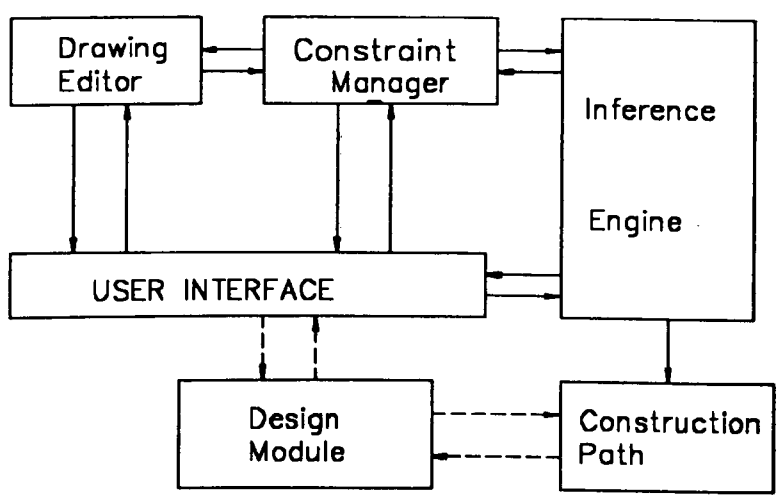
LOGICAD has five main modules

1. A user interface
2. A drawing editor
3. A constraint manager

4. An inference engine
5. A design specification module

All the modules are connected in a way as shown in Figure 5.1. The dotted line in the figure connecting the user interface and design module expresses the fact that the design module needs further development.

Figure 5.1 Structure of LOGICAD



5.1.1 User Interface Module

The LOGICAD system presented in this thesis is an interactive system which employs a declarative specification of geometric shape. As the front end, a user interface module is implemented which takes the form of pull-down and

pop-up menus. Another way of realizing user-computer interaction is through the dialog area in the bottom three lines of the screen. Using this module, commands are organized in a hierarchy of three levels. The user can reach each command by traversing the hierarchical tree structure of the commands. The main advantage of the user interface is its ease of use, flexibility, and information hidden.

The user interface module is composed of six predicates. The first user interface predicate has the form

open_menu(NAME).

This predicate opens a menu defined by the identifier **NAME** which is an integer. Its valid range is from 1 to 10.

The **menu** predicate

menu(NAME,LIST,SEL).

consists of three terms, namely the menu identifier **NAME**, a list of command selections **LIST** and the variable **SEL** which is to be unified according to the user's choices. If there is no pull-down menu, **SEL** is unified with the item selected from the list **LIST**. If a pull-down menu exists, the full instantiation of **SEL** is deferred until the final selection actions are made.

The **pull_down** predicate has the form

pull-down(MAIN,SUBLIST).

The content of the pull-down menu, **SUBLIST**, is displayed just below its ancester selection **MAIN** which is made from the argument **LIST** of the **menu** predicate above. Now the user is restricted either to picking a command from the list **SUBLIST**, or to simply pressing the middle button of the mouse to quit.

The **action** predicate

action(SEL,ACTION).

defines the operation to be made corresponding to the selection **SEL**. If **SEL** is made through a two level interaction, it takes the form *Command&Subcommand.* The infix operator **&** is declared with Wprolog precedence 32. Its meaning here is *part-of.* While the user makes his choices through a mouse or a key board, the interface module responds by traversing the commands hierarchy and organizing the user selected commands by the operator **&**. This predicate expression is where the user's input is implemented by the computer.

The **pop_up** menu predicate is defined as

pop_up(PULL,WINDOW,COMMANDS).

Where **PULL** is the selection made from the pull-down menu, **COMMANDS** is a list of choices contained in the pop-up menu. The user has to define the location and size of the **WINDOW** by specifying the lower left corner **(XL,YL)** and upper right corner **(XR,YR)** coordinates. For example, the pop-up window is defined by

```
pop_window(WINDOW,XL,YL,XR,YR).
```

When the menu **NAME** is no longer needed, it can be removed from the screen by the **close_menu** command.

```
close_menu(NAME).
```

The interaction in the dialog area is controlled internally through the predicate **ask** which is defined as

```
ask(Question,Answer):-
    write(Question), nl,
    read(Answer).
```

The variable **Question** is a text string representing data requested by the system and the variable **Answer** is the user response.

5.1.2 The Drawing Editor

As part of the system, the drawing editor serves as a tool for the creation of the sketched object. The functionality provided includes file management, variable setting, display options, primitive drawing and modification.

File Management

Under the **FILE** selection, commands are provided for the creating, deleting and retrieving of object files. All file names created are stored in a log file.

Each time a new file is created, its name is checked against the stored file names. This task is performed by Program 5.1.

```
check_file(NAME,FILELIST,CHECK):-
        ((member(NAME,FILELIST),
          write('Warning !,File '), write(NAME),
          write(' is already exist. quit/replace:'),
          CHECK = pos);
          (assertz(file(NAME)),
           CHECK = neg)).
```

Program 5.1 Check the existence of a file

If the name coincides with an existing file name, a warning is displayed which asks whether the user wants to quit or to replace. Otherwise, the new name is temporarilly stored in the database. When the **SAVE** command is selected later, the file name is added to the log file by the predicate expression

```
add_file(NAME,FILELIST):-
        wwrite(file([NAME|FILELIST]),log).
```

Primitive Drawing and Modification

The basic drawing primitives provided by LOGICAD are **dot**, **polyline**, **circle**, **arc**, and **ellipse**. Input of these primitives is either through a mouse pick action or joystick movement plus a key press on the keyboard. Unlike other CAD systems, LOGICAD does not support textual coordinates input. Selections **deletion** and **undo** are provided for the modification of primitive elements.

Under these headings, the physical appearance of the drawing primitives can be set. In particular, line type, line color, dot type, dot color, text attributes and fill area pattern can be accessed through these selections.

5.1.3 Constraint Manager

It is often difficult to maintain a consistent set of constraints in the case of geometric applications. In 2D geometry, since each vertex is free to move in both x and y directions, each vertex contributes two degrees of freedom. An object composed of only straight line segments with vertex number v has 2v degrees of freedom. Two items are needed to fix the position of some point on the object and one item is needed to fix its orientation in the plane. Thus 2v-3 items are to be supplied by the geometric constraints [Hillyard and Braid 1978]. When circle and fillet are included as in LOGICAD, they contribute more degrees of freedom. For a circle, three items are needed to fix its location and size. A fillet requires only one item. The extended formula thus becomes

$$DF = 2 \times v + 3 \times c + f \qquad (5.1)$$

where DF is the total number of degrees of freedom, v is the number of vertices, c is the number of circles and f is the number of fillets. In the case of fillet terms, two reference lines for the fillet should not be colinear or parallel. Formula (5.1) is used for the determination of the existence of over- or under-constrained problems. If the total number of constraints is more than

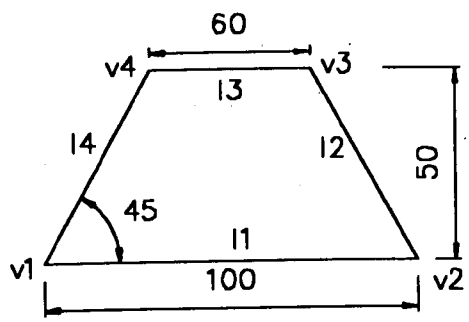
DF, then the geometric object must be over-constrained. If the total number of constraints is less than DF, the object as a whole is under-constrained. The object may also have some parts over-constrained and other parts under-constrained. When the total number of constraints is equal to DF, however, only limited information can be deduced. The geometric object may either be uniquely defined, or have some part over-constrained, or have other parts under-constrained. From this observation, we can see that even using this formula, it is still difficult to know when the constraint scheme is consistent. In the scheme presented in LOGICAD, constraints are checked at entry. This guarantees that only valid constraints are entered. Warnings are thus given when the user tries to input invalid constraints.

Constraint Grouping

To maintain a consistent set of constraints about an object, it is convenient to divide them by constraint types into three sets. Information about each set may then be stored separately. The definitions for the three constraint sets implemented in the LOGICAD scheme are

1. **Angular Set** (denoted AS) is a set of straight line segments which have angular constraints specified. This type of constraint is used to determine the orientation of geometric elements. The constraints such as **angle, paral, perp** and **dist** fall into this category.
2. **Metric Set** (denoted MS) is a set of straight line segments whose length have been constrained. This type of constraint is used to determine the size of geometric elements. Constraints **len** and **radius** fall into this category.

Figure 5.2 A quadrilateral shape



3. **Dependent Set** (denoted DS) is a set of geometric elements with reference to other geometric elements. This type of constraint shows the dependency relationships between geometric elements. DS type constraints in LOGICAD includes **dist** and **dp**.

From the above definitions, it can be seen that the constraint type **dist** belongs to both AS and DS because it contributes information to both of them. Table 5.1 shows the constraints defined in LOGICAD and their categories.

To illustrate the above description about constraint grouping, Figure 5. 2 shows a quadrilateral shape defined by the following constraints.

Constraint Names	Items of Information	Set	Meaning
len	1	MS	length of a line
angle	1	AS	angle between two lines
paral	1	AS	line parallel to a line
perp	1	AS	line perpendicular to a line
dist	2	AS,DS	distance between two lines
radius	1	MS	radius of a circle or fillet
dp	1	DS	distance from a point to a line

Table 5.1 Constraint categories

len(l3,60).

len(l1,100).

angle(l1,l3,50).

dist(l1,l3,50).

From formula (5.1), we can calculate the total number of degrees of freedom for Figure 5.2 by

$$TC = 2*4 + 3*0 - 3 = 5$$

From the calculation, it can be seen that 5 items of independent constraints are needed to uniquely define the shape. Since constraint **dist(l1,l3,50)** implies two items of information as is shown in Table 5.1, the total number of

constraints added is 5. A problem with this approach is that we need to determine whether the list of constraints is consistent or not. In LOGICAD, when the constraint manager is selected, five categories of information are set:

df(FILEID,TC).

as(FILEID,AS,COUNTA).

ms(FILEID,MS,COUNTM).

ds(FILEID,DS,COUNTD).

total(FILEID,COUNT).

representing the total number of constraints **TC**, the number of AS type constraints **COUNTA**, the number of MS type constraints **COUNTM**, the number of DS type constraint **COUNTD** and the total number of constraints **COUNT** supplied by the user respectively. Initially, AS, MS and DS are set as empty lists, and all the counters are set to zero. **FILEID** is the file name of the drawing.

For the example shown in Figure 5.2, if the user's input is the sequence

(1) a length of l1

(2) an angle between l1 and l4

(3) a length of l3

(4) a specification of a distance between l1 and l3

then the corresponding data base change is shown in Table 5.2.

Each time a new constraint is added, it is checked against its set. If the membership check gives the answer true, a warning about redundant constraints is issued. Otherwise its arguments are appended to its set list. In case of the AS set, only the differences with the database are appended. After the appending operation, the corresponding set account is incremented by 1 (or 2 in case of a **dist** type constraint). The total number of constraints added (**COUNT**) is compared with **TC**. If they are equal, the user must stop adding constraints since this means that the added constraints are just enough. If they are not, the process of constraint addition can continue.

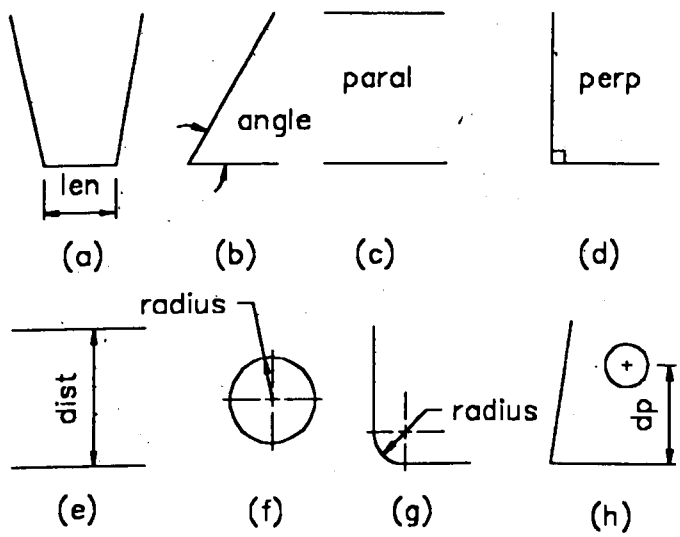
Input	**Constraint**	**AS-set**	**MS-set**	**DS-set**	**TC**
1	length of l1	--	[l1]	--	1
2	angle between l1,l4	[l1,l4]	--	--	2
3	length of l3	--	[l1,l3]	--	3
4	dist between l1,l3	[l1,l4,l3]	--	[l1,l3]	5

Table 5.2 Constraint assignment process

A tricky problem is the internal treatment of constraint type **dist**. Whenever this type of constraint is added, it is appended to both AS set and DS set. In the example, line l1 and l3 are stored in the DS set, and only line l3 is appended to the AS set because line l1 is already stored. No matter how many lines are stored, its account must only increment by 2.

In LOGICAD, the currently available constraint types are **len, angle, paral, perp, dist, radius** and **dp**. Their graphic illustration is shown in Figure 5.3.

Figure 5.3 Constraint types



With these constraint types, much of 2D engineering geometry can be modeled.

5.1.4 Inference Engine

In the inference engine module, new data is generated based on existing constraints and a solution path is found. There are clearly many factors that affect the efficiency of an inference process. The main factors in the author's view are the computing language chosen, the nature of the subject problem involved and the control structure and search method used.

Language

Procedural programming languages such as Fortran or Pascal require the user to explicitly define the execution sequence of an algorithm. Thus this type of programming language can be used to implement well-understood algorithms. On the other hand, Prolog is best known as a logic programming language. Data in Prolog is represented in terms of facts and rules. Unification and backtracking are Prolog's internal implementation mechanisms which are controlled in a depth-first manner. When programming in Prolog, the user has less concerned about how to solve a particular problem than about what is needed to solve the problem. In Prolog, it is possible to implement advanced problem solving techniques [Shenton and Chen 1990].

Nature of the Problem

There are many control strategies (such as forward chaining and backward chaining) and search methods (best-first search or dependency-directed backtracking) which can be found from the AI literature [Rowe 1988, Stallman and Sussman 1977]. However, which method can be used most efficiently depends on the nature of problem involved. In the real world, every material object has a certain shape and every person describes geometric shape in his own way. The diversity of geometry descriptions and object geometries adds to the complexity of the problem.

Control Structure and Search Method

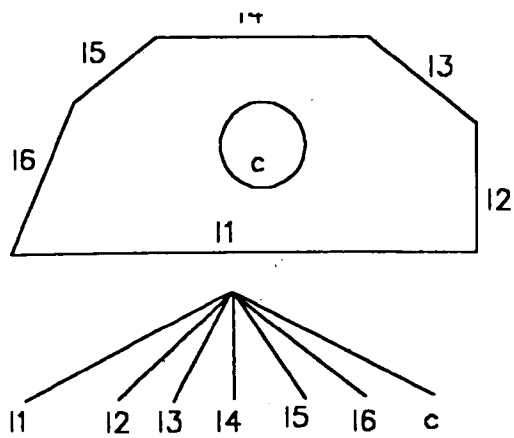
The process of finding a solution path from an initial state to a goal state is called search. Obviously, the search process needs some sort of *intelligent*

control or at least to be based on previous experience (heuristics). The two basic underlying control strategies are depth-first and breadth-first. They are easy to implement but are often inefficient for many problems. If the evaluation function which is a measurement of the difficulty from a certain state to the goal state is taken into consideration, the depth-first search variant is then becomes a *hill-climbing* (sometimes called *discrete optimization*) search; the breadth-first search becomes a *best-first* search [Rowe 1988]. Many more control strategies and search techniques can be found in the literature [Rowe 1988 and Bratko 1986]. In real applications, the choice of a control strategy can enormously affect the success and efficiency of a rule-based system [Rowe 1988].

Problems Solving Strategies in LOGICAD

Problem solving often requires a top-down decomposition process. In LOGICAD, the *divide and conquer* paradigm has been adapted. A 2D shape is divided into its composing lines as shown in Figure 5.4 and the line elements again can be described by parameters such as length of a line or angle between lines etc. During the problem solving process, new information about each composing line may be generated and asserted. Message flow between the lines is made possible by traversing the topological graph and constraint pointers. After all the composing lines have been defined, the object shape is then defined.

Figure 5.4 Object decomposition



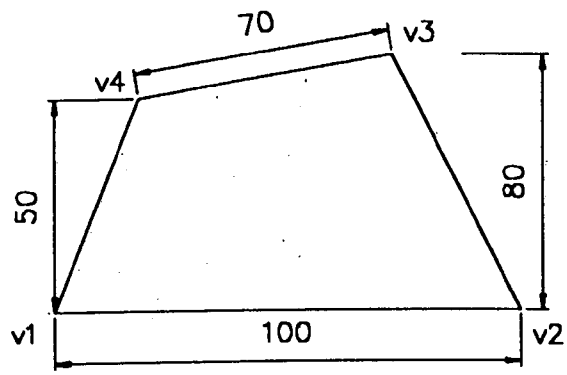
5.2 System Behavior

In this section, the procedures to access LOGICAD are given as a sequence of user commands. An example of how LOGICAD defines geometric shape is given and is compared to the way a conventional CAD system operates.

5.2.1 Initializing LOGICAD

LOGICAD is written in Wprolog. When logged on to the VM/SP operating system, the user can follow the interactions below to access the LOGICAD system. In the explanation, the user typed commands are in italic, and the computer responses are in bold with comment in plain font.

Figure 5.5 Example geometric construction problem



wprolog 2000

where the optional parameter 2000 specifies the execution stack memory size. The unit is in k byte blocks. Currently LOGICAD system needs 90k bytes storage space excluding the numeric procedures. The high demands for memory space comes from the internal Prolog control mechanism

Welcome to Waterloo Prolog 1.7

Copyright 1985 Intralogic Inc

Core Syntax

[shell].

shell.

[logicad].

This command consults the LOGICAD main control file

Once inside Prolog, the character strings must be typed

in lower case and finished with a period.

?-

waiting for user input

logicad.

the user then enters the LOGICAD system.

On the screen, the user is presented with the user interface system menu. He can now select the drawing editor, constraint manager, file management module and inference engine.

5.2.2 Differences to Conventional CAD systems

In a conventional CAD system, the user must firstly plan a sequence of operations for the definition of an object shape, sometimes mannual calculations are required for the sequence input. In LOGICAD, however, the user first draws a sketched object shape and he is then only concerned about what constraints should be added based on the sketch. The order of constraint addition is not important. To show the difference in object definition between LOGICAD and conventional CAD systems, a simple example as shown in Figure 5.5 is given.

Input to Conventional CAD systems

Consider a very simple example object in Figure 5.5. Suppose the coordinates for v1 are x1 = 50 and y1 = 50, and horizontal direction of line v1v2 is zero.

Obviously the relative coordinate and polar coordinate methods can not be sensibly used for this case. There are two ways to construct the object geometry with a conventional, none declarative CAD system.

The first is through arithmetic calculations. The coordinates for v2, v3 and v4 can be calculated using the equations

$$x2 = x1 + 100$$

$$y2 = y1$$

$$x4 = x1 + 50/\tan(40)$$

$$y4 = y1 + 50$$

$$y3 = y1 + 80$$

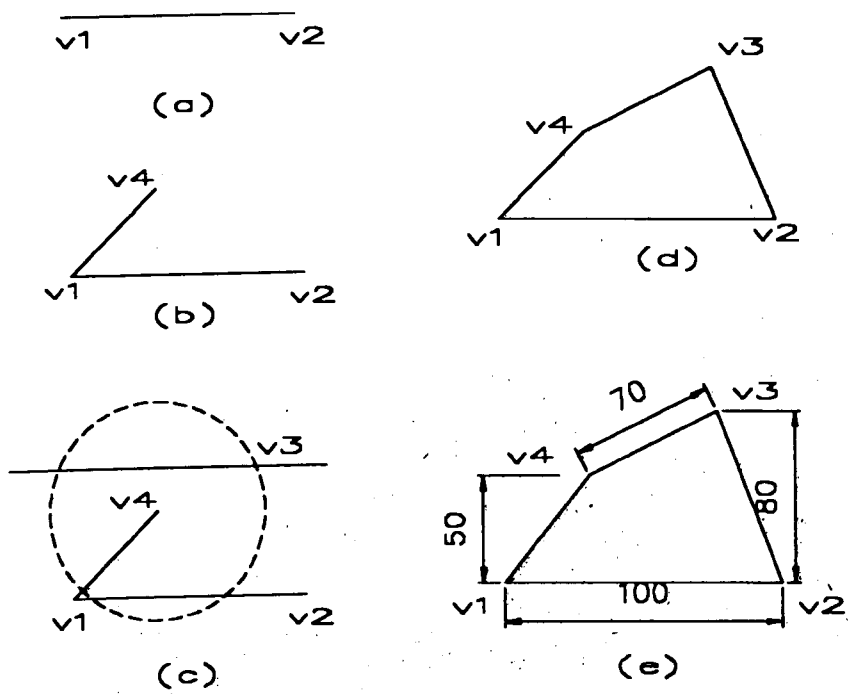
$$(x3-x4)*(x3-x4) + (y3-y4)*(y3-y4) = 70$$

When the coordinates for all the vertices have been calculated, the shape can be drawn through numerical coordinate input.

The second appoach is by geometric construction as shown in Figure 5.6. This method requires the user to have some knowledge of elementary geometry. The construction of the geometry can be made through the following steps :

1. Fix v1 at (50,50), draw a horizontal line v1v2 with length 100.
2. Draw a line parallel to line v1v2 with a distance 50, draw another line passing through point v1 with angle 40 to line v1v2. The intersection between the two lines is v4.
3. Draw a circle with radius 70 and center point v4. Draw a line parallel to v1v2 with distance 80. The intersection between the line and circle is v3.

Figure 5.6 Input by conventional CAD



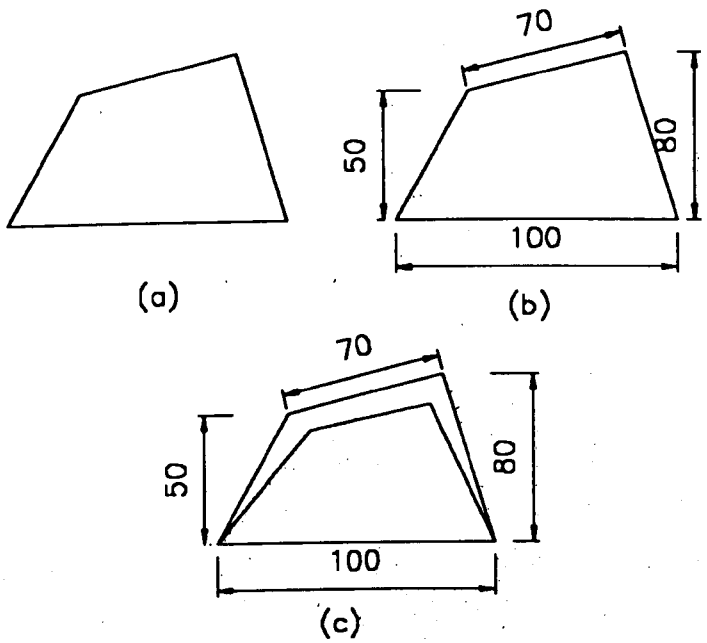
4. Join the points with straight lines and delete auxilliary lines.
5. Add dimensions.

From the above example, it can be seen that, in conventional CAD systems, the user is often required to plan a sequence for shape input, however, this is unnecessary in the LOGICAD system.

Input to LOGICAD

The advantages of geometric input to a declarative system such as LOGICAD can be seen from Figure 5.7. The user first sketches the object topology (Figure 5.7(a)), and then interactively adds constraints (Figure

Figure 5.7 Input by LOGICAD



5.7(b)). The proposed constraint resolution system as implemented in LOGICAD may then compute the precise geometry based on the constraints if required. The order of constraints is unimportant since the system can automatically plan a sequence for constraint satisfaction. This type of declarative input is much easier and more natural for the designer since he does not need to consider at input how the precise geometry is built.

5.3 Concluding Remarks

Current CAD systems are mainly used as passive geometric modelling tools. Although engineering analysis and design tools are beginning to appear in

conjunction with CAD systems, the majority of engineering analysis is still done by independent application programs and most of these programs do not even make use of geometric models or graphics input/output [Jayaram and Myklebust 1990]. The difficulty of integrating the CAD data base with application programs comes from the fact that

> Application program variables are usually in the form of dimensions while the geometry information required by IGES or a CAD system geometry interface is in absolute coordinates relative to a fixed origin [Jayaram and Myklebust 1990]

Thus there is a gap between the user interface levels of CAD systems and application programs. A popular approach to overcome this problem is to manually create a parametric model using a CAD system. The parameterized dimensions are then linked to appropriate application programs. This approach requires user interaction in the creation of parametric models. In the LOGICAD system presented in this thesis, object shape description is in terms of geometric relations and dimensions which are treated as constraints in the system. These data are just what application programs operate on. Thus the underlying techniques as implemented in the system provide a solution for the integration of CAD systems and application programs. In this chapter, the main modules of the LOGICAD system have been described. A novel approach to maintaining a consistent set of geometric constraints has been proposed and implemented.

CHAPTER 6

GEOMETRIC REASONING PROCESSES IN LOGICAD

A major problem confronting the builder of automatic problem-solving systems is that of combinatorial explosion of the search space [Stallman and Sussman 1977]. One way to attack this problem is to develop effective search mechanisms to reduce the search space. Another way is to represent the problems and their solutions in such a way that combinatorial searches are self limiting, that is, to formulate the domain concerned so that there is a finite set of *slots* to be filled [Stallman and Sussman 1977]. In this chapter, a novel knowledge representation scheme called geometric element based reasoning (GEBR) which can significantly reduce the search space for geometric reasoning is presented and elaborated. The proposed scheme treats the composing elements of a geometric object as variables. Each variable may have some attributes attached to it during the constraint propagation process. For example, a variable of type *circle* may have its radius or centre coordinates attached to it. The attached attributes are deduced facts in the inference process. If the attributes of a variable match a predefined pattern (a rule), the variable is then relaxed and is deduced to be known. When all variables of a geometric object are deduced as known, the geometric object is then also known and can be constructed. A method called *dependency-directed backtracking* [Stallman and Sussman 1977] is used for the propagation of constraints. The method "threads deduced facts with a justification which mentions the antecedent facts and the rule of inference used". The overall

control of the constraint propagation process is based on the *rule-cycle hybrid* method [Rowe 1988] which applies rules in database order as backward chaining and uses each rule in a forward chaining way to assert new facts. It is demonstrated in this chapter that the proposed scheme is efficient and practical.

6.1 The Computational Model

Current CAD systems largely use the representation techniques of wireframe, B-rep and CSG. These representations are useful in shape display and blueprint production. However, as computer applications have expanded to more diverse fields including engineering design, advanced manufacturing technology etc, there has developed a strong need to integrate the various "Islands" of automation on product viewpoints [Palframan 1985]. Current representation techniques lack applicability and application independency [Kawagoe and Managaki 1984].

In order to overcome such deficiencies, a parametric object model is proposed by Kawagoe and Managaki [Kawagoe and Managaki 1984] which has the following features:

1. Parameterized dimension manipulation ability
2. High-level independency
3. Motion representation ability

In their proposed model, a formal statement for a parametric object class S is given by the six tuple expression

{T,D,O,V,R,C}.

where T is the set of topologies which also consists of the possible topologies, D is the set of dimensions which consists of the possible geometry dimensions, O is the set of operations performed in sequence, V is the set of variables which can be defined for all parametric objects, R is the collection of relationships between V and T/O/D and C is the set of constraints which represents the limitations of parametric space spanned by variables and of parametric objects.

This proposal is later adapted by Krishnan and Patnaik in their shape design system GEODERM [Krishnan and Patnaik 1986].

Although the model is suitable for some applications, it has the inherent defects of parametric design such as

1. No access to undeclared variables.
2. A requirement for the user to plan a sequence of equation solving operations when defining the script.
3. Unidirectional constraints, that is constraints created by means of a parametric system which typically imply causality.

In this thesis, a simpler geometric representational model $\mathscr{R}$ is proposed and implemented. The model only has three components :

{T,D,(C)}.

Where T (topology) is a non-empty set of geometric object types of arbitrary nature. In the current implementation of LOGICAD, T includes only

straight lines, circles and arcs. D (dimension) is a finite set of constraints (dimensions) defined over the set T and C (constraints) is an optional limit on the range of constraint instances specified in set D.

The proposed scheme accordingly only requires the user to work on the simpler aspects of an object definition. The three components provide less information than the parametric model proposed by Kawagoe and Managaki [Kawagoe and Managaki 1984]. It is the system's job to generate more information based on production rules. Thus after the inference process, information relevant to the construction of the geometric model is added. The processed model $\mathscr{R}$ now contains the 4-tuple

$$\{T,D,(C),OP\}.$$

Where OP (operations) comprises the deduced causal relationships among the variables of a geometric object which is similar to a *macro* program in parametric systems. It is clear that the model has the advantages of the dimension manipulation ability of the parametric system but without the need for the user to predefine the script of an object.

This model is compatible with the existing two schemes of CSG and B-rep. A CSG type solid can be represented by the expression

$$Mcsg = \{\{0\},\{0\},\{0\},\{op\}\}.$$

This expression says that there are no topological or dimensional constraints explicitly defined for CSG type object, but that these are only implied in the boolean operations. The sequence of operations {op} defines a CSG tree from which the topology and dimensions of the object can be deduced. From this

model, it can be seen that the user needs not worry about the topology or geometry of an object. He only needs define a sequence of boolean operations.

In contrast with CSG solids, a B-rep object is explicitly defined by its geometry and topology. The expression is

Mb-rep = {{t},{d},{0},{0}}.

This expression states that B-rep objects include topology which are relations between vertices and edges, vertices and faces, edges and faces. The dimensional information is calculated by the vertex coordinates. This model shows that the user is required to describe the object topology and to provide precise vertex coordinates.

From the above description, it can be concluded that the model $\mathscr{R}$ is a complete representation of object information. Compared with B-rep, the model $\mathscr{R}$ does not require the user to precisely input the topology and geometry of an object. As compared with CSG and the parametric model, $\mathscr{R}$ can relieve the user from the explicit input of object construction sequences.

6.2 Representation of Object Topology

The shape of an object is defined by its geometric and topological information. In B-rep, all this information must be explicitly input by the user. In current CAD practice, not all the topological information needs to be stored explicitly since some information is dependent on other data and there is generally a trade-off between memory space and computation speed [Woo 1985]. Thus efficient data structures which allow quick access to geometric

and topological information are required [Woo 1985]. The input of precise geometry, however, presents a difficult task for the user because he must have a command of elementary geometry and is responsible for planning the sequence of input. In many CAD systems, the input of geometry and topology is actually performed at the same time. This makes the input of object shape even more difficult.

An objective for the development of declarative and constraint based systems is to ease the user's burden in the input process. Almost all existing systems of this type are devoted to 2D applications. Most of the systems use a set of characteristic points or control points to represent an object [Lin et al 1981, Nelson 1985]. Thus some tedious and non-practical point operation commands such as "on-line" and "end-point" are provided for the definition of object shape. These are necessary because the underlying techniques of such systems only support point set manipulations. In the proposed system implemented in LOGICAD, the system uses a line element based set to represent object topology and geometry by means of a directed graph (*digraph*) method.

6.2.1 Graph Structure

The graph is a very general kind of data structure that can be used to represent numerous situations. It may be portrayed as a set of points with connecting lines. This suggests that a graph is basically a geometric object [Smith 1987]. This interpretation may be misleading since the nodes and arcs of a graph structure can have some special non-geometric meanings.

With graphs we are concerned with two sets of entities. The first is the set of vertices or nodes $V = \{v_1, v_2, ..., v_n\}$ and the second is the set of edges or arcs $E = \{e_1, e_2, ..., e_n\}$ which connect pairs of vertices. Edges may or may not be directed. The directed graph is called the *digraph*.

The use of the graph structure in the geometric reasoning and representation problem is determined by the nature of the geometric problem itself. In the proposed scheme as implemented in LOGICAD, constraints are multi-directional, thus the graph structure can represent the constraint network more naturally than other data structures. Since object topology refers to the connectivity of geometric elements, this again suggests the use of a graph structure.

For a graph representation, there are three basic information retrieval operations. They are

1. **successors(V,SUC)**, to locate all vertices pointed to by V,
2. **predecessor(V,PRED)**, to locate all vertices pointing to V,
3. **vertice(E,Vi,Vj)**, to locate the endpoints **Vi** and **Vj** of edge **E**.

Implementation of these basic operations in Prolog is straightforward. Some higher level operations such as inserting and deleting edges and vertices are subsequently defined on top of these basic operations.

6.2.2 Graph Representation of Object Topology

The usual choices for representing a graph structure in computer systems are to use either a set of lists or an array. Since Prolog only support list oper-

ations, LOGICAD uses the list to represent information about constraints and object topology.

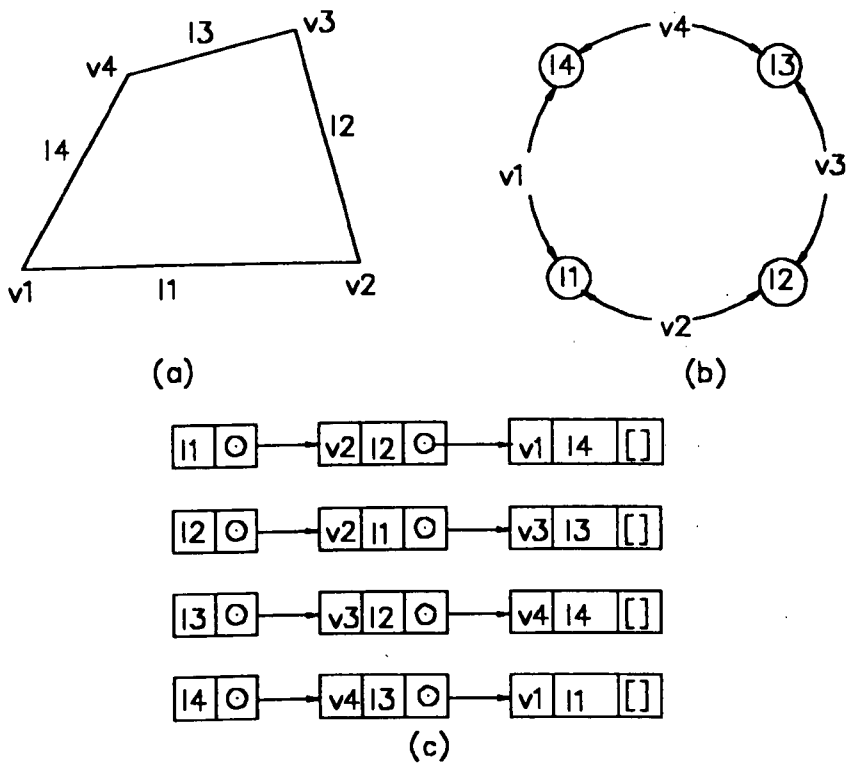
As an example, Figure 6.1(a) shows a quadrilateral shape. Its topology is represented as a digraph shown in Figure 6.1(b). The set of lists for the topology of Figure 6.1(a) is given in Figure 6.1(c). As can be seen from the figure, the edges and vertices in the physical shape are represented as vertices and edges respectively in the digraph. The digraph is thus the 'dual' graph of the object regarded as a graph. This is more natural because vertices imply topological information. In the list, each vertex header points to all its connected vertices and edges. The vertices of the shape which are represented as edges in the digraph are mainly used to trace topological information. With this kind of representation, it is easy to find from the data base that line l4 is connected to line l3 through vertex v4.

Depth-first and breadth-first search processes can be used in the graph search subject to two complications: we may encounter some vertices more than once in our search while not encounter other vertices at all. In order to solve the first problem, we must mark the vertices when they are visited in order to avoid infinite loops. The second problem is solved by looking for unmarked vertices, and then initiating further searches as needed. Every time we initiate a new search, we construct a new search tree in the graph.

6.3 Preprocessor

The completion of the sketching process defines the topology of the object, that is, defines the set {t} which is then explicitly stored in the data base. Its

Figure 6.1 Graph representation of object topology



topological graph can now be drawn as in Figure 6.1(b). Each node represents an edge of the object and the vertices serve as pointers (arcs) which make access to the topological information much easier. At this stage, it is necessary to set up an empty list structure to accomodate constraint information. This is done through the following two steps:

(a) Calculate LEN = |t| (the length of {t}),

(b) Set a list of empty lists with length LEN.

The purpose of the two operations is to assign each line element a place to store its own constraint information. This also facilitates the addition, deletion and alteration of constraints.

The addition of constraints is through the constraint manager as described in Chapter 5. Each time a constraint is added, it is checked against the relevant set information (AS, MS, DS) for consistency. If it is already stored in the data base, or if the addition of the constraint may cause redundancy, a warning is given by the system. If the constraint is not in the data base, the constraint is stored. The process goes on until a consistent set of constraints have been accumulated or until the user wishes to stop the process.

Since the constraints supported in LOGICAD are axis independent, the object description is thus also axis independent. This property eliminates the demand for precise coordinate descriptions of the object. When we wish to show the shape of the object on the screen in a particular coordinate system, what we need to do is to fix a point and the orientation of the object. This is a more natural way of object description than that which conventional CAD systems offer.

6.4 Constraint Satisfaction

Constraints have a dual nature because they can represent both descriptions and commands. Since constraints reflect the diversity of the physical world, there is no single method of representation that can meet the requirements of universal problem solving in the real world. In this thesis, we deal mainly with geometric constraints. A method similar to the idea of dependency-

directed backtracking [Stallman and Sussman 1977] is proposed and utilized for constraint propagation. The rule-cycle hybrid method [Rowe 1988] is used to control the constraint propagation process.

Now a shape may be composed of hundreds of line elements. These line elements may be arranged in numerous ways. It is very difficult to define rules for each arrangement. For example, Aldefeld [Aldefeld 1988] defines constraint propagation rules based on elementary geometry. Figure 6.2 (a) shows an arrangement of two lines and one circle. Typically he defines a rule for this arrangement as

condition : (Tangent mc-)
(Angle lm-)
(C-Coeffs c-)
(L-Coeffs l-)
conclusion : (L-Coeffs m*).

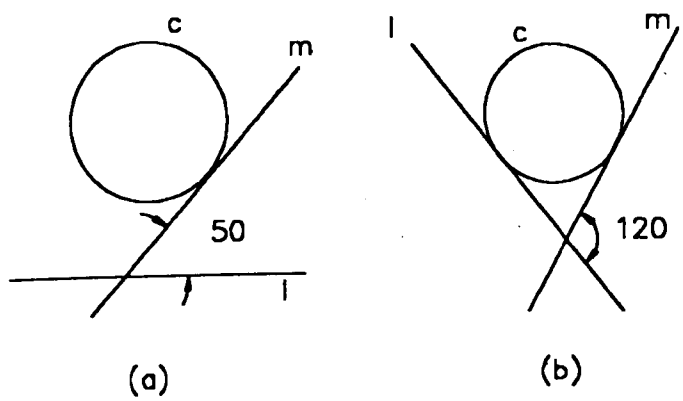
However, the known conditions may change depending on applications. Thus based on Aldefeld's method, we can also define a rule about the same geometric arrangement as the rule

condition : (Tangent mc-)
(Angle lm-)
(Direction l-)
(C-Coeffs c-)
conclusion : (L-Coeffs m*).

More rules may be defined for this arrangement with different conditions and conclusions. Another difficulty in addition to that of multiple rules for the same geometric arrangement, is that, with the same line elements, there may be many overall arrangement (Figure 6.2(b)). Rules are needed to define the new arrangements. The problem arising in this observation is "the completeness of rules and their optimal design" [Aldefeld 1988]. Given a shape which is composed of N line elements with the average attributes for each line element as M, then the possible maximum number of different shape arrangement is N!. The number of combinations of all the attributes is (M*N)!. Thus the maximum number of rules is N! * (M * N)!. This may be a significant number especially when the number of line elements increase.

Although there may be numerous line elements involved in a shape, there are only a limited number of line types (such as straight lines, circular lines) and each line type may only be defined in a finite number of ways. In LOGICAD, geometric elements are divided into line types, and rules can be defined for each line type. Geometric constraints are also grouped into two categories. The first is that of local constraints such as the length of a line or radius of a circle and the second is that of the constraints that have a dependency relationship among the constrained variables such as the angle between two lines or the distance of two lines. The dependency relationships (constraints) between variables (line elements) are represented as pointers (represented as arcs in the graph structure). Suppose a shape has N line elements. If the N line elements may be categorized into P line types and for each line type, the number of average attributes are M. Then the possible maximum rules are M! * P! *N!. By this form of data-typing, the number of rules can be greatly reduced, and the efficiency of deduction may be im-

Figure 6.2 Same line elements with different arrangement

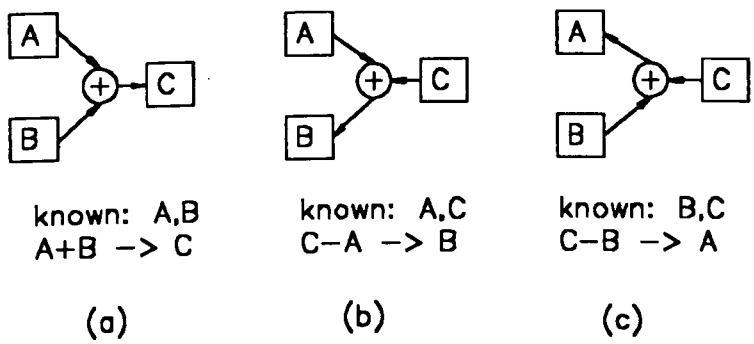


proved. This is of course a primary motivation behind general data-typing in most computer systems.

It is up to the user to specify the constraints (descriptions) on an object, but it is up to the system to satisfy them. Satisfying constraints is not always trivial since constraints are typically multidirectional. For example, the 'plus' constraint shown in Figure 6.3 should take three possible variants into consideration. Thus one task of the system is to decide among the possible choices which one is to be chosen for the particular context.

Constraint computation in LOGICAD is performed by a series of one-step deductions called constraint propagation [Stallman and Sussman 1977]. It is

Figure 6.3 The "plus" constraint



initiated when a change to a constrained variable occurs. This propagation activates the constraints which constrain the variable and causes new descriptions to be added to all other related variables to preserve the affected relationships. When propagated to the rest of the constraint system, these changes cause other constraints to be activated. Constraint propagation consists of repeatedly identifying the activated constraints, deriving new descriptions for the constrained variables and propagating the changes to the rest of the constraint system. The process ends when no new description is generated in a rule cycle. Constraint propagation implements a kind of antecedent reasoning and efficiently exploits the dependencies in the con-

straint system to derive a system-wide consistent solution [Stallman and Sussman 1977].

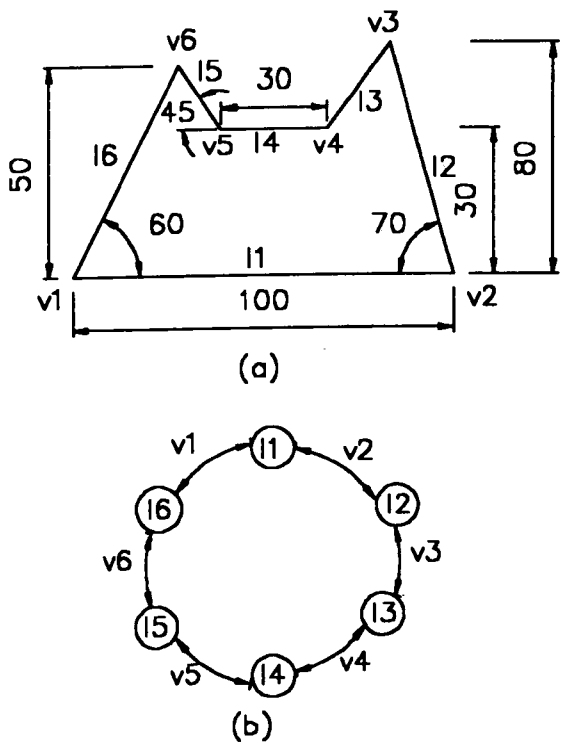
6.4.1 Angular Constraint Resolution

All constraints which fall into the AS set are treated as angular constraints. These include **angle, perp, paral** and **dist**. Since all these constraints have the same characteristics which constrain the orientation of geometric elements, a single method is used for the resolution of this type of constraints.

An exception is for the internal representation of constraint type **dist** since **dist** type constraints belong to both the AS set and the DS set. The constraint **dist** can be thought of as implying parallism and equation dependency because if one of the lines in the constraint is known, the equation of the other line can be deduced and vice versa. Thus in the proposed scheme, when the constraint **dist** is added, it is treated internally as constraints **paral** and **eqn**. Because it is unusual for the user to use line equations to define an object shape, the constraint type **eqn** is only used internally by the system in the constraint propagation process.

Figure 6.4 shows a constrained polygon (Figure 6.4(a)) and the corresponding topological graph (Figure 6.4(b)). In LOGICAD, dimensions are treated as geometric constraints. Their internal representation in the system can be seen from Figure 6.5 and Table 6.1. Figure 6.5 is the constraint graph, it shows the relationships among the line elements. From the figure, it can be seen that angular constraints are multidirectional. Table 6.1 is the internal Prolog representation. Each line segment is assigned a list to store constraint infor-

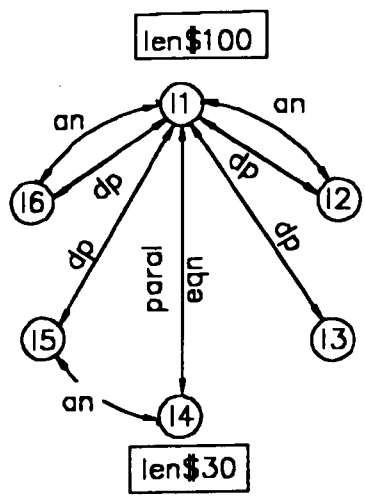
Figure 6.4 Example problem one



mation relevant to itself. The operator ":" is a pointer while operator "$" indicates value.

If the shape is to be displayed on the screen, a point on the object and the object orientation must be fixed. These constraints relate the object description with a particular coordinate system. The added information is stored internally in a separate list. When the inference process starts, this information is passed to the relevant elements by traversing the topological graph. In the example shown in Figure 6.4, the vertex v1 may be accordingly

Figure 6.5 Constraint graph for example one



fixed at (10,10) with the direction of the line l1 at zero degrees to the X axis. That is

[v1$10$10,l1$0].

By traversing the topological graph, vertex v1 is appended to l6. Both vertex v1 and the direction of l1 are appended to l1. The internal representations of l1 and l6 are thus changed to the following:

l1 [v$1,d,len$100,an: l6$60,an: l2$70,paral: l4,eqn: l4]

l6 [v$1,an: l1$60,lenr: l1$50]

During the constraint propagation process, the information flow may be facilitated with the help of the topological graph of the object. In this example, information about vertex v1 is easily passed to its two connecting line elements l1 and l6.

Since line l1 has received user input referring to the object orientation, the angular resolution process starts from l1. From the data base (Table 6.1), it can be seen that l1 points to l6, l2 and l4 through angular constraints. That is when the orientation of l1 is known, the orientations of the pointed line elements can be deduced from l1. The deduced information is stored in the data base in the form

deduced(1,d,user: [l1]).

deduced(2,d,l1: [l6,l2,l4]).

Each deduced fact is asserted to the data base and has its antecedent recorded. By so doing, the complete memory of how every fact is deduced is recorded. The recorded memory is useful in handling contradiction. A contradiction occurs when the deduced facts are inconsistent with the previously added facts. The system can then backtrack from where the contradiction occurs. By scanning backward along the chains of deductions, it is also easy to provide explanations about the deduction process to the user.

The angular resolution process then proceeds from the known list **[l6,l2,l4]**. At this point, however, a cyclic process occurs since angular constraints are multidirectional. Thus it is necessary to rearrange the element information after each one-step deduction. In LOGICAD, the rearrangement follows the rule

Elements	Database
l1	**[len$100,an: l6,an: l2,paral: l4,eqn: l4,dp: l2,dp: l3,dp: l5,dp: l6]**
l2	**[an: l1,dp: l1$v3]**
l3	**[dp: l1$v3]**
l4	**[len$30,paral: l1,eqn: l1,an: l5]**
l5	**[an: l4,dp:l1$v6]**
l6	**[an: l1,dp: l1$v6]**

Table 6.1 Database for example one

IF direction (d) of L1 is known &

L1 points to L2 through angular constraints

THEN delete **an (paral, perp) : L2** from L1 &

delete **an (paral, perp) : L1** from L2 &

assert d to L2.

This rule ensures that the resolution process will not go into an infinite loop. Starting from the list **[l6,l2,l4]**, no information can be deduced from lines l6 and l2. From l4, however, the direction of l5 can be deduced. The process then attempts to proceed from list **[l5]**, but no more can be deduced. After the angular resolution process, the sum of the deduced facts are stored. These being

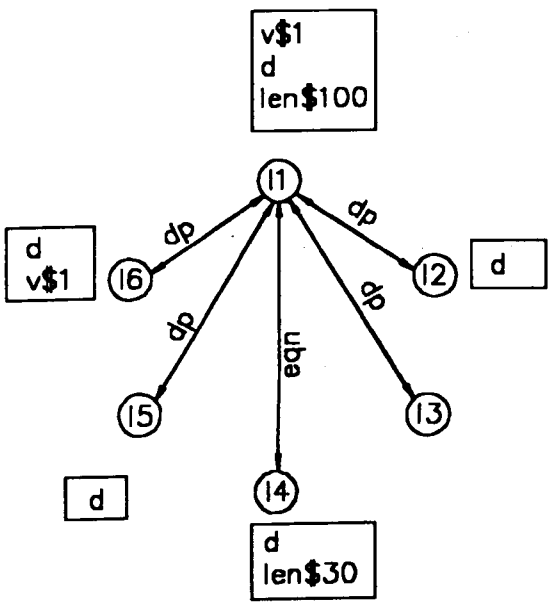
deduced(1,d,user: [l1]).

deduced(2,d,l1: [l6,l2,l4]).

deduced(3,d,l4: [l5]).

The data base accordingly changes to that of Table 6.2. The corresponding constraints graph change is shown in Figure 6.6.

Figure 6.6 Constraint graph of example one



The order of the deduced facts is important since it is used to dictate the geometric construction. The order is also useful for quick adaptation of the constraint system to a constraint change because the deduced facts maintain the dependency relationship. For example, if the direction of l1 is changed, all the other line elements are affected. But if the angle between l4 and l5 is changed, it is easy to see that only the direction of l5 is affected. There is no

need to redo the whole constraint propagation process. This minimizes the effort to adjust the constraint system to the changed circumstance.

Elements	Information
l1	[v$1,d,len$100,eqn: l4]
l2	[d,dp: l1$v3]
l3	[dp: l1$v3]
l4	[d,len$30,eqn: l1]
l5	[d,dp: l1$v6]
l6	[d,dp: l1$v6]

Table 6.2 Database after the resolution of angular constraints

6.4.2 Constraint Propagation

When all angular constraints have been relaxed, the resulting constraint graph is as shown in Figure 6.6. Now the constraint propagation rule module is initiated. Rules are represented in terms of lists and are pattern-invoked [Chan and Paulson 1987]. The control mechanism used for the constraint propagation process is the 'rule-cycle hybrid' method which combines forward and backward chaining. With the 'rule-cycle hybrid' process, rules are tried in data base order as with backward chaining, but each rule is used in a forward chaining way to assert new facts. If a particular rule pattern matches the data stored in a line element, some new facts are generated and added to the line data list. Information relevant to other line elements are passed through by tracing the topological graph and the constraint pointers.

After each successful match, the system traces the line elements which have received new data because these elements have the potential for a successful match. These elements are then invoked. If they fail, the system then backtracks to their antecedent. Each time a data list has a successful match or receives new information, it is marked. Each data list may only be marked once per rule cycle. The process goes on until no successful match can be made with the rule. At this point, a new rule pattern is invoked to repeat the matching process. The whole process is repeated until the last rule is used, then a new cycle starts from the first rule, but this time, only the marked data list is invoked and matched with the rule pattern. The match between a rule pattern and a data list may not be exact by equality but may be by inclusion since forward chaining sometimes generates surplus information. Whenever the pattern of a rule is a subset of the data list, the match is successful, otherwise a new match is initiated. The propagation process may stop in two cases: firstly if all line elements have been deduced as being known; secondly if no more new facts can be generated in a rule cycle.

In the propagation process, the pointers attached to the constraint types **eqn** and **dp** are processed according to the rules

IF L1 and L2 are related by **eqn** &

line L1 (or L2) is known

THEN delete **eqn: L2 (eqn: L1)** from L1 (L2) &

change **eqn: L1 (eqn: L2)** to **eqn** in L2 (L1)

and

IF L1 is known &

dp: L2

THEN delete **dp: L2** from L1 &

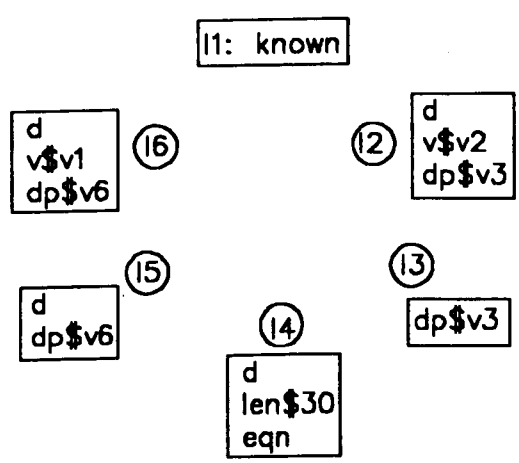
change **dp: L1$V** to **dp$V**

6.5 Examples Showing the Constraint Propagation Process

6.5.1 Example One

Figure 6.4 shows an object with geometric constraints. The internal representation of the constraints are shown in Table 6.1. Figure 6. 5 is a graph of the constraint relationships. After angular constraints have been resolved, the resulting constraint graph is shown in Figure 6.6.

Figure 6.7 Line l1 is deduced to be known



Starting from the specification shown in Figure 6.6, the rule :

rule 1 IF one vertex is known &

direction is known &

length is known

THEN line element is known

is invoked. This rule is represented internally as

r(1,l,[vV,d,len_]).

where the first term is the rule number, the second term shows what can be deduced and the last term is a list of conditions required for the deduction. The rule can match successfully with the data already stored for line l1, thus l1 is deduced to be known. The **dp** type and **eqn** type constraints which point to l1 are relaxed and **v$v2** is passed to l2. After this match, the corresponding constraint graph is shown in Figure 6.7.

Since the rule has no more successful matches, another rule from the rule base is invoked, this rule can be stated as

rule 2 IF one vertex is known &

direction is known &

distance from the unknown vertex to a line is known

THEN line element is known

The internal representation of the rule is

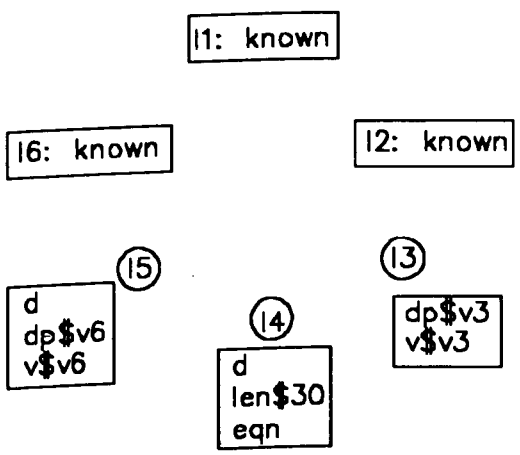
r(2,l,[v$V1,d,dp$V2]).

This rule pattern can match successfully with the data lists of l2 and l6 as can be seen from Figure 6.7. After the match, new facts about l2 and l6 are asserted and data relevant to other data lists are passed through. The data

base is then changed to a new state (see Figure 6.8). The constraint propagation process continues. This time, only the rule

r(4,eqn,[d,v$V]).

Figure 6.8 Lines 12 and 16 are known



can be unified with the data list of 15. The rule says that if the direction and an end point of a line are known, then the equation of the line can be deduced. Thus, after the match, **eqn** is asserted to the data list of 15 and the data base changes to that of Figure 6.9. From Figure 6.9, The rule

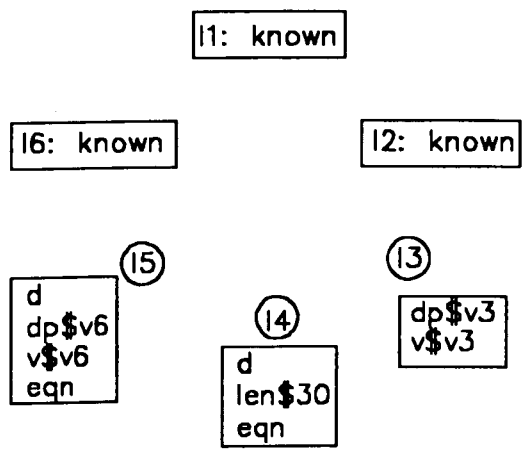
r(5,v,[[eqn],[eqn]]).

can match successfully with the data lists of 14 and 15, thus the intersection of 14 and 15 which is v5 can be deduced and appended to both the data lists of 14 and 15. Now the data base changes to that of Figure 6.10. It is easy to see that rule 1 has a successful match with the data list of 14. Thus 14 is deduced to be known and v4 is appended to the data list of 13. At this point

shown in Figure 6.11, the data lists of l3 and l5 can both be unified with the rule

r(3,[v$V1,v$V2]).

Figure 6.9 eqnation of l5 is deduced



Now all the line elements of the object are deduced to be known. The object itself can therefore be constructed. The whole constraint propagation process is shown in Table 6.3. The following is a list of all the geometric rules supported in the LOGICAD system.

r(1,l,[vV,d,lenL]).

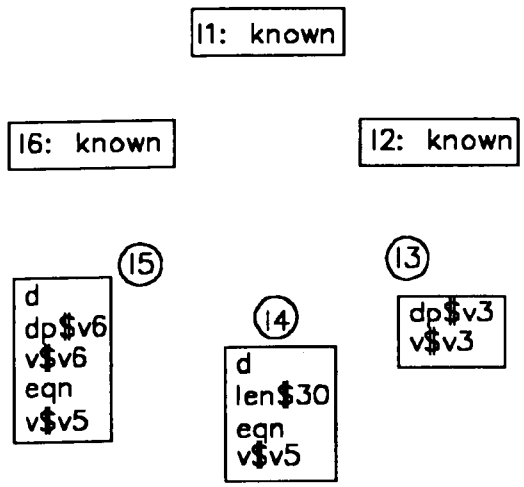
r(2,l,[v$V1,d,dp$V2]).

r(3,l,[v$V1,v$V2]).

r(4,eqn,[d,v$V]).

r(5,v,[[eqn],[eqn]]).

Figure 6.10 Vertex v5 is deduced



r(6,v,[[v$V1,len$_],[v$V2,len$_]]).

r(7,v,[[dp$V],[eqn]]).

r(8,v,[d,dp$V]).

r(9,f,[rad,L1,L2]).

r(10,c,[rad,dpV,dpV]).

r(11,c,[v$V1,v$V2,v$V3]).

r(12,c,[v$V1,v$V2,rad]).

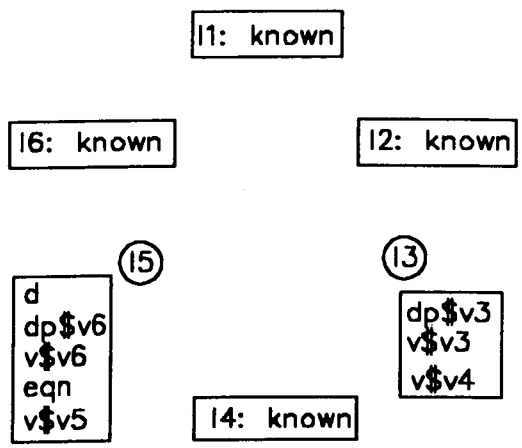
6.5.2 Example Two

This example shows the constraint propagation process of a geometric shape with circle and fillet as shown in Figure 6.12. The topological graph and constraint graph of the object are shown in Figure 6.13 and Figure 6.14 respectively. The initial constraint specifications are stored in the form shown in Table 6.4.

Sequences	Rules used	Deduced information
1	1	l1 is known
2	2	l2 and l6 are known
3	4	equation of l5 is known
4	5	intersection of l4 and l5 is known
5	1	l4 is known
6	3	l3 and l5 are known

Table 6.3 Constraint propagation processes of example one

Figure 6.11 Line l4 is deduced



Elements	Database
l1	**[len$100,paral: l3,eqn: l3,perp: l2,dp: l6,dp: l7,dp: c]**
l2	**[perp: l1,dp: c,paral: l4,eqn: l4]**
l3	**[paral: l1,eqn: l1,paral: l5,eqn: l5,f: f1]**
l4	**[paral: l2,eqn: l2,paral: l6,eqn: l6,f: f1]**
l5	**[paral: l3,eqn: l3,f: f2]**
l6	**[paral: l4,eqn: l4,f: f2,dp: l1$v7]**
l7	**[dp: l1$v7]**
c	**[dp: l1,dp: l2,rad$20]**
f1	**[f: l3,f: l4,rad$10]**
f2	**[f: l5,f: l6,rad$10]**

Table 6.4 Database of example two

A **fillet** type constraint is dependent on its two reference lines. During the constraint propagation process, whenever a reference line is deduced to be known, the information is passed to the data list of the relevant **fillet** type constraint. If the two reference lines and the radius of the fillet are both known, the fillet can then be deduced. This description is represented as rule number 9 in the rule base.

Figure 6.12 Eample problem two

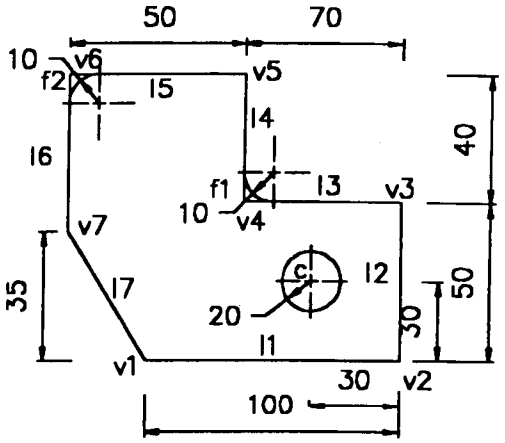
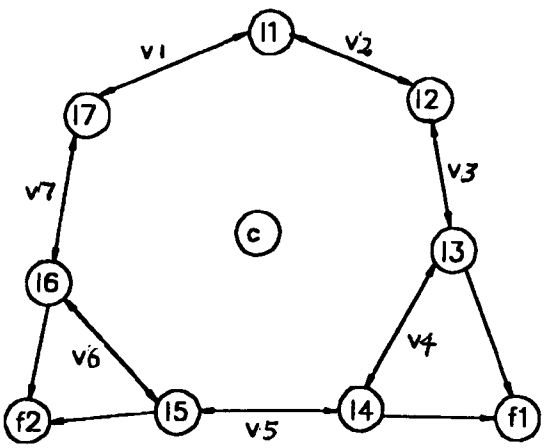


Figure 6.13 The topological graph of example two



In order to achieve coordinate system independent descriptions of shapes, geometric constraints should not refer to x and y coordinates. A circle may be defined by its centre position and radius. The centre position is usually defined in terms of x and y coordinates in conventional CAD systems. In the LOGICAD system, however, the centre position of a circle is deduced from the constraint propagation process.

In the example problem, suppose the direction d and end vertex v1 of line l1 are fixed. This information is internally appended to the data list of l1. When propagated to the rest of the data lists, the following facts are generated and asserted

deduced(1,d,user: [l1]).

deduced(2,d,l1: [l2,l3]).

deduced(3,d,l2: [l4]).

deduced(4,d,l3: [l5]).

deduced(5,d,l4: [l6]).

After this process, angular constraints are relaxed. The resulting data base is thus changed to that of Figure 6.15 The process proceeds from the data base shown in Figure 6.16 with rule 1. The rule can match successfully with the data list of l1. Thus l1 is deduced to be known. Since l1 points to l6, l7 and c by the constraint pointer **dp** and to l3 by **eqn**. These constraints are resolved and new data are appended to their corresponding data list as shown in Figure 6.16. Now Figure 6.16 shows the current data base. This time, rule 4 can match the data list of l2 therefore new data about the equation of l2 is deduced and appended to l2. Since l2 has received new data, it is marked. When propagated to the rest of the data lists, more new data is generated as shown

in Figure 6.17. The process goes on with distinct stages in the data base as shown in Figure 6.17 through to Figure 6.23. The whole constraint propagation process, rules applied and new data asserted are listed in Table 6.5.

Sequences	Rules Used	Deduced Information
1	1	l1 is known
2	4	equation of l2 is known
		intersection of l2 and l3 is known
3	5	intersection of l3 and l4 is known
		intersection of l4 and l5 is known
		intersection of l5 and l6 is known
4	3	l2,l3,l4 and l5 are known
5	8	fillet f1 is known
6	9	circle c is known
7	7	v7 is known
8	3	l6 and l7 are known
9	8	fillet f2 is known

Table 6.5 Constraint propagation processes of example two

From the above descriptions, we can see that the constraint propagation process consists of repeatedly identifying the activated constraints, deriving new data for the constrained variables and propagating the change to the rest of the constraint system.

Figure 6.14 The constraint graph of example two

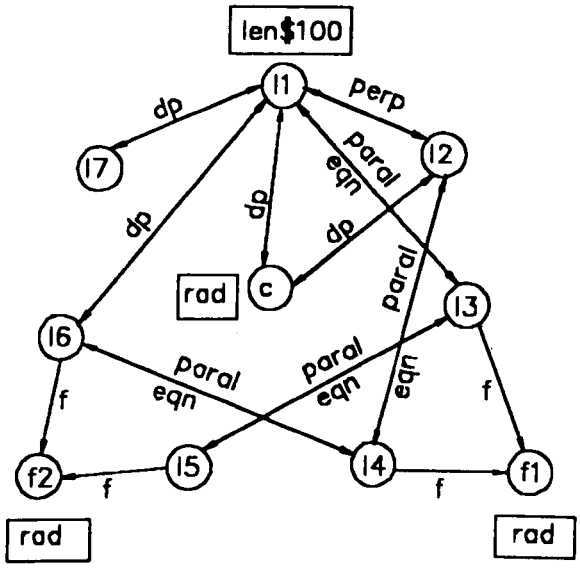


Figure 6.15 Fix a point v1 and the direction of l1

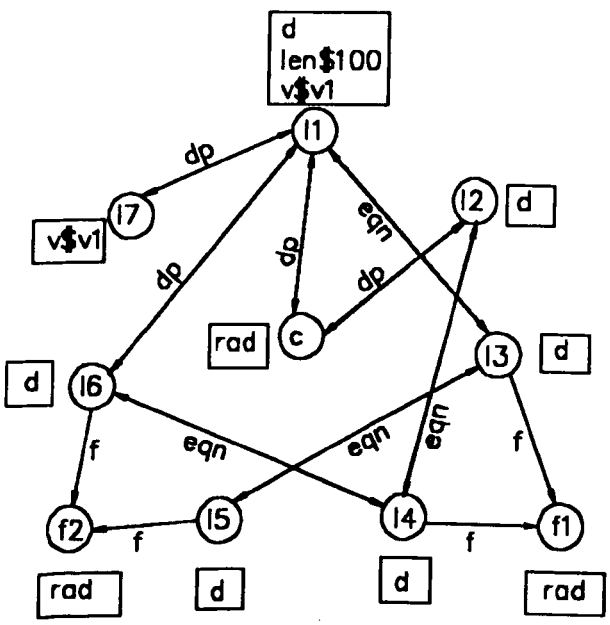


Figure 6.16 Line l1 is deduced

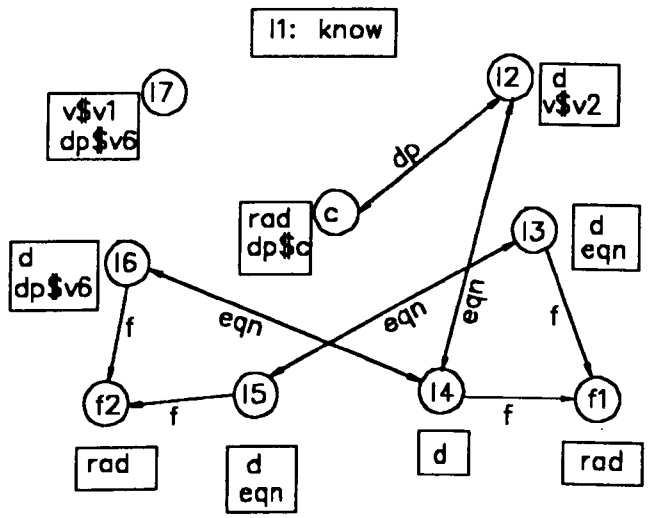


Figure 6.17 Equation of l2 is deduced

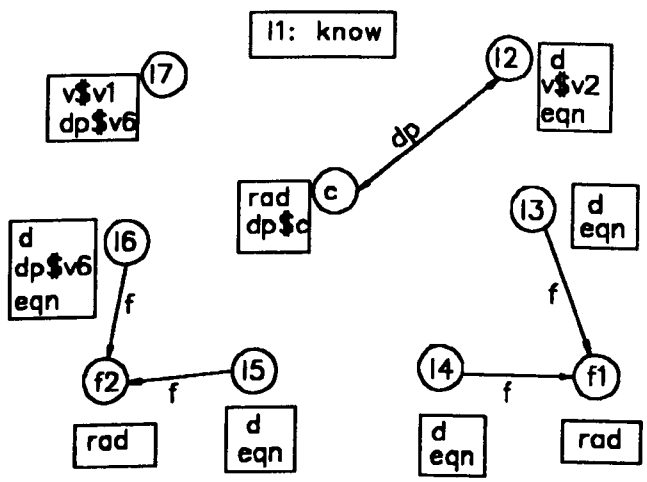


Figure 6.18 Intersections v3, v4 and v5 are duduced

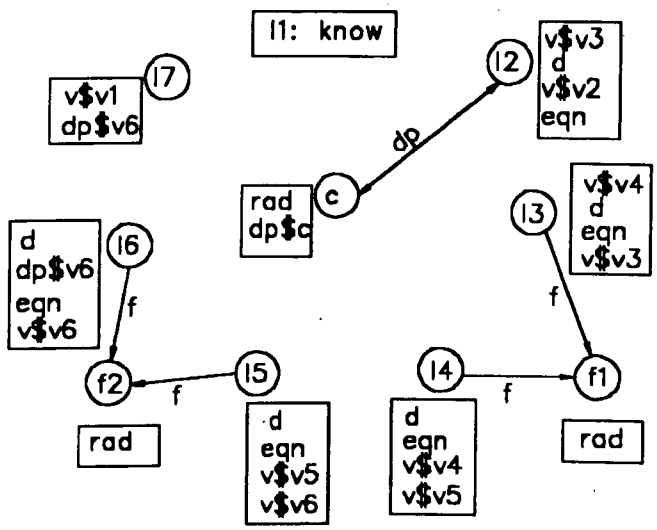


Figure 6.19 Lines 12, 13, 14 and 15 are known

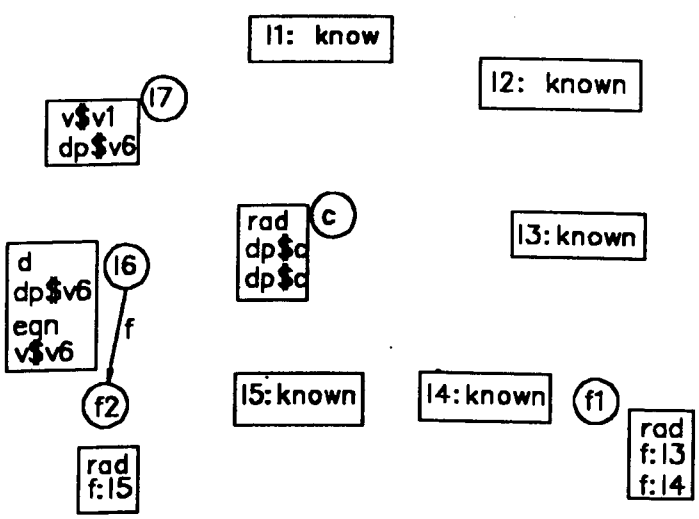


Figure 6.20 Fillet f1 is deduced

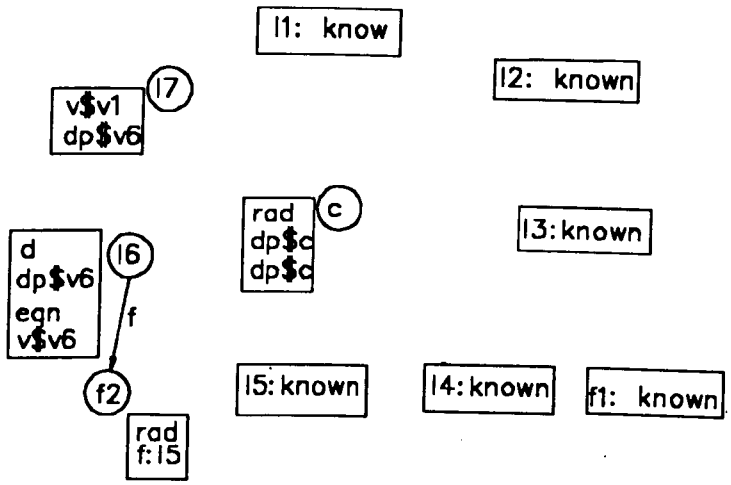


Figure 6.21 The circle c is deduced

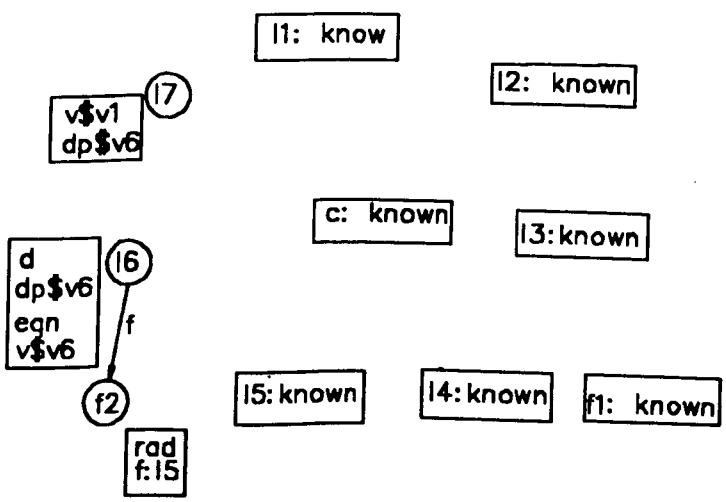


Figure 6.22 Vertex v7 is deduced

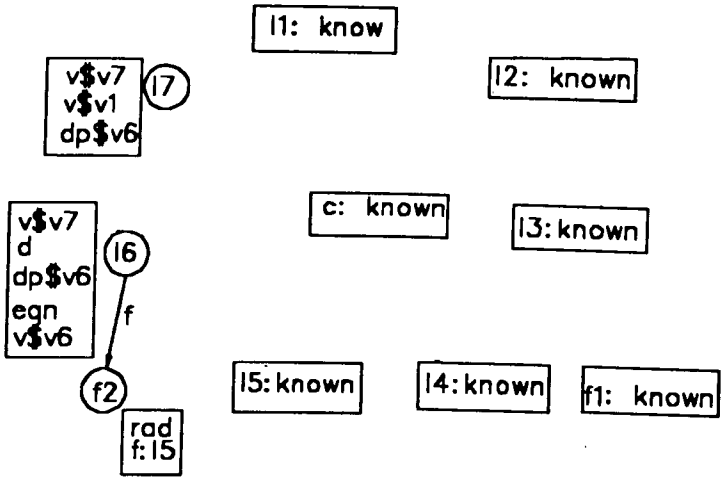
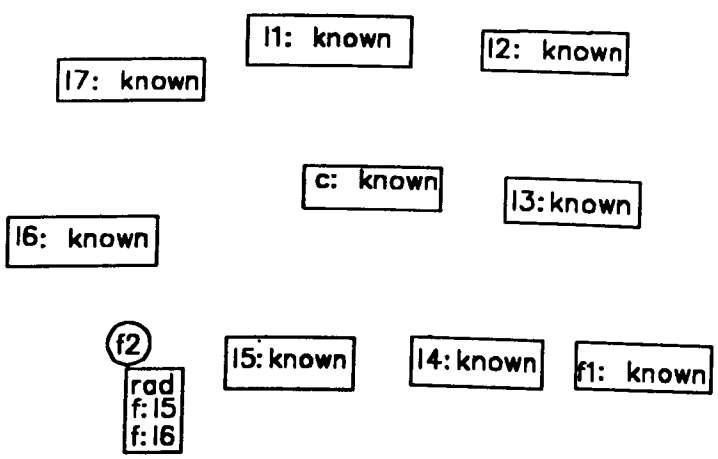


Figure 6.23 Lines l6 and l7 are known



6.5.3 Numerical Computations

In the proposed scheme implemented in LOGICAD, a rule does not explicitly incorporate algebraic knowledge for each one-step deduction. A rule only serves to propagate the information that a particular line element is derivable in the geometric reasoning process. For the actual computation, a sequence of numerical procedures are used, where each procedure corresponds to exactly one rule. These numerical procedures may only be executed after the propagation process has finished. The implementation of numerical procedures in Prolog has presented a problem since Prolog (including Wprolog) in general only provides simple arithmetic calculations like addition, substraction, multiplication and division. Although some more complicated calculations such as **sin(AN,VAL)**, **cos(AN,VAL)** and **power(B,P,VAL)** have been specially implemented in the LOGICAD system on top of Wprolog, the numerical procedures in Prolog are awkward and somewhat inefficient. For example, a simple procedure which may be attached to rule 6 is to find the intersections of two known circles. The Prolog program is

```
intersect(Xl1,Yl1,Rl1,Xk1,Yk1,Rk1,X1,Y1,X2,Y2):-
        Xl is float(Xl1),Yl is float(Yl1),
        Rl is float(Rl1),Rk is float(Rk1),
        Xk is float(Xk1),Yk is float(Yk1),
        RKSQ is Rk*Rk,  RLSQ is Rl*Rl,
        DELRSQ is (RLSQ)-(RKSQ),
        SUMRSQ is RKSQ + RLSQ,
        XLK is (Xl)-(Xk),YLK is (Yl)-(Yk),
        DISTSQ is (XLK)*(XLK) + (YLK)*(YLK),
```

```
DSTINV is '0.5'/(DISTSQ),
SCL is '0.5'-(DELRSQ)*(DSTINV),
X is (XLK)*(SCL) + Xk,
Y is (YLK)*(SCL) + Yk,
ROOT is '2.0'*SUMRSQ*DISTSQ
-DISTSQ*DISTSQ-(DELRSQ)*(DELRSQ),
power(ROOT,'0.5',ANS),
ROOT1 is (DSTINV)*(ANS),
XFAC is (XLK)*(ROOT1),
YFAC is (YLK)*(ROOT1),
X1 is fix(X)-fix(YFAC), Y1 is fix(Y) + fix(XFAC),
X2 is fix(X) + fix(YFAC), Y2 is fix(Y)-fix(XFAC).
```

Program 6.1 Calculate intersections of two circles

An experimental interface between Wprolog and Fortran has been implemented and investigated for LOGICAD. A USERDEF ASSEMBLE file must be used to describe all user subroutines in the IBM 3081 VM Wprolog system. Each subroutine is described using the $UDEF macro. The resulting USERDEF ASSEMBLE file for Fortran subroutines is:

```
'Type in the name(s) of your Fortran file(s)'
 Parse Upper Pull fortfiles
 'LIBRARY FORTVS GLIB '
 'SOFTLINK WPROLOG'
 'GLOBAL MACLIB UDEF'
 'ASSEMBLE USERDEF'
```

```
/* Compile each file in turn */
Do loop = 1 To WORDS(fortfiles)
'FORTVS' WORD(fortfiles,loop)
End /* Do loop */
'LOAD PROLOG USERDEF' fortfiles '(CLEAR ORIGIN 20000'
'GENMOD PROLOG'
Exit
```

Program 6.2 The Fortran/Prolog interface

Each time the subroutines are to be used, a macro command which is the name of the ASSEMBLE file must be issued. The connection between Prolog and Fortran is very slow since all Fortran subroutines must be compiled before they can be used within Wprolog. The connection may also affect the Prolog implementation because the numerical results in Fortran may cause integer and floating point number overflow in Wprolog.

6.6 Concluding Remarks

This chapter has described a novel geometric knowledge representation scheme which may suitably be called geometric element based reasoning (GEBR). The scheme allied with the dependency-directed backtracking search technique and the rule-cycle hybrid control structure has demonstrated an efficient way to solve geometric reasoning problems. Based on the GEBR scheme, geometric constraints are represented as a digraph. The constraints are incrementally resolved by a series of one-step deductions. The system keeps track of how every deduction is made. These records are used by

the system to trace down an inconsistent subset of a constraint set and to provide explanations of the derivation of a fact to the user.

The LOGICAD system has the known advantages of a rule-based system, such as the explicit and transparent representation of the geometric knowledge, the separation of the knowledge from its processing, and the capability to add new rules as needed without having to modify the inference component. In the LOGICAD system, constraints are satisfied incrementally and thus the popular but problem laden approach of transforming constraints into a set of nonlinear algebraic equations is avoided. It is suggested that the GEBR scheme as implemented in the LOGICAD system may be used as the basis for further applications such as mechanism kinematics and conceptual design.

CHAPTER 7

POSSIBLE EXTENSIONS TO LOGICAD

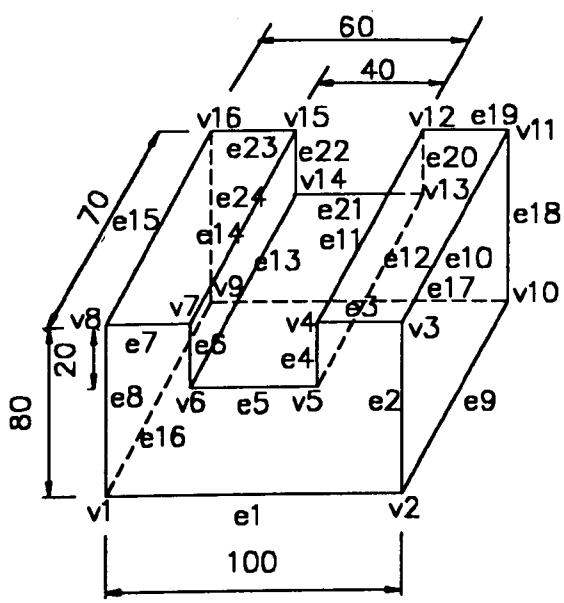
This thesis has presented a prototype ICAD system which we call LOGICAD. Although it is currently developed for 2D geometric inferencing modelling applications, it is argued in this chapter that the methods underlying the system may be consistently extended to many applications. This chapter suggests ways of applying the methods of previous chapters and presents consistent extensions to these methods to include applications such as 3D geometric inferencing, 2D mechanism kinematics, conceptual design, parametric design amongst others.

7.1 3D Applications

Three dimensional modelling of object geometry is a much more difficult task than the two dimensional problem. Existing CAD systems build object geometry by means of boundary descriptions or boolean operations. Few constraint based systems have touched the 3D application field. Bruderlin [Bruderlin 1986] proposes a method for 3D geometric construction through geometric reasoning. He uses a point set approach to represent object topology. In his system, a 3D CSG based modeller is used for the sketch of object topology, then one or two projections are generated from the solid model. Geometric constraints are added based on the projections. The author argues that this approach has inherited the drawbacks of wireframe representation such as ambiguous geometry. Working on projections may also

cause confusions for the user because it is sometimes difficult to understand the projection relationships. Another defect of this approach is that it is not a natural method for 3D modelling because geometric constraints for 3D object are mainly imposed on surfaces or edges. In this section, a possible extension of the GEBR (geometric elements based reasoning) method to 3D applications is outlined.

Figure 7.1 An object composed of orthogonal faces



Now since an object in 3D space can be described in terms of its binding surfaces, a major purpose of 3D geometric reasoning may be expected to be the deduction of the equations for all these surfaces, If all the equations of the

surfaces are known, then the geometry of the object can be calculated. This process is embedded in the following three steps :

1. Deduce the equations for all the surfaces.
2. Compute the intersections (edges) of all the surfaces.
3. Compute the intersections (vertices) of the edges.

After the above calculations, the precise geometry is known. The topological information of an object may be represented by its boundary edges. The internal representation of the object topology may be by means of a winged - edge data structure as described in Chapter 4. In the following discussion, object shapes are restricted to those that have orthogonal faces.

Example

Figure 7.1 shows an object with dimensions. The object is composed of ten surfaces. Each surface is bound by a number of edges. These can be seen from Figure 7.1 and Table 7.1. All the geometric constraints are listed as the following Prolog facts

dist(f3, f9, 100).

dist(f2, f8, 80).

dist(f2, f4, 80).

dist(f8, f6, 20).

dist(f1, f10, 70).

dist(f9, f5, 60).

dist(f5, f7, 40).

perp(f1, f2).

perp(f1, f9).

perp(f2, f9).

In order to construct the precise geometry of the object under a particular coordinate system for the 3D problem, instead of fixing a vertex and the orientation of an edge as in the 2D case, an edge and the orientation of a surface must be fixed for 3D problem. Suppose, in this example, edge **e1** and surface **f2** are fixed, that is

line(e16, known).

vert(v1, known).

vert(v9, known).

orient(f2, known).

Surfaces	Edge descriptions
f1	**[e1, e2, e3, e4, e5, e6, e7, e8]**
f2	**[e1, e9, e17, e16]**
f3	**[e9, e18, e10, e2]**
f4	**[e3, e10, e19, e11]**
f5	**[e4, e11, e20, e12]**
f6	**[e5, e12, e21, e13]**
f7	**[e6, e13, e22, e14]**
f8	**[e7, e14, e23, e15]**
f9	**[e8, e15, e24, e16]**
f10	**[e17, e18, e19, e20, e21, e22, e23, e24]**

Table 7.1 Surface information

Based on geometric reasoning rules, new information can be generated. For example, a suitable rule states that if the orientation of a surface is known and one of its boundary edges is also known, then the equation of the surface can be deduced. In Prolog, the rule can be represented as

```
eqn(FACE,deduceble):-
    orient(FACE,known),
    fi_edges(FACE,EDGES),
    line(E,known),
    member(E,EDGES).
```

A set of rules can be built into a rule base. The constraint propagation process for the example is illustrated in Table 7.2. The table shows the rules applied and the corresponding data generated. After all the equations of the faces have been found, it is easy to compute the equations for all the edges. For instance, the following program can be used

```
edge_eqn(EDGE, EQN):-
  ei_f(EDGE, [F1, F2]),
  intersect(F1, F2, EQN).
```

The predicate expression **intersect(F1, F2, EQN)** may be a subroutine (possibly in a procedural programming language) which finds the intersection of two planes. When all the edge equations are available, coordinates for all the vertices can be found by calculating the intersections of the edge equations.

Rule statements	Facts generated
eqn(FACE, deduceble) :-	**eqn(f2, known).**
orient(FACE, known),	--
fi_e(FACE, ELIST),	--
line(E, known),	--
member(E, ELIST).	--
eqn(FACE, deduceble) :-	**eqn(f8, known).**
dist(FACE, F, _),	**eqn(f4, known).**
eqn(F, known).	**eqn(f6, known).**
eqn(FACE, deduceble) :-	**eqn(f9, known).**
perp(FACE, F),	**eqn(f3, known).**
ei_f(E, [FACE, F]),	**eqn(f5, known).**
line(E, known),	**eqn(f7, known).**
eqn(F, known).	--
eqn(FACE, deduceble) :-	**eqn(f1, known).**
perp(FACE, F1),	**eqn(f10, known).**
perp(FACE, F2),	--
perp(F1, F2),	--
vi_f(V, [FACE, F1, F2]),	--
vert(V,known),	--
eqn(F1,known),	--
eqn(F2,known).	--

Table 7.2 The reasoning process for the 3D example

7.2 Mechanism Kinematics

Computer aided mechanism design is a numerical oriented task. It usually requires simultaneous solution of a set of nonlinear equations. Some mechanism analysis systems are actually subroutine packages [Molian 1984]. Subroutines may be called by application programs to perform specific tasks. This section describes the possible application of the GEBR method to mechanism analysis. The advantages of using this method would be

1. A declarative specification of mechanism variables.
2. A sequential equation solver would avoid the solution of simultaneous equations.

Example

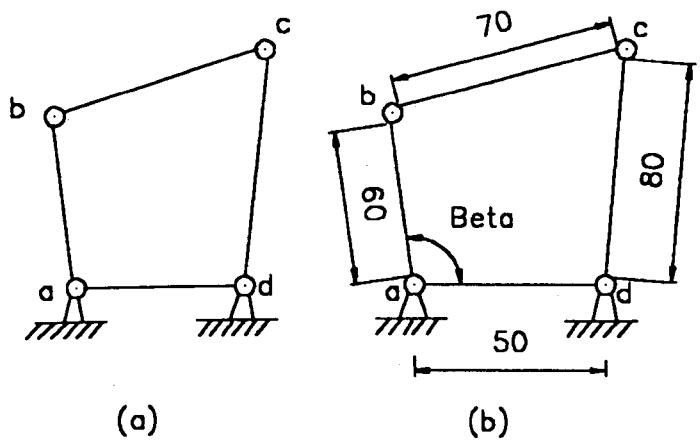
Figure 7.2(a) shows a sketch of the four-bar mechanism. After the declarative specification of the geometric constraints, the mechanism is as shown in Figure 7.2(b).

To study the mechanism, suppose points a and d are fixed. At this stage, the inference engine can be invoked. As a result of the inference process, a sequential construction path is generated which takes the following form:

line ad is deduced by rule
r(3,l,[v$V1,v$V2]).

line ab is deduced by rule
r(1,l,[vV,d,lenLEN]).

Figure 7.2 Four-bar mechanism example



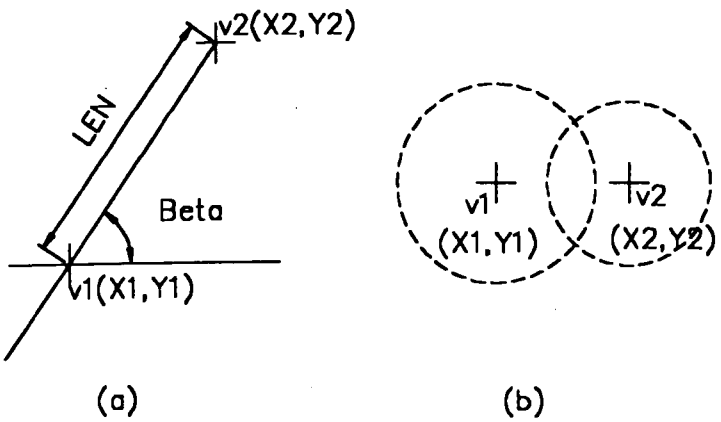
point c is deduced by rule

r(6,v,[[v$V1,len$L1],[v$V2,len$L2]]).

Each rule may be attached to a numerical procedure. For instance, rule 1 may be related with what is shown in Figure 7.3(a) which shows the calculation of **X2** and **Y2** based on the known conditions **X1, Y1, LEN** and **Beta** . Rule 6 may be related with Figure 7.3(b) which finds the intersections of two circles.

If, in Figure 7.2 (b), the angle Beta is the driving parameter, it can be input by an initial value and an incremental step value. Each time the step in-

Figure 7.3 Graphical illustration of numerical procedures



creases, the sequential procedures are executed and thus the new geometry is found.

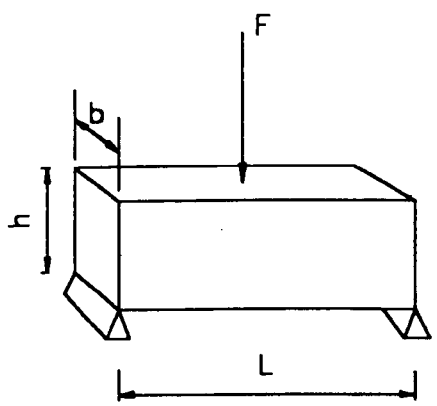
Sometimes multiple solutions may occur. For example, two circles will in general have two intersections. In this case, the intersection nearest to the previous vertex coordinates is used first, the other intersection is also stored just in case that alterative is required by the backtracking mechanism.

7.3 Conceptual Design

The GEBR method can be extended to support conceptual design by linking geometry to engineering equations [Shenton 1987]. In the design process, the

designer is often involved in establishing constraint equations that the designed object must satisfy. Once the constraint equations have been solved, the designer is able to see the geometric instances of the designed object. This can be realized by linking object geometry and geometric features to engineering equations.

Figure 7.4 Linking Geometry to Equation



For example, the rectangular beam in Figure 7.4 is carried on a simple support and subjected to a central load F. If the maximum allowable bending stress of the beam is σ_{max}, from simple bending theory, we could establish a relationship between the maximum allowable load F and the geometric parameters L, b and h by the equation

$$F = \frac{3\sigma_{max}L}{2bh^2}$$

The load F may further be related to other constraints. If F changes, it will affect the beam geometry. On the other hand, if the geometry changes, the maximum allowable load F may also change. Through this type of constraint connection, the computer can help the designer at a higher level of abstraction of the design process than conventional CAD systems.

7.4 Miscellaneous Applications

Apart from the applications discussed above, GEBR could also possibly serve as the basis for more applications such as parametric design, geometric feature extraction and geometric construction from scanned engineering drawings.

7.4.1 Geometric Construction from Scanned Engineering Drawings

The engineering industry has been migrating from manual to computer aided design and manufacturing. One of the serious obstacles which inhibits a more rapid migration process is the vast amount of manual technical documents which have accumulated during the years [Karima 1985]. Another obstacle is the tradition of communication by means of the engineering projection drawings.

The core of technical documentation is the engineering drawing which is produced by engineers to be read by engineers and requires geometric and

engineering knowledge to interpret properly. Nowadays, since so many drawings are still active and operational, there is demand for a fast and reliable means of transforming them into a digital form suitable for interactive graphics editing and subsequent reproduction and distribution. The common way of transformation is through digital scanning followed by image processing procedures to reproduce the drawings in the computer. However, the image is not to scale and thus needs an algorithm to construct the image to its precise geometry based on the scanned information. Since the LOGICAD system can be used to construct precise geometry based on a sketched object shape, the LOGICAD system is thus suitable for this application.

7.4.2 Geometric Feature Extraction

Geometric feature extraction is of significant importance for CAD/CAM integration [Henderson and Chang 1988]. A geometric feature is described by the relationships of its composing geometric elements. Since the LOGICAD system supports the manipulation of geometric constraints, we can thus define a geometric feature in terms of a set of constraints based on a certain topology. Pattern matching can be used in the feature extraction process.

7.4.3 Parametric Design

Current parametric design systems often provide some form of text macro facility. These systems suffer the drawbacks that the user needs to acquire the skills of a programmer in order to prescribe a complete solution to his problem. Although parametric interaction is favored by many users, the task of writing a macro program often discourages potential users. In the LOGICAD

approach, what the user is required to do is just to sketch the topology of the model he wants to create and then to specify a set of dimensions and relations. The system will generate the macro program automatically. It may also be possible for LOGICAD to parameterize only a subset of the dimensions and relations.

7.5 Concluding Remarks

This chapter has discussed the possible applications of the geometric knowledge representation scheme GEBR to 3D geometric inferencing, 2D mechanism kinematics and parametric design etc. There are some possible advantages in applying GEBR to these applications such as declarative semantics and computational efficiency. Based on the GEBR scheme, more tasks traditionally performed by users can be automated by the computer. For instance, current parametric design systems require the user to write a macro program to define a parametric model. This process can be automated based on GEBR through geometric inferencing.

CHAPTER 8

DISCUSSIONS AND CONCLUSIONS

Research in intelligent CAD systems has revealed the demand for more explicit representations and manipulation of geometric information. In this thesis, a novel geometric knowledge representation scheme of geometric element based reasoning has been proposed and developed. Based on this scheme, a rule-based 2D geometry system called LOGICAD is implemented. The geometric element based reasoning scheme is based on the observation that engineering drawings are dimensioned in such a way that all parameters of geometric elements can be computed (or graphically constructed) in a step by step sequence. Geometric rules are used to propagate the initial set of constraints to a sequence of constraint satisfaction steps.

The implementation scheme has the known advantages of a rule based approach, such as the explicit and transparent representation of knowledge, the separation of knowledge from the system control component, and the capability to add new rules as needed without having to modify the system control component. The LOGICAD system also has the merit of all declarative systems, that is, the sequence of constraints specifications need not be considered by the system. Using this system as implemented in the LOGICAD software for example, the designer can construct an idealised sketch in any order he wishes, and specify the geometric constraints on the sketch in any order. The system will then itself plan the sequence of constraint satisfactions. A further advantage of the method illustrated by LOGICAD is the separation of the symbolic from the numerical aspects. Symbolic computation is responsible for

the planning of the constraint satisfaction sequence, and the numerical computation is decomposed into the numerical satisfaction of each constraint based on the planned constraint satisfaction sequence. Thus the numerical computation is kept to a minimum and is restricted to straight forward geometric algorithms. All problems related to instability or accumulations of errors that sometimes afflict numerical methods are completely avoided.

LOGICAD provides a much easier method for parametric design than conventional CAD systems. A common approach using parametric design in current CAD systems is to provide some form of text macro facility. These systems suffer the drawback that users need to acquire the skills of a programmer in order to become an effective macro writer. Once learnt, the text macro continues to present significant obstacles to the user. For instance, in order to iterate a design, a text editor must be provided which may be completely separated from the CAD system. In LOGICAD, the macro is generated by the system through geometric reasoning. The user interface of LOGICAD may also provide a facility for constraint modifications.

An approach for possible extension of the LOGICAD scheme to conceptual design is proposed. This approach makes use of the connecting equations among the geometric constraints and the functional constraints. Thus the system may assist the designer at a higher level of abstraction in the design process than is possible with conventional CAD systems.

For many constraint programming systems, it is difficult to maintain a consistent set of constraints. In the case of geometric modelling, the problem of constraint maintenance is even more difficult because some implicit and

structural constraints are difficult to recognize. Furthermore, there is not one common approach which can detect that a constraint scheme for a given object shape is under -, or over - constrained or consistent. The famous Euler formula provides limited information for the maintenence of consistent geometric constraints. In the LOGICAD software, a novel method is used in which geometric constraints are checked at data entry. By so doing, the system can guarantee that no redundant constraints are entered.

A major limitation of the proposed scheme is the difficulty of solving constraints that involve a high degree of simultaneity such as the **area** for some surface. These type of constraints (global constraints) often simultaneously relate all the geometric elements of an object. Fortunately, global constraints are usually not used for the description of object shape. If this probem should occur, the rule based approach can not plan a constraint satisfaction sequence. A possible approach to overcome this problem is through the introduction of surrogate constraints which can be progressively relaxed until the global constraints are satisfied. The approach may involve the following two steps:

1. The global constraints are hidden. Choose some surrogate constraints (independent of the given constraint set) which combine with the rest of the given constraints to make a consistent constraint scheme.
2. Generate the construction path. The construction path is then iterated while relaxing the surrogate constraints until the global constraints are satisfied.

Intelligent CAD research requires the facility of more explicit representation and manipulation of geometric knowledge. In this thesis, the proposed scheme is shown by its implementation in the LOGICAD software to provide such a facility.

REFERENCE

1. Akiner, V. T., 1986, "Topology-1: A Knowledge-Based System for Reasoning about Objects and Space", Design Studies, Vol. 7, No. 2, April 1986, pp94-105

2. Aldefeld, B., 1986, "Rule-Based Approach to Variational Geometry", Knowledge Engineering and Computer Modeling in CAD (Proc. CAD86), Butterworth, 1986, pp59-67

3. Aldefeld, B., 1988, "Variation of Geometry Based on a Geometric-Reasoning Method", Computer-Aided Design, Vol. 20, No. 3, April 1988, pp117-126

4. Ansaldi, S. and Falcidieno, B., 1988, "The Problem of Form Feature Classification and Recognitions in CAD/CAM", Theoretical Foundation of Computer Graphics and CAD, R. A. Earnshaw (ed), Springer-Verlag 1988, pp899-917

5. Arbab, F., 1987, "Example of Geometric Reasoning in OAR", in Intelligent CAD Systems II, V. Akman P. J. W. ten Hagen P. J. Veerkamp (Eds.), Springer-Verlag 1987, pp32-58

6. Arbab, F. and Wang, B., 1989, "A Geometric Constraints Management System in Oar" Proc. Third Eurographic Workshop on Intelligent CAD Systems, April 1989 Holland, pp231-252

7. Baumgart, B., 1975, "A Polyhedron Representation for Computer Vision", Proc. AFIPS, Vol. 44, 1975, pp598-596

8. Bijl, A., 1987, "Strategies for CAD", in Intelligent CAD Systems I, P. J. W. ten Hagen T. Tomiyama (Eds.), Springer-Verlag, 1987, pp2-20

9. Bond, A and Soeterman, B., 1987, "Integrating Prolog and CADAM to Produce an Intelligent CAD system", June 1987, Proc. Western Conference on Expert System, System, Anaheim, CA., U. S. A.

10. Borning, A., 1981, "The Programming Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory", ACM Transaction on Programming Language, Vol. 3, No. 4, October 1981, pp353-387

11. Bratko, I., 1986, "Prolog Programming for Artificial Intelligence", Addison-Wesley Publishing Company, Inc., 1986

12. Bruderlin, B., 1985, "Using Prolog for Constructing Geometric Objects Defined by Constraints", Springer LNCS, No. 204, Springer-Verlag, 1985, pp448-459

13. Bruderlin, B., 1986, "Constructing Three-Dimensional Geometric Objects Defined by Constraints", workshop on Interactive 3D Graphics, October, 1986, pp111-129

14. Brumsen, H. P and Treur, T., 1990, "Modelling Dynamic Aspects of Design Processes", Proc. 4th Eurographics Workshop on Intelligent CAD Systems, April 1990, France

15. Bundy, A and Welham, B., 1981, "Using Meta-level Interface for Selective Application of Multiple Rewrite Rules in Algebraic Manipulation", Artificial Intelligence 16:2, 1981

16. Cahn, W. T. and Paulson, B. C., 1987, "Exploratory Design Using Constraints", AI EDAM, Vol.1, No.1, Academic Press Ltd. 1987, pp59-71

17. CIME, 1989, "CAD Follows New Script", Mechanical Engineering, November 1989, pp62-67

18. Clocksin, W. F and Mellish, C. S., 1981, "Programming in Prolog", Springer-verlag, New York, 1981

19. Colmerauer, A., 1978, "Metamorphsis Grammars", in Natural Language Communication with Computers (L, Bolc, ed.) Springer-Verlag, 1978, pp133-189

20. Cugini, U., Devoti, C and Calli, P., 1985, "System for Parametric Definition of Engineering Drawing", Proc. MICAD'85, 1985, pp50-63

21. Darvas, F., 1980, "Logic Programming in Chemical Information Handling and Drug Design", Proc. Logic Programming Workshop, July, 1980

22. David, B. T., 1987, "Multi-Expert System for CAD", in Intelligent CAD Systems I P. J. W. ten Hagen T. Tomiyama (Eds.), Springer-Verlag, 1987, pp57-68

23. Doyle, J., 1979, " A Truth Maintenance System", Artificial Intelligence, Vol. 12, No. 3, 1979, pp231-272

24. Duhovmik, J., 1987, "Systematic Design in Intelligent CAD Systems", in Intelligent CAD Systems I, P.J.W. ten Hagen T. Tomiyama(Eds.), Springer-Verlag, 1987, pp224-239

25. Foundyller, C. M and Jenkin, B. L. eds. "CAD-CAM,CAE: Survey Review and Buyer's guide", Daratech, Cambridge, 1986

26. Franklin, Wm. R., Wu, P. Y. F etc., 1986, "Prolog and Geometry Projects", IEEE CG&A, November 1986, pp46-55

27. George, J. 1988, "Capture Knowledge Creates Better Design" Eureka Transfer Technologgy, September 1988, pp141-142

28. Gibbins, P., 1988, "Logic with Prolog", Oxford University Press, 1988

29. Gloess, P. Y and Guerin, J. L., 1990, "A Graphic Editor for an Object Oriented Logic", Proc. Fourth Eurographics Workshop on Intelligent CAD Systems, April 1990, France

30. Gonzalez, J. C., William, M. H and Aitchison, I. E., 1984, "Evaluation of the Effectiveness of Prolog for a CAD Application", IEEE CG&A, March 1984, pp67-75

31. Gossard, D. C., Light, R. A and Lin, V., 1980, "A Computer-Aided Design System for Design by Variation", Proc. Fourth International Conference on Production Engineering, Tokyo, August 1980

32. Gossard, D. C and Lin, V., 1982, "Representation of Part Family Through Variational Geometry", PROLAMAT 82, Leningrad, May 1982

33. Groover, M. P. and Zimmer, E. M.,1984, "CAD/CAM: Computer-Aided Design and Manufacturing", Prentice-Hall, Inc., Englewood Clifs, New Jersey, 1984

34. Hampshire, N, 1988, "The Intelligent Amendment", Workstation Magazine, November 1988, pp27-29

35. Haugeland, J., 1985, "Artificial Intelligence: The Very Idea", The MIT Press, 1985

36. Henderson, M. R and Chang, G. J., 1988, "FRAPP: Automated Feature Recognition and Process Planning From Solid Model Data", Proc. ASME Computers in Engineering Conference, San Francisco, August 1988, pp529-536

37. Hillyard, R. C., 1978, "Dimensions and Torerances in Shape Design", PhD Thesis, University of Cambrdge, UK, June 1978

38. Hillyard, R. C and Braid, I. C., 1978a, "Analysis of Dimensions and Tolerances in Computer-Aided Mechanical Design", Computer-Aided Design, Vol. 10, June 1978, pp161-166

39. Hillyard, R. C and Braid, I. C., 1978b, "Characterizing Non-ideal Shapes in Terms of Dimensions and Tolerances", Computer Graphics, ACM-Siggraph, Vol. 12, No. 3, August 1978, pp234-238

40. Hubner, W and Markov, Z. I., 1986, "GKS Based Graphical Programming in Prolog", Computer Graphics Forum 5, 1986, pp41-50

41. Intralogic Inc., 1985, "Waterloo Core Prolog User's Manual", Version 1.6, Intralogic Inc., Waterloo, Ontario, Canada, 1985

42. ISO, 1985, "Information Processing Systems - Computer Graphics - Graphical Kernel System (GKS) - Functional Descriptions", ISO 7942, ISO Secretariat, Geneva, 1985

43. Jayaram, S and Myklebust, A., 1990, "Automatic Generation of Geometry Interfaces and CADCAM Systems", Computer-Aided Design, Vol.22, No. 1, 1990, pp50-56

44. Karima, M etc., 1985, "From Paper Drawings to Computer-Aided Design", IEEE CG&A, February 1985, pp27-38

45. Kawagoe, K and Managaki, M., 1984, "Parametric Object Model: A Geometric Data Model for Computer-Aided-Engineering", NEC Research&Development, Nippon Electric Co., Tokyo, 1984, Vol. 72, pp23-32

46. Koegel, J. F., 1987, "A Theoretical Model for Intelligent CAD", in Intelligent CAD System I, P. J.W. ten Hagen T. Tomiyama (Eds.), Springer-Verlag, 1987, pp206-223

47. Kowalski, R., 1979, "Algorithm = Logic + Control", Communication of ACM, July, 1979, pp424-436

48. Krishnamurti, R., Sykes, R and Kemp, R., 1987, "The Prolog/GKS Binding", EdCAAD, University of Edinburgh, 1987

49. Krishnan, D and Patnaik, K. M., 1986, "GEODERM: Geometric Shape Design System Using an Entity-Relationship Model", Computer-Aided Design, Vol. 18, No. 4 May 1986, pp207-218

50. Kriwaczek. F., 1982, "Some Applications of Prolog to Decision Support Systems", MSc Thesis, Imperial College, London, 1982

51. Leler, Wm., 1988, "Constraint Programming: Their Specification and Generation", Addison-Wesly Publish Company, Inc., 1988

52. Li, D., 1984, "A Prolog Database System", Research Studies Press, England, 1984

53. Light, R. A and Gossard, D. C., 1982, "Modification of Geometric Models Through Variational Geometry", Computer-Aided Design, Vol. 14, No. 4, July 1982 pp209-214

54. Light, R. A and Gossard, D. C., 1983, "Variational Geometry: A New Method for Modifying Part Geometry for Finite Element Analysis", Computer and Structures, Vol. 17, No. 5-6, June 1983, pp903-909

55. Lin, V. C., Gossard, D. C and Light, R. A., 1981, "Variational Geometry in Computer Aided Design", Computer Graphics, Vol. 15, No. 3, August 1981, pp171-179

56. Llewelyn, A. L., 1989, "Review of CAD/CAM", Computer-Aided Design, Vol. 21, No. 5, June 1989, pp297-302

57. Maarten, J .G. M., "A System for Graphical Interaction on Parameterized Models".

58. Mantyla, M. 1988, "An Introduction to Solid Modeling", Computer Science Press, 1988

59. McCord, M. C., 1985, "Modular Logic Grammars", Proc. 23rd Annual Meeting of the Association for Computational Linguistics, Chicago, July, 1985, pp104-117

60. Milanese, V., 1988, "A Prolog Environment for GKS-Based Graphics", Computer Graphics Forum 7, 1988, pp9-20

61. Molian, S., 1984, "Software for Mechanism Design", Proc. Mechanism Conference, U.K, 1984

62. Mostow, J., 1985, "Toward Better Models of the Design Process", AI Magazine, Vol. 6, No. 1, 1985, pp44-57

63. Moto-Oka, T etc., 1982, "Challenge for Knowledge Information Processing Systems", in Fifth Generation Computer Systems, North-Holland, Amsterdam, 1982, pp3-89

64. Myers, L and Pohl, J., 1990, "ICAD DEMO1", Proc. Fourth Eurographics Workshop on Intelligent CAD Systems, France, April 1990

65. Nelson, G., 1985, "Juno: A Constraint-Based Graphics System", SIGGRAPH Computer Graphics, Vol. 19, No. 3, July 1985, pp235-243

66. Nii, P., 1984, " Choice Without Backtracking", Proc. AAAI-84, 1984, pp79-85

67. Nii, P., 1986, "Blackboard System Part One-the Blackboard Model of Problem Solving and the Evolution Blackboard Architecture", AI Magazine, Vol.7, No.2, 1986, pp38-53

68. Nii, P., 1986, "Blackboard System Part Two-Blackboard Application Systems, Blackboard Systems from Knowledge Engineering Perspective", AI Magazine, Vol. 8, No. 3, 1986, pp82-106

69. Pahl. G, Beitz. W., 1988, "Engineering Design", The Design Council, London, 1988

70. Parden, G and Newell, R. G., 1984, "A Dimension Based Parametric Design System", Proc. CAD84, Brighton UK, 1984, pp252-259

71. Parnas, D. L., 1972, "On the Criteria to Be Used in Decomposing Systems into Modules", Communications of the ACM 15 (12), December 1972, pp1053-1058

72. Pineda, L and Klein. E., 1989, "On the Integration of Graphical and Linguistic Knowledge for CAD Systems", Proc. Third Eurographics Workshop on Intelligent CAD Systems, April 1989, Holland

73. Popplestone, R. J., 1986 "The Edinburgh Designer System as a Framework for Robotics: The Design of Behavior", AI for Engineering Design Analysis and Manufacture, Vol. 1, No. 1, pp25-36

74. Requicha, A. A. G and Voelcker, H. B., 1988, "Solid Modeling: A Historical Sumary and Contemporary Assessment", IEEE CG&A, March 1982, pp9-22

75. Rogers, D. F., 1985, "Procedural Elements for Computer Graphics", McGraw-Hill Book Company, 1985

76. Ruttkay, Z., Allen, R. H and Laczik, B., 1987, "A Multiparadigm User Interface for Intelligent CAD Systems", in Intelligent CAD System I, P. J.W. ten Hagen T. Tomiyama (Eds.), Springer-Verlag, 1987, pp242-255

77. Schuster, R., Voge, E and Tripprer, D., 1986, "The Use of Computers in Design and Planning-Integration via Interface Management", Comput. & Graphics, Vol. 10, No. 4, 1986, pp277-295

78. Serrano, D and Gossard, D., 1988, "Constraint Management Management in MCAE", Artificial Intelligence in Engineering Design, Gero Ed. J. S., 1988, pp217-240

79. Shah, J. J and Rogers, M. T, 1988, "Expert Form Features Modelling Shell", Computer-Aided Design, Vol. 20, No. 9, November 1988, pp515-524

80. Shenton, A. T., 1987, "Computer Integration of Geometric Modelling and Engineering Design Systems by Rule Based Programming", 3rd Inter. Conference on Effective CAD/CAM, November 1987, London, pp57-68

81. Shenton, A. T, Taleb-Bendiab, A and Chen, Y., 1989, "Computer-Aided Constraint Development Systems for Conceptual and Embodiment Engineering Design", Proc. 3rd Eurographic Workshop on Intelligent CAD Systems, April 1989, Holland

82. Shenton, A. T and Chen, Y., 1990, "A Logic Programming System for Design Processes: Clausal Form Clause Resolution by Best-First And/Or Inference through Partial Unitfication", Proc. Fourth Eurographics Workshop on Intelligent CAD Systems, Mortefontaine, France, April 1990

83. Shigley, J. E., 1977, "Mechanical Engineering Design", 3rd Edition, McGraw-Hill Book Company, New York, 1977

84. Smithers, T., 1989, "AI-Based Design versus Geometry-Based Design or Why Design Cannot Be supported by Geometry alone", Computer-Aided Design, Vol. 21, No. 3, April 1989, pp141-150

85. Smith, H. F., 1987, "Data Structures: Form and Function", Harcourt Brace Jovanovich, Inc., 1987

86. Stallman, R. M and Sussman, G. J., 1977, "Forward reasoning and Dependency-directed backtracking in a system for Computer-Aided Circuit Analysis", Artificial Intelligence 9, 1977, pp135-196

87. Sunde, G., 1986, "Specification of Shape Dimension and Other Geometric Constraints", Proc. IFIP WG5. 2 Workshop on Geometric Modeling, New York May 1986, pp199-213

88. Sunde, G., 1987, "A CAD System with Declarative Specification of Shape", Proc. First Eurographic Workshop on Intelligent CAD Systems, April 1987, Noordwijkerhout, The Netherland

89. Sutherland, I. E., 1963, "Sketchpad: A Man-Machine Graphical Communication System", Proc. AFIPS Spring Joint Computer Conference, Detroit, U. S. A., 1963, pp329-346

90. Sutherland, I. E and Hodgman, G. W., 1974, "Reentrant Polygon Clipping", CACM, Vol. 17, pp32-42, 1974

91. Swinson, P. S., 1980, "Prescriptive to Descriptive Programming: A Way Ahead for CAAD", Proc. Logic Programming Workshop, July 1980

92. Tektronix Inc., 1986, "TEK Programmer's Reference", Tektronix Inc., March 1986

93. Tomiyama, T., 1987, "Object Oriented Programming Paradigm for Intelligent CAD Systems", in Intelligent CAD Systems II, V. Akman P. J. W. ten Hagen P.J. Veerkamp (Eds.), Springer-Verlag, 1987, pp3-17

94. Tomiyama, T and Yoshikawa, H., 1987, "Extended General Design Theory", Proc. IFIP WG5.2 Working Conference on Design Theory for CAD, 1987, pp95-124

95. Trevett, N. and Davis, B., 1987, "The Software Explosion", CADCAM International, January 1987, pp23-24

96. Veerkamp, P and Akman, V etc., 1987, "IDDL: A Language for Intelligent Interactive Integrated CAD Systems", in Intelligent CAD Sytems II, V. Akman P. J. W. ten Hagen P. J. Veerkamp (Eds.), Springer-Verlag, 1987, pp58-77

97. Veerkamp, P and Kiriyama, T. etc., 1990, "Representation and Implementation of Design Knowledge for Intelligent CAD", Proc. Fourth Eurographic Workshop on Intelligent CAD Systems, April 1990, France

98. Veth, B., 1987, "An Integrated Data Description Language for Coding Design Knowledge", in Intelligent CAD Systems I, P. J. W. ten Hagen T. Tomiyama (Eds.), Springer-Verlag, 1987 pp295-313

99. Weiler, K and Atherton, P., 1977, "Hidden Surface Removal Using Polygon Area Sorting", Computer Graphics (Proc. SIGGRAPH 77), Vol.11, pp214-222,

100. Wirth, N., 1976, "Algorithms + Data Structures = Programs", Prentice-Hall, 1976

101. Woo, T. C., 1985, "A Combinatorial Analysis of Boundary Data Structure Schemata", IEEE CG&A, Vol. 5, No. 3, March 1985, pp19-28

102. Worden, R. P., 1984, "Context in Knowledge Bases", Proc. 2nd Alvey Working Group on Large Knowledge Based Systems, November 1984

103. Zaumen, W. T., 1983, "Computer-Assisted Circuit Evaluation in Prolog for VLSI", Proc. ACM Database Week, San Jose, CA., 1983

A1

TEKTRONIX TERMINAL/PROLOG COMMUNICATION

```
/*************************************************************/
/*- This programme shows how to find the corresponding code ---*/
/*- between ASCII and EBCDIC based on which the integer bit ---*/
/*- and coordinates input and output are programmed ------------*/
/*************************************************************/
getlist(0,[HEAD|TAIL],HEAD).
getlist(NX,[HEAD|TAIL], OUT):-
        N is NX - 1,
        getlist(N,TAIL,OUT).
/*--------- change code from ASCII to EBCIDIC ---------------*/
asciebci(ASCI,EBCI):-
        ASCI =< 255, listx(LISTX),
        getlist( ASCI,LISTX,EBCI).
listx(
  [00,01,02,03,55,45,46,47,22,05 37,11,12,13,14,15,
  16,17,18,19,60,61,50,38,24,25,63,39,28,29,30,31,
  64,90,127,123,91,108,80,125,77,93,92,78,93,92,78,107,96,75,97,
  240,241,242,243,244,245,246,247,248,249,122,94,76,126,110,111,
  ]124,193,194,195,196,197,198,199,200,201,209,210,211,212,213,214,
  215,216,217,226,227,228,229,230,231,232,233,074,224,079,095,109,
  121,129,130,131,132,133,134,135,136,137,145,146,147,148,149,150,
  151,152,153,162,163,164,165,166,167,168,169,192,106,208,161,007]).
 /*---------- change code from EBCIDIC to ASCI ------------*/
  ebciasci(EBCI,ASCI):- getlist(EBCI,
  [000,001,002,003,255,009,255,127,255,255,255,011,012,013,014,015,
  016,017,018,019,255,010,008,255,024,025,255,255,028,029,030,031,
  255,255,255,255,255,010,023,027,255,255,255,255,255,005,006,007,
  255,255,022,255,255,255,255,004,255,255,255,255,020,021,255,026,
```

```
023,255,255,255,255,255,255,255,255,255,091,046,060,040,043,093,
038,255,255,255,255,255,255,255,255,255,033,036,042,041,059,094,
255,255,255,255,255,255,255,255,255,255,124,044,037,095,062,063,
255,255,255,255,255,255,255,255,255,096,058,035,064,039,061,034,
255,097,098,099,100,101,102,103,104,105,255,255,255,255,255,255,
255,106,107,108,109,110,111,112,113,114,255,255,255,255,255,255,
255,126,115,116,117,118,119,120,121,122,255,255,255,255,255,255,
255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,
123,065,066,067,068,069,070,071,072,073,255,255,255,255,255,255,
125,074,075,076,077,078,079,080,081,082,255,255,255,255,255,255,
092,255,083,084,085,086,087,088,089,090,255,255,255,255,255,255,
048,049,050,051,052,053,054,055,056,057,255,255,255,255,255,255]
,ASCI).
/*------- for reading echo of characters from terminal --------*/
ibmp(DATA):- get0(EBCI),
             asciebci(ASCI,EBCI),
             DATA is  ASCI - 32.
ibmasci(CHAR):- get(EBCI),
                name(CHAR,[EBCI]).
/*-------- output ASCI characters to terminal ----------------*/
ibmput(ASCI):- asciebci(ASCI,EBCI),put(EBCI).
/*-------- output X Y cordinates to terminal -----------------*/
ibmpack(XI,YI):-
             HIY is YI/128 ,
             LOY is (YI - HIY*128)/4 ,
             EXY is YI - HIY*128 - LOY*4 ,
             HIX is XI/128 ,
             LOX is (XI - HIX*128)/4 ,
             EXX is XI - HIX*128 - LOX*4 ,
             OUTHIY is 32 + HIY ,
             OUTEXT is 96 + EXY*4 + EXX ,
             OUTLOY is 96 + LOY ,
             OUTHIX is 32 + HIX ,
             OUTLOX is 64 + LOX ,
             ibmput(OUTHIY) ,
```

```
            ibmput(OUTEXT) ,
            ibmput(OUTLOY) ,
            ibmput(OUTHIX) ,
            ibmput(OUTLOX) .
/*---------- rcecieve integer report from terminal ---------*/
 in_integer(SIGN,INTEGER):-
        ibmp(HI1),
        ibmp(HI2),ibmp(LO),
        HI_I1 is HI1*1024, HI_I2 is HI2*16,
        LO_1 is LO mod 16,
        INTEGER is HI_I1 + HI_I2 + LO_I,
        SIGN is (LO/16) mod 2.
/*---------- decide the sign of the integer ----------------*/
 out_integer(INTEGER):-
        in_integer(SIGN,INTE),
       ((SIGN = 0, INTEGER is (-1)*INTE),!;
       INTEGER = INTE).
/*---------- receive X Y report from the terminal ----------*/
 xy_in(KEY,HIY,EX,LOY,HIX,LOX):-
        ibmasci(KEY),
        ibmp(HIY),
        ibmp(EX),
        ibmp(LOY),
        ibmp(HIX),
        ibmp(LOX).
/*---------- calculated the X Y coordinates value ----------*/
 xy_out(KEY,X,Y):-
        xy_in(KEY,HIY,EX,LOY,HIX,LOX),
        HIX1 is HIX*128,
        LOX1 is LOX*4,
        EXx is EX mod 4,
        X is HIX1 + LOX1 + EXx,
        HIY1 is HIY*128,
        LOY1 is LOY*4,
        EXy1 is EX/4,
```

```
        EXy is EXy1 mod 4,
        Y is HIY1 + LOY1 + EXy.
/*---------- output integer to the terminal ----------*/
integer_bit(N):-    /*---- when integer is positive ------*/
        N > =0,
        Q is N/16,
        Q1 is N-Q*16,
        Q2 is 48 + Q1,
        ((Q < 1,ibmput(Q2));encod2(Q,Q2)).
integer_bit(N):-    *---- when integer is negative ------*/
        N < 0,P is abs(N),
        Q is P/16,
        Q1 is P-Q*16,
        Q2 is 32 + Q1,
        ((Q < 1,ibmput(Q2));encod2(Q,Q2)).
encod2(N,LO):-
        Q is N/64,
        Q1 is N-Q*64,
        Q2 is Q1 + 64,
        ((Q < 1, ibmput(Q2), ibmput(LO));
         encod3(Q,LO,Q2)).
encod3(N,LO,HI1):-
        Q is N/64,
        Q1 is N-Q*64,
        Q2 is Q1 + 64,
        ibmput(Q2), ibmput(HI1), ibmput(LO), !.
/*--------- output real number to terminal ----------*/
real_bit(N):-
        integer_bit(N),
        integer_bit(0).
real_bit(I_P):-
        integer_bit(I),
        (P = 125,integer_bit('-1'));
        (P = 5,integer_bit('-2'));
        (P = 25,integer_bit('-3')).
```

A2

EXTENDED NUMERICAL PREDICATES

```
/*-----------SIN COS POWER CALCULATIONS-------------------*/
/*---sin(DEG,ANSWER)--cos(DEG,ANSWER)----------------------*/
/*---power(NUM,POWER,ANSWER)--exp(NUM,ANSWER)-----*/
/*---log_10(NUM,ANS)--antilog(NUM,ANS)--------------------------*/
/*---plus(A,B,C)-->A + B = C--product(A,B,C)-->A * B = C--*/
deg_rad(DEG,RAD) :-  /*--float(DEG,DEGF)--*/
        product(DEG,'0.0174532',RAD).
rad_deg(RAD,DEG):-   /*--float(RAD,RADF)--*/
        deg_rad(DEG,RAD).
cos(DEG,COS):-  DEG < 0,
        plus(DEG,360,NEW_DEG), !, cos(NEW_DEG,COS).
cos(DEG,COS):-  DEG > 360,
        plus(NEW_DEG,360,DEG), !, cos(NEW_DEG,COS).
cos(DEG,COS):-  DEG > 180, plus(NEW_DEG,180,DEG), !,
        cos(NEW_DEG,ANS), plus(ANS,COS,0).
cos(DEG,COS):-  deg_rad(DEG,RAD),
        product(RAD,RAD,X_FACT),
        plus(X_FACT,M_X_FACT,0),
        sum_series(M_X_FACT,1,1,0,10,COS),!.
sin(DEG,SIN):-  DEG < 0,
        plus(DEG,360,NEW_DEG), !, sin(NEW_DEG,SIN).
sin(DEG,SIN):-  DEG > 360,
        plus(NEW_DEG,360,DEG), !, sin(NEW_DEG,SIN).
sin(DEG,SIN):-  DEG > 180, plus(NEW_DEG,180,DEG), !,
        sin(NEW_DEG,ANS), plus(ANS,SIN,0).
sin(DEG,SIN):-  deg_rad(DEG,RAD),
        product(RAD,RAD,X_FACT),
        plus(X_FACT,M_X_FACT,0),
        sum_series(M_X_FACT,RAD,RAD,1,11,SIN),!.
sum_series(M_X_FACT,OLD_TERM,SUM,LIMIT,LIMIT,SUM).
sum_series(M_X_FACT,OLD_TERM,GRO_SUM,
```

```
        OLD_POWER,LIMIT,SUM):-
        plus(2,OLD_POWER,NEW_POWER),
        plus(1,OLD_POWER,NPMINUS_1),
        product(OLD_TERM,M_X_FACT,TERM1),
        product(TERM2,NEW_POWER,TERM1),
        product(NEW_TERM,NPMINUS_1,TERM2),
        plus(GRO_SUM,NEW_TERM,NEW_GRO_SUM),!,
        sum_series(M_X_FACT,NEW_TERM,NEW_GRO_SUM,
        NEW_POWER,LIMIT,SUM).
plus(X,Y,Z):-    nonvar(X), nonvar(Y),Z is X + Y.
plus(X,Y,Z):-    nonvar(X), nonvar(Z),Y is Z - X.
plus(X,Y,Z):-    nonvar(Y), nonvar(Z),X is Z - Y.
product(X,Y,Z):- nonvar(X), nonvar(Y),Z is X * Y.
product(X,Y,Z):- nonvar(X), nonvar(Z),Y is Z / X.
product(X,Y,Z):- nonvar(Y), nonvar(Z),X is Z / Y.
rproduct((XN,XD),(YN,YD),(ZN,ZD)):-
        rnonvar((XN,XD)), rnonvar((YN,YD)),
        product(XN,YN,ZN), product(XD,YD,ZD).
rproduct((XN,XD),(YN,YD),(ZN,ZD)):-
        rnonvar((XN,XD)), rnonvar((ZN,ZD)),
        product(XD,ZN,YN), product(ZD,XN,YD).
/*-----rnonvar((XN,XD)), rnonvar((ZN,ZD))-----*/
/*-----YN is XD * ZN, YD is ZD * XN.----------*/
rproduct((XN,XD),(YN,YD),(ZN,ZD)):-
        rnonvar((YN,YD)), rnonvar((ZN,ZD)),
        product(YD,ZN,XN), product(ZD,YN,XD).
rnonvar((N,D)):-   nonvar(N), nonvar(D).
float(IN,OUT):-    float_pos(IN,OUT);
        float_neg(IN,OUT).
float_pos(IN,OUT):- IN > = 0, OUT is float(IN).
float_neg(IN,OUT):- IN <  0, XI is '-1.0' * IN,
        XF is float(XI), OUT is '-1.0' * XF.
log_scale(NUM,EXP,FACT):-
        log_scale(0,NUM,EXP,FACT).
log_scale(EXP,FACT,EXP,FACT):-
```

```
        FACT >= '1.0', FACT < '10.0' ,!.
log_scale(EXP,FACT,E_ANS,F_ANS):-
        FACT >= '10.0', plus(EXP,1,EXP_N),
        product(FACT_N,'10.0',FACT),
        log_scale(EXP_N,FACT_N,E_ANS,F_ANS).
log_scale(EXP,FACT,E_ANS,F_ANS):-
        FACT < '1.0', plus(EXP_N,1,EXP),
        FACT < '1.0', plus(EXP_N,1,EXP),
        product(FACT,'10.0',FACT_N),
        log_scale(EXP_N,FACT_N,E_ANS,F_ANS).
power(NUM,POWER,ANS):-
        log_10(NUM,LOG_NUM),
        float(POWER,POWERF),
        product(POWERF,LOG_NUM,TOTAL),
        antilog(TOTAL,ANS),!.
antilog(NUM,ANS) :- float(NUM,NUMF),
        product(NUMF,'2.30258',X), exp(X,ANS).
log_10(IN_NUM,ANS):-float(IN_NUM,NUM),
        log_scale(NUM,EXP,FACT),
        scaled_log_10(FACT,F_ANS),
        float(EXP,EXPF), plus(F_ANS,EXPF,ANS),!.
scaled_log_10(NUM,ANS) :-
        log(NUM,E_ANS),product('2.30258',ANS,E_ANS).
log(NUM,ANS) :-  big_y(NUM,B_Y),product(B_Y,B_Y,BYSQ),
        log_series(B_Y,B_Y,BYSQ,1,271,S_ANS),
        product('2.0',S_ANS,ANS).
log_series(SUM,TERM,BYSQ,LIMIT,LIMIT,SUM).
log_series(GRO_SUM,TERM,BYSQ,P_POWER,LIMIT,ANS):-
        product(TERM,BYSQ,TERM1), plus(2,P_POWER,POWER),
        product(FACT,float(POWER), float(P_POWER)),
        product(TERM1,FACT,NEW_TERM),
        plus(GRO_SUM,NEW_TERM,NEW_SUM),

log_series(NEW_SUM,NEW_TERM,BYSQ,POWER,LIMIT,ANS).
big_y(X,B_Y) :- plus('1.0',TERM1,X),
```

```
        plus('1.0',X,TERM2),
        product(B_Y,TERM2,TERM1).
exp(NUM,ANS):- float(NUM,NUMF),
        exp_series(NUMF,'1.0','1.0',0,35,ANS), !.
exp_series(NUM,SUM,TERM,LIMIT,LIMIT,SUM).
exp_series(NUM,GRO_SUM,TERM,P_POWER,P_LIMIT,ANS):-
        plus(1,P_POWER,POWER),
        product(TERM1,POWER,TERM),
        product(NUM,TERM1,NEW_TERM),
        plus(GRO_SUM,NEW_TERM,NEW_SUM), !,
        exp_series(NUM,NEW_SUM,NEW_TERM,
        POWER,P_LIMIT,ANS).
tan(DEG,ANS):-
        sin(DEG,A),cos(DEG,B),
        ANS is A/B.
/*------END OF sin, cos, power CALCULATIONS----------------*/
```

A3

GRAPHICAL COMMANDS IN PROLOG

1. Output Function

polyline(POINTS).

IN-- POINTS is a list of coordinate pairs in the form of [X1: Y1,X2: Y2,...,Xn: Yn]

polymarker(POINTS).

IN-- POINTS is a list of coordinate pairs as for polyline

text(X: Y,TEXT).

IN-- X: Y is the text position

IN-- TEXT is an Ascii string within single quotes

fill_area(POINTS).

IN-- POINTS is a list of coordinate pairs as for polyline

circle(X: Y,RADIUS).

IN-- X: Y is the centre coordinate of the circle

IN-- RADIUS is the radius of the circle

arc(POINTS).

IN-- POINTS is a list of three coordinate pairs

ellipse(X: Y,LONG,SHORT).

IN-- X: Y is the centre coordinates

IN-- LONG and SHORT are the long and short axises respectively

move(X: Y).

IN-- move the current position to X: Y

2. Output Attributes

line_type(TYPE).
IN-- TYPE is an integer between 0 and 7

line_index(INDEX).
IN-- INDEX is an integer between 0 and 15

marker_type(TYPE).
IN-- TYPE is an integer between 0 and 15

marker_index(INDEX).
IN-- INDEX is an integer between 0 and 15

text_index(INDEX).
IN-- INDEX is an integer between 0 and 15

text_path(PATH).
IN-- Integer,PATH =0,1,2,3 for right,left,up and down

text_precision(PRES).
IN-- Integer PRES= 1,2 for string or stroke precision

text_rotate(ANGLE).
IN-- ANGLE is a real number indicating rotating angle

char_size(WIDTH,HEIGHT,SPACE).
IN-- WIDTH,HEIGHT and SPACE are integer values

char_slant(ANGLE).
IN-- ANGLE is a real number

fill_pattern(PATTERN).
IN-- PATTERN is an integer

pick_id(ID).
IN-- ID is an integer value

3. Transformation Functions

window(NAME,Xl: Yl,Xr: Yr).
IN-- NAME is an integer
IN-- Xl: Yl,Xr: Yr are lower left and upper right position

viewport(NAME,Xl: Yl,Xr: Yr).
IN-- NAME is an integer
IN-- Xl: Yl,Xr: Yr are lower left and upper right position

select_trans(NAME).
IN-- NAME is an integer selecting the transformation

4. Segment Functions

begin_seg(SEG).
IN-- SEG is an integer

end_seg.

delete_seg(SEG).
IN-- SEG is an integer

seg_vis(SEG,VIS).
IN-- SEG is an integer,VIS =0,1 for invisible and visible

insert_seg(SEG).
IN-- SEG is an integer

seg_hilight(SEG,SWITCH).
IN-- SEG is an integer,SWITCH=0,1 for hilight of and on

seg_detect(SEG,DET).
IN-- SEG is an integer,DET=0,1 for indetectable and detectable

seg_prior(SEG,PRIOR).
IN-- both SEG and PRIOR are integer

seg_transform(SEG,X-SCALE,Y-SCALE,ANGLE,X,Y).

IN-- SEG is segment number

IN-- X-SCALE and Y-SCALE are scale factors for x and y direction

IN-- ANGLE is segment rotation angle

IN-- X and Y are segment translation in X and Y direction

seg_position(SEG,X: Y).

IN-- move SEG to new position X: Y

set_pivot(X: Y).

IN-- set segment pivot point

5. Input Functions

input_device(DEVICE).

IN-- string,DEVICE = joydisk or mouse

gin_locate(KEY,X,Y).

OUT-- KEY is the button pressed by the user

OUT-- X and Y are coordinates output from terminal

gin_pick(SEG,ID,KEY,X,Y).

OUT-- SEG and ID are the picked segment number and pick ID

OUT-- KEY is the button pressed by the user

OUT-- X and Y are the coordinates output from terminal

gin_rubber(CODE,MODE).

IN-- CODE is an integer specifying the device-function code

IN-- MODE is an integer selecting rubberbanding operation

pick_aperture(WIDTH).

IN-- WIDTH is an integer for pick operation accuracy

set_snap(X-SPACE,Y-SPACE).

IN-- X-SPACE and Y-SPACE are snap space in X and Y directions

6. Screen Monitor Functions

echo(SWITCH).

IN-- SWITCH = 0,1 for terminal echo off and on

set_dialog(LINES).

IN-- LINES is an integer specifying the dialog area

set_dialog_vis(VIS).

IN-- VIS = 0,1 for dialog area invisible and visible

set_surface(N,BITPLANE).

IN-- N is the number of surfaces,N < 5

IN-- BITPLANE is a list of N integers

set_surf_prior(SURF,PRIOR).

IN-- SURF is an integer name for a surface

IN-- PRIOR is an integer,PRIOR less,priority higher

set_surf_vis(SURF,VIS).

IN-- SURF is an integer name for a surface

IN-- VIS = 0,1,2 for surface invisible,visible and blink

color_mode(MODE).

IN-- MODE = 1,2,3 for RGB,CMY and HLS

A4

A LIST OF THE LOGICAD PROGRAM

```
/*--This is the main program of the LOGICAD system ---*/
logicad:- [menu], [asci], [primitiv], [drawing],
    [constrai], [sincos],
    open_ws, !, repeat,
    ask('TYPE TERMINAL TYPE(tek/pc): ',TERM),
    (TERM = tek; TERM = pc), back_color(66,66,66),
    asserta(term(TERM)), /*border(1), */
    menu(['FILE','EDIT','OPTION','DISPLAY','DESIGN',
    'BUILD','CONST','DRAW'],SEL),
    assert(conwin(no)),
    asserta(con([],[])),
    asserta(id(1)),  /* set the first pick id  */
    asserta(v(1)),   /* set the first vertex number */
    assert(set_linetype(0)), /* set the default line type */
    begin_seg(20),   /* segment 20 is used for editing */
    action(SEL,ACT),ACT.

/* -------- The menu program module ----*/
?-op(32,xfy,&).
?-op(31,xfy,:).
?-op(30,xfy,S).
?-op(32,xfy,@).
?-op(33,xf,#).
/*--LIST is a list of characters--*/
/*--SEL is the selection from the LIST--*/
menu(LIST,SEL):-length(LIST,N),
    surf_visible(1,1),
    window(1,0,0,4095,3070),
    viewport(1,0,0,4095,3070),
    norm_tran(1),
    line_index(1), line_type(0),
    M1 is '4095'/N, M is fix(M1),
    sel_view(1),view_attrib(1,0,1),begin_seg(1),
    fill_pattern('-2'),
    fill_area([(4095,2910),(4095,3070),(0,3070),(0,2910)]),
    polyline(0,2910,4095,2910),
    rectangle(0,4095,0,3070),
```

```
    window(3,0,2910,4095,3070),
    viewport(3,0,2910,4095,3070),
    line_index(1), text_index(1),
    write_hbar(M,M,N),
    write_hmenu(LIST,80,M), end_seg,
    norm_tran(3),
    xy_out_loc(K,X,Y),
    asserta(choose(M,LIST)),
    choose(X,M,LIST,SEL).
choose(X,M,LIST,SEL):-
/* store items in i(Xmin,ITEM,Xmax) */
    store(M,0,LIST,STORE),
/* find ITEM from STORE by X */
    compare(X,STORE,MAIN),
    find_item(MAIN,STORE,i(MIN,MAIN,MAX)),
   ((pull_down(MAIN,ITEMS),
    pull_down(MIN,MAX,ITEMS,SEL&MAIN&PULL));
    SEL = MAIN).
pull_down(MIN,MAX,LIST,SEL&MAIN&PULL):-length(LIST,N),
    surf_visible(2,1), sel_view(2), view_attrib(2,0,0),
    norm_tran(1),
   /* Yp is the Ymin of PULL menu */
    Yp is 2910-(N*150),
    window(2,MIN,Yp,MAX,2910),
    viewport(2,MIN,Yp,MAX,2910),
    norm_tran(2),
    fill_pattern('-2'),
    fill_area[(MIN,Yp),(MAX,Yp),(MAX,2910),(MIN,2910)]),
    line_index(1),
    line_type(0),
    write_hor(2910,150,N,MIN,MAX),
    line_index(2),
    text_index(1),
    write_text(LIST,MIN,160,N),text_index(15),
    reverse(LIST,R), text_index(2),
    xy_out_loc(K,X,Y),
    ((K = 2, fill_pattern('-3'),
    fill_area[(MIN,Yp),(MAX,Yp),(MAX,2905),(MIN,2905)]),
    surf_visible(2,0),norm_tran(3),
    sel_view(1), delete_norm(2),xy_out_loc(KK,XX,YY),
```

```
    choose(M,MLIST),choose(XX,M,MLIST,SEL));
    (store(150,Yp,R,STORE), /* store i(Ymin,Item,Ymax) */
    compare(Y,STORE,PULL), /* find ITEM from STORE by Y */
    ((pop_up(PULL,PLIST),pop_up(PULL,POP_UP,POP_SEL));
    SEL = MAIN&PULL),
    fill_pattern('-3'),
    fill_area([(MIN,Yp),(MAX,Yp),(MAX,2905),(MIN,2905)])),
    surf_visible(2,0),
    sel_view(1),delete_norm(2), norm_tran(1)).
pop_up(1,[1]). /* a fact to pretect no clause */
write_text([],MIN,STEP,N).
write_text([H|T],MIN,STEP,N):-
    XMIN is MIN+80, N1 is N-1, Y is 2910-STEP+50,
    STEP1 is STEP+150,
    text(XMIN,Y,H),
    write_text(T,MIN,STEP1,N1).
write_hor(START,STEP,1,XMIN,XMAX).
write_hor(START,STEP,N,XMIN,XMAX):-
    START1 is START-STEP,
    polyline(XMIN,START1,XMAX,START1),
    N1 is N-1,
    write_hor(START1,STEP,N1,XMIN,XMAX).
compare(X,[i(N,TEXT,N1)|T],SEL):-
    X < N1,SEL = TEXT.          /* find the item picked */
compare(X,[i(N,TEXT,N1)|T],SEL):-   /* from X value */
    compare(X,T,SEL),!.
/* store each item with Xmin and Xmax */
store(M,MM,LIST,STORE):-
    store(M,MM,LIST,[],STORE).
store(M,MM,[],L,STORE):-reverse(L,STORE).
store(M,MM,[H|T],L,STORE):-
    MM1 is MM+M,
    store(M,MM1,T,[i(MM,H,MM1)|L],STORE).
/* find item with Xmin and Xma*/
find_item(A,[i(N,A,N1)|T],i(N,A,N1)):-!.
find_item(A,[i(N,B,N1)|T],L):-
    find_item(A,T,L),!.
write_hbar(M,M1,1).
write_hbar(M,MM,N):-  polyline(MM,2910,MM,3070),
    N1 is N-1,MM1 is MM+M,
```

```
    write_hbar(M,MM1,N1).
write_hmenu([],S,M).
write_hmenu([H|T],START,M):- text(START,2950,H),
    NEW is START + M,
    write_hmenu(T,NEW,M).
menu(['FILE','EDIT','OPTION','DISPLAY','BUILD',
    'CONST','DRAW'],CHOICE).
pull_down('EDIT',['ERASE','MOVE','UNDO','CLEAR']).
pull_down('DISPLAY',['ZOOM','PAN']).
pull_down('OPTION',['SHADE','FONTS','GRID',
        'L_TYPE','D_TYPE']).
pull_down('FILE',['NEW','EXISTING','SAVE','END','QUIT']).
pull_down('CONST',['WINDOW','RADIUS','ANGLE','LEN','LENR',
        'DIST','PERP','PARAL']).
pull_down('BUILD',['NAME','POINT','DIRECTION','START']).
pull_down('DESIGN',['SIMULATE','CHANGE','MECHNISM']).
pull_down('DRAW',['DOT','LINE','POLY',
        'CIRCLE','ARC','FILLET','ELLIPSE','TEXT']).
action('DRAW'&'DOT',draw_dot).
action('DRAW'&'LINE',draw_line).
action('DRAW'&'POLY',draw_poly).
action('DRAW'&'CIRCLE',draw_circle).
action('DRAW'&'ARC',draw_arc).
action('DRAW'&'ELLIPSE',draw_ellipse).
action('DRAW'&'TEXT',draw_text).
action('DRAW'&'FILLET',fillet).
action('DESIGN'&'SIMULATE',simulate).
action('CONST'&'WINDOW',cons_win).
action('CONST'&'RADIUS',radius).
action('CONST'&'DIST',dist).
action('CONST'&'ANGLE',add_angle).
action('CONST'&'LEN',add_len).
action('CONST'&'LENX',lenx).
action('CONST'&'LENY',leny).
action('CONST'&'PERP',add_perp).
action('CONST'&'PARAL',add_paral).
action('OPTION'&'SHADE',shading).
action('OPTION'&'FONTS',fonts).
action('OPTION'&'GRID',griding).
action('OPTION'&'L_TYPE',ltype).
```

```
action('OPTION'&'D_TYPE',dtype).
action('EDIT'&'ERASE',erasing).
action('EDIT'&'MOVE',moving).
action('EDIT'&'UNDO',undoing).
action('EDIT'&'CLEAR',clearing).
action('DISPLAY'&'ZOOM',zooming).
action('DISPLAY'&'PAN',panning).
action('FILE'&'NEW',new_file).
action('FILE'&'EXISTING',existing).
action('FILE'&'SAVE',saving).
action('FILE'&'END',ending).
action('FILE'&'QUIT',quiting).
action('BUILD'&'POINT',point).
action('BUILD'&'DIRECTION',direct).
action('BUILD'&'NAME',name).
action('BUILD'&'START',start).
member(X,[X|T]).
member(X,[Y|T]):- member(X,T).
append(X,LIST,[X|LIST]).
append_list([],RES,RES).
append_list([X|Y],LIST,RES):-
    append_list(Y,[X|LIST],RES),!.
delete(X,[X|T],T).
delete(X,[H|T],[H|T1]):-
    delete(X,T,T1).
length([],0).
length([A|T],L):-
    length(T,N), L is N+1.
reverse(List,R):-
    reverse(List,[],R).
reverse([],R,R).
reverse([H|T],M,R):-
    reverse(T,[H|M],R).
traversal(LIST,RESULT):-
    traversal(LIST,LIST,[],RESULT),!.
traversal([],LIST,RESULT,RESULT).
traversal([H|T],LIST,M,RESULT):-
    delete(H,LIST,LIST1),
    check_top(H,LIST1,TOP),
    traversal(T,LIST,[TOP|M],RESULT).
```

```
check_top(_,[],(TOP&X)&Y):-TOP = X&Y.
check_top(l(ID1,V1,V2),[l(ID2,W1,W2)|T],TOP):-
      (member(V,[V1,V2]),
      member(V,[W1,W2]),
      check_top(l(ID1,V1,V2),T,TOP&ID1:ID2$V));
      check_top(l(ID1,V1,V2),T,TOP).
griding:-  write('Set the grid space'),
      ask('Set X-space:',X),
      ask('Set Y-space:',Y),
      gin_loc_grid(X,Y),!,repeat,
      xy_out_loc(K,XX,YY),
      XX > =2910, try_other(XX).
new_file:-  ask('TYPE FILE NAME:',NAME),
      asserta(file(NAME)),
      assertz(vert([])),
      assertz(line([],[],[])),
      assertz(name([])),
      xy_out_loc(K,X,Y),
      try_other(X).
start:-    [pcessor],extract.
make_list(1,[[]]).
make_list(N,[[]|T]):-
      N1 is N-1,
      make_list(N1,T),!.
ending:-   end_seg,vert(V),
     surf_prior(1,1),surf_prior(2,2),
     line(L,C,F),length(L,L4),
     length(V,L1),length(C,L2),length(F,L3),
     TOTAL is L1*2+L2*3+L3-3, LEN is L2+L3+L4,
     asserta(metric([],0)),
     asserta(orient([],0)),
     asserta(total(TOTAL,0)),xy_out_loc(K,X,Y),
     try_other(X).
quiting:-
      ask('DO YOU WANT TO SAVE THE FILE:  y/n ?',ANS),nl,
     ((ANS = n,close_ws);
      vert(LIST),line(L,C,F),
      name(LISTN),con(D,CONS),file(FILE),
      wwriteq(vert(LIST),FILE),
      wwriteq(line(L,C,F),FILE),
```

```
        wwriteq(name(LISTN),FILE),
        wwriteq(con(D,CONS),FILE),
        close_ws).
window(N,X1,Y1,X2,Y2):-
        asserta(window(N,X1,Y1,X2,Y2)).
viewport(N,X1,Y1,X2,Y2):-
        asserta(viewport(N,X1,Y1,X2,Y2)).
norm_tran(N):- window(N,X1,Y1,X2,Y2),
        window(X1,Y1,X2,Y2),
        viewport(N,X3,Y3,X4,Y4),
        viewport(X3,Y3,X4,Y4).
delete_norm(N):- retract(window(N,_,_,_,_)),
        retract(viewport(N,_,_,_,_)).
hilight(ID):-  line(LINE,C,F),
        find_line(ID,LINE,l(ID,N1,N2)),
        vert(VERT), find_vert(N1,VERT,v(N1,X1,Y1)),
        find_vert(N2,VERT,v(N2,X2,Y2)),
        line_index(1),polyline(X1,Y1,X2,Y2),
        line_index(2),
        asserta(poly(hi,X1,Y1,X2,Y2)).
hicircle(ID):-  line(L,C,F),
        find_circle(ID,C,c(ID,X,Y,R)),
        line_index(1),circle(X,Y,R),
        line_index(2),
        asserta(circle(hi,X,Y,R)).
find_line(ID,[l(ID,N1,N2)|T],l(ID,N1,N2)).
find_line(ID,[l(ID1,N1,N2)|T],L):-
        find_line(ID,T,L).
find_vert(N,[v(N,X,Y)|T],v(N,X,Y)).
find_vert(N,[v(N1,X1,Y1)|T],V):-
        find_vert(N,T,V).
find_circle(ID,[c(ID,X,Y,R)|T],c(ID,X,Y,R)).
find_circle(ID,[c(ID1,X1,Y2,R1)|T],L):-
        find_line(ID,T,C).

/*--- This program support the drawing functions--*/
try_other(X):- choose(M,LIST),
        choose(X,M,LIST,SEL),
        action(SEL,ACT),ACT.
/*--- line drawing command--- */
```

```
draw_line:- xy_out_loc(K,X,Y),
      set_linetype(T), line_type(T),
      ((Y > =2910,try_other(X));
      draw_line1(X,Y)).
draw_line1(X,Y):-
       xy_out_loc(K1,X1,Y1),
       id(ID),pick_id(ID),ID1 is ID+1,
       retract(id(ID)),asserta(id(ID1)),
       v(N),N1 is N+1,N2 is N1+1,
       retract(v(N)),asserta(v(N2)),
       polyline(X,Y,X1,Y1), draw_line.
/*----end of line drawing command----------*/
/*----polyline drawing command-------------*/
draw_poly:- nl,write('DRAW POLYLINE:'),nl,
       set_linetype(T), line_type(T),
       xy_out_loc(K,X,Y),
       v(N),vert(VERT),retract(vert(VERT)),
       assertz(vert([v(N,X,Y)|VERT])),
       draw_poly(N,X,Y,X,Y).
/* keep the first point Xo,Yo of a polyline */
draw_poly(No,Xo,Yo,X,Y):- nl,write('DRAW POLYLINE:'),nl,
       xy_out_loc(K,X1,Y1),
       v(N),id(ID),ID1 is ID+1,
       ((Y1 > =2910,pick_id(ID),polyline(X,Y,Xo,Yo),
       retract(id(ID)),asserta(id(ID1)),
       line(LINE,C,F),retract(line(LINE,C,F)),
       assertz(line([l(ID,N,No)|LINE],C,F)),
       try_other(X1));
       retract(v(N)),N1 is N+1,asserta(v(N1)),
       vert(VERT),retract(vert(VERT)),
       assertz(vert([v(N1,X1,Y1)|VERT])),
       retract(id(ID)),asserta(id(ID1)),
       pick_id(ID),line(LINE,C,F),retract(line(LINE,C,F)),
       assertz(line([l(ID,N,N1)|LINE],C,F)),
       polyline(X,Y,X1,Y1),
       draw_poly(No,Xo,Yo,X1,Y1)).
draw_text:- ask('TYPE THE TEXT(OR TYPE  no  TO END):',TEXT),
       (((TEXT=no;TEXT=n),xy_out_loc(K,X,Y),try_other(X));
       (nl,write('Pick a point to put the text:'),
       xy_out_loc(K,X,Y),
```

```
        text(X,Y,TEXT),draw_text)).
draw_dot:-  xy_out_loc(K,X,Y),
        ((Y > =2910,try_other(X));
        polymarker(X,Y), draw_dot).
draw_circle:- nl,write('LOCATE THE CENTRE OF THE CIRCLE:'),nl,
        set_linetype(T),
        line_type(T),
        xy_out_loc(K,X,Y),
       ((Y > =2910,try_other(X));
       draw_circle1(X,Y)).
draw_circle1(X,Y):-
       nl,write('PICK A POINT ON THE CIRCLE:'),nl,
       xy_out_loc(K1,X1,Y1),
       id(ID),ID1 is ID+1,
       line(L,C,F),retract(line(L,C,F)),
       pick_id(ID),retract(id(ID)),asserta(id(ID1)),
       circle(X,Y,X1,Y1,RAD),
       assertz(line(L,[c(ID,X,Y,RAD)|C],F)),
       draw_circle.
draw_arc:-  nl,write('PICK A POINT ON ARC:'),nl,
       set_linetype(T), line_type(T),
       xy_out_loc(K1,X1,Y1),
       ((Y1 >2910,try_other(X1));
       draw_arc(X1,Y1)).
draw_arc(X1,Y1):- nl,write('PICK A POINT ON ARC:'),nl,
       xy_out_loc(K2,X2,Y2),
       nl,write('PICK A POINT ON ARC:'),nl,
       xy_out_loc(K3,X3,Y3),
       arc(X1,Y1,X2,Y2,X3,Y3), draw_arc.
ltype:-   ask('SET LINE TYPE(from 0 to 7):',ANS),
     line_type(ANS),
     retract(set_linetype(_)),
     assert(set_linetype(ANS)),
     nl,write('OK'),
     xy_out_loc(K,X,Y), try_other(X).
dtype:-     ask('SET MARKER TYPE(from 0 to 10):',ANS),
     marker_type(ANS),
     nl,write('OK'),
     xy_out_loc(K,X,Y),
     try_other(X).
```

```
shading:-  ask(' SET SHADING PATTERN:',PART),
     nl,write('PICK BOUNDARY POINTS:'),nl,
     pick_points(LIST),
     fill_pattern(PART), fill_area(LIST),
     xy_out_loc(K,X,Y), try_other(X).
pick_points(LIST):-
     pick_points([],LIST),!.
pick_points(L,LIST):-
     xy_out_loc(K,X,Y),
     ((Y > =2910,pick_ok(L,LIST));
     pick_points([(X,Y)|L],LIST)).
pick_ok(LIST,LIST).

/*--Program for adding constraints--*/
fillet:-  write('Pick two lines'),nl,
     gin_pick(K,X,Y,SEG,ID),
     hilight(ID),
     ((Y > =2910,try_other(X));
     write('Pick the other line:'),
     gin_pick(K1,X1,Y1,SEG,ID1),
     hilight(ID1),
     ask('Input the Radius:',RAD),
     retract(poly(hi,Xh1,Yh1,Xh2,Yh2)),
     retract(poly(hi,Xhh1,Yhh1,Xhh2,Yhh2)),
     polyline(Xh1,Yh1,Xh2,Yh2),
     polyline(Xhh1,Yhh1,Xhh2,Yhh2),
     line(L,C,F),retract(line(L,C,F)),
     assertz(line(L,C,[f: ID: ID1$RAD|F])),
     retract(con(D,CONS)),
     asserta(con(D,[f: ID: ID1$RAD|CONS])),
     fillet).
add_len:-  nl,write('CONSTRAIN LENGTH: '),nl,
     gin_pick(K,X,Y,SEG,ID),
     ((Y > =2910,try_other(X));
     hilight(ID),
     ask('INPUT LENGTH: ',LEN),
     poly(hi,Xh1,Yh1,Xh2,Yh2),
     polyline(Xh1,Yh1,Xh2,Yh2),
     retract(poly(hi,Xh1,Yh1,Xh2,Yh2)),
     ((conwin(yes),
```

```
    conwin(X3,Y3,ROWS,CCC),
    Y4 is Y3-80,retract(conwin(X3,Y3,ROWS,CCC)),
    asserta(conwin(X3,Y4,ROWS,[len: ID$LEN|CCC])),
    text(X3,Y4,len),X4 is X3+3*50,text(X4,Y4,':'),
    X5 is X4+50,X6 is X5+100,
   ((name(LISTN),length(LISTN,NN),NN>0,
    find_name(ID,LISTN,NAME),
    text(X5,Y4,NAME));text(X5,Y4,ID)),
    text(X6,Y4,'S'),X7 is X6+50,
    text(X7,Y4,LEN));true),
    con(D,CONS),retract(con(D,CONS)),
    asserta(con(D,[len: ID$LEN|CONS])),
    add_len).
ad_dp:- nl,write('A distance from a point to a line: '),nl,
    write('Pick a point'),nl,
    xy_out_loc(Kp,Xp,Yp),
    write('Pick a line'),nl,
    gin_pick(K,X,Y,SEG,ID),
    ((Y>=2910,try_other(X));
    hilight(ID),
    ask('Input distance: ',DIST),
    poly(hi,Xh1,Yh1,Xh2,Yh2),
    polyline(Xh1,Yh1,Xh2,Yh2),
    retract(poly(hi,Xh1,Yh1,Xh2,Yh2)),
    ((conwin(yes),
    conwin(X3,Y3,ROWS,CCC),
    Y4 is Y3-80,retract(conwin(X3,Y3,ROWS,CCC)),
    asserta(conwin(X3,Y4,ROWS,[dp: ID$DIST|CCC])),
    text(X3,Y4,dp),X4 is X3+4*50,text(X4,Y4,':'),
    X5 is X4+50,X6 is X5+100,
    ((name(LISTN),length(LISTN,NN),NN>0,
    find_name(ID,LISTN,NAME),
    text(X5,Y4,NAME));text(X5,Y4,ID)),
    text(X6,Y4,'$'),X7 is X6+50,
    text(X7,Y4,DIST));true),
    con(D,CONS),retract(con(D,CONS)),
    asserta(con(D,[dp: ID$DIST|CONS])), ad_dp).
radius:-  gin_pick(K,X,Y,SEG,ID),
    ((Y>=2910,try_other(X));
    hicircle(ID),circle(hi,X0,Y0,R),
```

```
        retract(circle(ni,_,_,_)),
        ask('INPUT RADIUS:',RAD),
        circle(X0,Y0,R),
        ((conwin(yes),
        conwin(X3,Y3,ROWS,CCC),
        Y4 is Y3-80,retract(conwin(X3,Y3,ROWS,CCC)),
        asserta(conwin(X3,Y4,ROWS,[radius: ID$RAD|CCC])),
        text(X3,Y4,radius),X4 is X3 + 6*50,text(X4,Y4,':'),
        X5 is X4 + 50,X6 is X5 + 100,
        ((name(LISTN),length(LISTN,NN),NN > 0,
        find_name(ID,LISTN,NAME),
        text(X5,Y4,NAME));text(X5,Y4,ID)),
        text(X6,Y4,'S'),X7 is X6 + 50,
        text(X7,Y4,RAD));true),
        con(D,CONS),retract(con(D,CONS)),
        asserta(con(D,[rad: ID$RAD|CONS])),
        radius).
write_to_nth(TERM,[H|T],1,[[TERM|H]|T]).
write_to_nth(TERM,[H|T],N,[H|T1]):-
        N1 is N-1,
        write_to_nth(TERM,T,N1,T1).
dist:-      nl,write('       PICK A LINE:'),nl,
        gin_pick(K,X,Y,SEG,ID),
        ((Y > =2910,try_other(X));
        hilight(ID),poly(hi,Xi1,Yi1,Xi2,Yi2),
        retract(poly(_,_,_,_,_)),
        nl,write('PICK ANOTHER LINE:'),nl,
        gin_pick(K1,X1,Y1,SEG,ID1),
        hilight(ID1),poly(hi,Xii1,Yii1,Xii2,Yii2),
        retract(poly(_,_,_,_,_)),
        ask('INPUT THE DISTANCE:',DIST),
        polyline(Xi1,Yi1,Xi2,Yi2),
        polyline(Xii1,Yii1,Xii2,Yii2),
        ((conwin(yes),
        conwin(X3,Y3,ROWS,CCC),
        Y4 is Y3-80,retract(conwin(X3,Y3,ROWS,CCC)),
        asserta(conwin(X3,Y4,ROWS,[dist: ID&ID1$DIST|CCC])),
        text(X3,Y4,dist),X4 is X3 + 4*50,text(X4,Y4,':'),
        X5 is X4 + 50,X6 is X5 + 100,name(LISTN),
        ((name(LISTN),length(LISTN,NN),NN > 0,
```

```
    find_name(ID,LISTN,NAME),
    text(X5,Y4,NAME));text(X5,Y4,ID)),
    text(X6,Y4,'&'),X7 is X6 + 50,
    ((find_name(ID1,LISTN,NAME1),
    text(X7,Y4,NAME1));text(X7,Y4,ID1)),
    X8 is X7 + 100,X9 is X8 + 50,
    text(X8,Y4,'$'),
    text(X9,Y4,DIST));true),
    con(D,CONS),retract(con(D,CONS)),
    asserta(con(D,[dist: ID&ID1$DIST|CONS])), dist).
add_angle:- nl,write('      PICK A LINE:'),nl,
    gin_pick(K,X,Y,SEG,ID),
    ((Y > =2910,try_other(X));
    hilight(ID), poly(hi,Xi1,Yi1,Xi2,Yi2),
    retract(poly(_,_,_,_,_)),
    nl,write('      PICK ANOTHER LINE:');nl,
    gin_pick(K1,X1,Y1,SEG1,ID1),
    hilight(ID1),poly(hi,Xii1,Yii1,Xii2,Yii2),
    retract(poly(_,_,_,_,_)),
    ask('INPUT ANGLE: ',AN),
    polyline(Xi1,Yi1,Xi2,Yi2),
    polyline(Xii1,Yii1,Xii2,Yii2),
    ((conwin(yes),
    conwin(X3,Y3,ROWS,CCC),
    Y4 is Y3-80,retract(conwin(X3,Y3,ROWS,CCC)),
    asserta(conwin(X3,Y4,ROWS,[angle: ID&ID1$AN|CCC])),
    text(X3,Y4,angle),X4 is X3 + 5*50,text(X4,Y4,':'),
    X5 is X4 + 50,X6 is X5 + 100,name(LISTN),
    ((name(LISTN),length(LISTN,NN),NN > 0,
    find_name(ID,LISTN,NAME),
    text(X5,Y4,NAME));text(X5,Y4,ID)),
    text(X6,Y4,'&'),X7 is X6 + 50,
    ((find_name(ID1,LISTN,NAME1),
    text(X7,Y4,NAME1));text(X7,Y4,ID1)),
    X8 is X7 + 100,X9 is X8 + 50,
    text(X8,Y4,'$'),
    text(X9,Y4,AN));true),
    con(D,CONS),retract(con(_,_)),
    asserta(con(D,[an: ID&ID1$AN|CONS])),
    add_angle).
```

```
add_paral:- nl,write('PICK A LINE:'),nl,
    gin_pick(K,X,Y,SEG,ID),
    ((Y > =2910,try_other(X));
    hilight(ID),poly(hi,Xi1,Yi1,Xi2,Yi2),
    retract(poly(_,_,_,_,_)),
    nl,write('PICK ANOTHER LINE:'),nl,
    gin_pick(K1,X1,Y1,SEG1,ID1),
    hilight(ID1),poly(hi,Xii1,Yii1,Xii2,Yii2),
    retract(poly(_,_,_,_,_)),
    ((conwin(yes),
    conwin(X3,Y3,ROWS,CCC),
    Y4 is Y3-80,retract(conwin(X3,Y3,ROWS,CCC)),
    asserta(conwin(X3,Y4,ROWS,[paral: ID&ID1|CCC])),
    text(X3,Y4,paral),X4 is X3 + 5*50,text(X4,Y4,':'),
    X5 is X4 + 50,X6 is X5 + 100,name(LISTN),
    ((name(LISTN),length(LISTN,NN),NN > 0,
    find_name(ID,LISTN,NAME),
    text(X5,Y4,NAME));text(X5,Y4,ID)),
    text(X6,Y4,'&'),X7 is X6 + 50,
    ((find_name(ID1,LISTN,NAME1),
     text(X7,Y4,NAME1));text(X7,Y4,ID1)));
     true),
     con(D,CONS),retract(con(_,_)),
     asserta(con(D,[paral: ID&ID1|CONS])),
     polyline(Xi1,Yi1,Xi2,Yi2),
     polyline(Xii1,Yii1,Xii2,Yii2),
     add_paral).
add_perp:- nl,write('PICK A LINE:'),nl,
    gin_pick(K,X,Y,SEG,ID),
    ((Y > =2910,try_other(X));
    hilight(ID),poly(hi,Xi1,Yi1,Xi2,Yi2),
    retract(poly(_,_,_,_,_)),
    nl,write('PICK ANOTHER LINE:'),nl,
    gin_pick(K1,X1,Y1,SEG1,ID1),
    hilight(ID1),poly(hi,Xii1,Yii1,Xii2,Yii2),
    retract(poly(_,_,_,_,_)),
    ((conwin(yes),
    conwin(X3,Y3,ROWS,CCC),
    Y4 is Y3-80,retract(conwin(X3,Y3,ROWS,CCC)),
    asserta(conwin(X3,Y4,ROWS,[perp: ID&ID1|CCC])),
```

```
      text(X3,Y4,perp),X4 is X3 + 4*50,text(X4,Y4,':'),
      X5 is X4 + 50,X6 is X5 + 100,name(LISTN),
      ((name(LISTN),length(LISTN,NN),NN > 0,
      find_name(ID,LISTN,NAME),
      text(X5,Y4,NAME));text(X5,Y4,ID)),
      text(X6,Y4,'&'),X7 is X6 + 50,
      ((find_name(ID1,LISTN,NAME1),
      text(X7,Y4,NAME1));text(X7,Y4,ID1)));
      true),
      con(D,CONS),retract(con(_,_)),
      asserta(con(D,[perp: ID&ID1|CONS])),
      polyline(Xi1,Yi1,Xi2,Yi2),
      polyline(Xii1,Yii1,Xii2,Yii2),
      add_perp).
cons_win:-  nl,write('1.CREATE-WINDOW;2.VISIBLE;3.INVISIBLE'),nl,
      read(CHOICE),
      ((CHOICE = 2,v_seg(15));
      (CHOICE = 3,i_seg(15));
      CHOICE = 1,
      nl,write('Pick two points for constraint window'),nl,
      nl,write('PICK THE LOWER LEFT POINT:'),nl,
      xy_out_loc(K1,X1,Y1),
      nl,write('PICK THE UPPER RIGHT POINT:'),nl,
      xy_out_loc(K2,X2,Y2),
      begin_seg(15),   /* USED FOR CONSTRAIT WINDOW */
      fill_pattern('-1'),
      fill_area([(X1,Y1),(X2,Y1),(X2,Y2),(X1,Y2)]),
      Y3 is Y2-100,polyline(X1,Y3,X2,Y3),
      X3 is X1 + 100,X4 is X2-100,Y4 is Y3 + 20,
      polyline(X3,Y3,X3,Y2),polyline(X4,Y3,X4,Y2),
      X5 is X3 + 150,text(X5,Y4,'CONSTRAINTS'),
      X6 is X1 + 20,text(X6,Y4,'I'),
      X7 is X4 + 20,text(X7,Y4,'M'),
      ROWS is (Y3-Y1)/60, Y5 is Y3-20,
      assert(conwin(X3,Y5,ROWS,[])),
      asserta(conwin(yes)),retract(conwin(no)),
      window(5,X1,Y1,X2,Y2),
      viewport(5,X1,Y1,X2,Y2),
      rectangle(X1,X2,Y1,Y2),end_seg),
      xy_out_loc(K,X,Y), try_other(X).
```

```
name:-      nl,write('PICK A LINE SEGMENT:'),nl,
      gin_pick(K,X,Y,SEG,ID),
      ((Y > =2910,try_other(X));
      ((hilight(ID),poly(hi,Xi1,Yi1,Xi2,Yi2));
      (hicircle(ID),circle(hi,X0,Y0,R))),
      name(LISTn),retract(name(_)),
      ask('Type the name of element',NAME),
      nl,write(' Put to the proper location'),nl,
      xy_out_loc(K1,X1,Y1),
     ((integer(Xi1),polyline(Xi1,Yi1,Xi2,Yi2));
      circle(X0,Y0,R)),
      assert(name([n(ID,NAME,X1,Y1)|LISTn])),
      text(X1,Y1,NAME),name).
direct:-    nl,write('        FIX DIRECTTION:'),nl,
       gin_pick(K,X,Y,SEG,ID),
       hilight(ID),poly(hi,Xi1,Yi1,Xi2,Yi2),
       con(D,LISTn),
       ask('INPUT THE DIRECTION:',DIR),
       polyline(Xi1,Yi1,Xi2,Yi2),
       asserta(con([IDSDIR|D],LISTn)),
       xy_out_loc(K1,X1,Y1), try_other(X1).
point:-     nl,write('FIX A POINT:'),nl,
       vert(VERT),!,
       xy_out_loc(K,X,Y),   /* how to find the point ? */
       find_point(VERT,[X,Y],V),
       marker_type(3),polymarker(X,Y),
       nl,write('OK'),nl,
       con(D,LISTn),retract(con(_,_)),
       asserta(con([V$X$Y|D],LISTn)),
       xy_out_loc(K1,X1,Y1), try_other(X1).
find_point([],_,_):-nl,write('PICK AGAIN:'),nl.
find_point([v(V,X,Y)|T],[X,Y],V).
find_point([v(N,X1,Y1)|T],[X,Y],V):-
       find_point(T,[X,Y],V),!.
find_name(ID,[n(ID,RES,_,_)|T],RES).
find_name(ID,[n(ID1,NAME,_,_)|T],RES):-
       find_name(ID,T,RES).
check_metric(ID):- metric(METRIC,N),
       member(ID,METRIC),
       write('REDUNDENT CONSTRAINT'),!.
```

```
check_metric(ID):- metric(METRIC,N),total(TOTAL,CON),
        TOTAL = <CON,nl,
        write('REDUNTENT CONSTRAINT'),!.
check_metric(ID):- total(T,CON),
        T = := CON + 1,
        metric(M,N),N1 is N + 1,
        retract(metric(M,N)),asserta(metric([ID|M],N1)),
        retract(total(T,CON)),asserta(total(T,T)),
        write('The model is just properly defined after this').
check_metric(ID):- metric(METRIC,N),total(TOTAL,CON),
        retract(metric(METRIC,N)),
        N1 is N + 1,asserta(metric([ID|METRIC],N1)),
        CON1 is CON + 1,retract(total(TOTAL,CON)),
        asserta(total(TOTAL,CON1)).
check_orient(ID1,ID2):-
        orient(ORIENT,N),
        member(ID1,ORIENT),member(ID2,ORIENT),
        write('REDUNDENT CONSTRAINT'),!.
check_orient(ID1,ID2):-
        total(TOTAL,CON), TOTAL = <CON,
        write('REDUNDENT CONSTRAINT'),!.
check_orient(ID1,ID2):-
        orient(O,N),total(T,CON),T = := CON + 1,
        not member(ID1,O),not member(ID2,O),
        retract(orient(O,N)),retract(total(T,CON)),
        N1 is N + 1,CON1 is CON + 1,
        asserta(orient([ID1,ID2|O],N1)),
        asserta(total(T,CON1)),
        write('The model is just properly defined after this').
check_orient(ID1,ID2):- total(T,CON),T = := CON + 1,
        orient(O,N),N1 is N + 1,CON1 is CON + 1,
        retract(orient(O,N)),retract(total(T,CON)),
        asserta(total(T,CON1)),
        ((not member(ID1,O),asserta(orient([ID1|O],N1)));
        asserta(orient([ID2|O],N1))),
        write('The model is just properly defined after this').
check_orient(ID1,ID2):-
        orient(O,N),total(T,CON),
        not member(ID1,O),not member(ID2,O),
        retract(orient(O,N)),retract(total(T,CON)),
```

```
        N1 is N+1,CON1 is CON+1,
        asserta(orient([ID1,ID2|O],N1)),
        asserta(total(T,CON1)).
check_orient(ID1,ID2):-
        orient(O,N),total(T,CON),N1 is N+1,CON1 is CON+1,
        retract(orient(O,N)),retract(total(T,CON)),
        asserta(total(T,CON1)),
        ((not member(ID1,O),asserta(orient([ID1|O],N1)));
        asserta(orient([ID2|O],N1))).
ask(QUEST,ANS):-nl,tab(8),write(QUEST),nl,wread(ANS),nl.

/*----- Process the constraints ------ */
extract:-
        (con([IDSD,VSXSY],LIST);
        con([VSXSY,IDSD],LIST)),
        process(LIST,RES),
       /* file'const' store deduced inform */
        line(LINE,C,F),
        reverse(LINE,LINE1),
        wwriteq(deduced(d,user:[ID]),const),
        p-angle(RES,IDSD,LISTOK,[ID]), write(LISTOK),nl,
        wwriteq(deduced(v,user:V),const),
        p_vertex(LINE1,V,LISTOK,LISTALL),write(LISTALL),
        [unify],!,unify(LISTALL),
        xy_out_loc(KK,XX,YY),
        try_other(XX).
p-angle(LIST,IDSD,LISTOK,RECORD):-
        /* select the IDth item from LIST */
        select(ID,LIST,NTH),
       /* delete NTH from LIST */
        delete(NTH,LIST,LIST1),
       /* delete all an:_ related items*/
        delete_all(an:_,NTH,NTH1),
       /* assert to original */
        assert_nth([d|NTH1],ID,LIST1,LIST2),
        subtract(NTH,NTH1,SUB),get_number(SUB,NUM),
        append_list(RECORD,NUM,NEW),
        wwriteq(deduced(d,ID:NUM),const),
        transfer(SUB,LIST2,LISTOK,NEW).
transfer([],LISTOK,LISTOK,RECORD).
```

```
transfer([an:ID|T],LIST,LISTOK,RECORD):-
      select(ID,LIST,NTH),
     ((member(an:_,NTH),
      delete(NTH,LIST,LIST1),
      delete_all(an:_,NTH,NTH1),
      assert_nth([d|NTH1],ID,LIST1,LIST2),
      subtract(NTH,NTH1,SUB),get_number(SUB,NUM),
      subtract(NUM,RECORD,NEW),length(NEW,LLL),
     ((LLL > 0,wwriteq(deduced(d,ID:NEW),const));
     true),
     append_list(NEW,RECORD,CORD),
     length(SUB,LEN),
     ((LEN > 1,append_list(SUB,T,TT),
     transfer(TT,LIST2,LISTOK,CORD));
    (append_list(SUB,T,TT),
     LEN = 1,transfer(TT,LIST2,LISTOK,CORD));
     LEN = 0,transfer(T,LIST,LISTOK,CORD))));
     transfer(T,LIST,LISTOK,RECORD)).
p_vertex([],V,LISTALL,LISTALL).
p_vertex([l(N,V1,V2)|T],V,LISTOK,LISTALL):-
     not V1 = V,not V2 = V,
     p_vertex(T,V,LISTOK,LISTALL).
p_vertex([l(N,_,V)|T],V,LISTOK,LISTALL):-
     select(N,LISTOK,NTH),
     ((NTH = [known],
     p_vertex(T,V,LISTOK,LISTALL));
     (delete(NTH,LISTOK,LISTO),
     assert_nth([v$V|NTH],N,LISTO,LIST1),
     asserta(vset([V])), /* keep the deduced vertices */
     p_vertex(T,V,LIST1,LISTALL))).
p_vertex([l(N,V,_)|T],V,LISTOK,LISTALL):-
     select(N,LISTOK,NTH),
     ((NTH = [known],
     p_vertex(T,V,LISTOK,LISTALL));
     (delete(NTH,LISTOK,LISTO),
     assert_nth([v$V|NTH],N,LISTO,LIST1),
     p_vertex(T,V,LIST1,LISTALL))).
select(1,[H|T],H).
select(N,[H|T],Nth):-
     N1 is N-1, select(N1,T,Nth).
```

```
assert_nth(X,1,T,[X|T]).
assert_nth(X,N,[H|T],[H|T1]):-
    N1 is N-1,
    assert_nth(X,N1,T,T1).
delete_all(X,[X|T],T1):- (member(X,T),
    delete_all(X,T,T1)); T=T1.
delete_all(X,[H|T],[H|T1]):-
    delete_all(X,T,T1),!.
member(X,[X|T]).
member(X,[H|T]):-  member(X,T).
subtract(DIFFER,[],DIFFER):-true.
subtract(L1,[H|T],DIFFER):-
    ((member(H,L1),
    delete(H,L1,LL),
    subtract(LL,T,DIFFER));
    subtract(L1,T,DIFFER)).
get_number([],[]).
get_number([an:ID|T],[ID|T1]):-
    get_number(T,T1).
process(LIST,RES):-
    line(L,C,F),
    length(L,LEN1),length(C,LEN2),
    length(F,LEN3),LEN is LEN1+LEN2+LEN3,
    asserta(len(LEN)),
    make_list(LEN,EMLIST),
    process(LIST,EMLIST,RES).
process([],RES,RES).
process([H|T],EMLIST,RES):-
    ((H=len:ID$V,
    write_to_nth(len,EMLIST,ID,EMLIST1),
    process(T,EMLIST1,RES));
    (H=leny:ID$V,
    write_to_nth(leny,EMLIST,ID,EMLIST1),
    process(T,EMLIST1,RES));
    (H=lenx:ID$V,
    write_to_nth(lenx,EMLIST,ID,EMLIST1),
    process(T,EMLIST1,RES));
    (H=rad:ID$V,
    write_to_nth(radius,EMLIST,ID,EMLIST1),
    process(T,EMLIST1,RES));
```

```
        (H = paral:ID&ID1,
        write_to_nth(an:ID1,EMLIST,ID,EMLIST1),
        write_to_nth(an:ID,EMLIST1,ID1,EMLIST2),
        process(T,EMLIST2,RES));
        (H = perp:ID&ID1,
        write_to_nth(an:ID1,EMLIST,ID,EMLIST1),
        write_to_nth(an:ID,EMLIST1,ID1,EMLIST2),
        process(T,EMLIST2,RES));
        (H = an:ID&ID1SAN,
        write_to_nth(an:ID1,EMLIST,ID,EMLIST1),
        write_to_nth(an:ID,EMLIST1,ID1,EMLIST2),
        process(T,EMLIST2,RES));
        (H = dist:ID&ID1SDIST,
        write_to_nth(an:ID1,EMLIST,ID,EMLIST1),
        write_to_nth(an:ID,EMLIST1,ID1,EMLIST2),
        write_to_nth(dist:ID1,EMLIST2,ID,EMLIST3),
        write_to_nth(dist:ID,EMLIST3,ID1,EMLIST4),
        process(T,EMLIST4,RES))),!.

/*--- The unification processes---*/
unify(OBLIST):-
      ((known(OBLIST),nl,write('Succeed'));
      (see(rule), unify1(OBLIST))).
unify1(OBLIST):-
       read(ITEM),
       ((ITEM = eof,seen,unify(OBLIST));
       (ITEM = p(Pn,[G,P]),
       check_unify(Pn,P,OBLIST,1,OBLIST,OB,RECORD),
       unify1(OB));
      (ITEM = ps(Pn,[P1,P2]),
       check_unify_ps(Pn,P1,P2,OBLIST,1,OBLIST,OB),
       unify1(OB))).
check_unify(Pn,P,[],LEN,OBLIST1,OB,RECORD):-
      ((atom(RECORD),
      check_unify(Pn,P,OBLIST1,1,OBLIST1,OB,NEWREC));
      OB = OBLIST1).
check_unify(Pn,P,[H|T],LEN,OBLIST,OB,RECORD):-
      subtract(P,H,SUB),   /* P unify with H */
      make_varable(P,P1),
     ((length(SUB,0),       /* succeed */
```

CHAPTER 1

INTRODUCTION

Existing CAD systems can assist many areas of design work but designers are not satisfied with what existing CAD systems offer. This chapter identifies aspects of the ineffectiveness and inefficiency of existing CAD systems when used in the design process. It is argued that artificial intelligence (AI) offer many features such as declarative knowledge bases, automated reasoning techniques and intelligent control methods which are suitable for intelligent CAD (ICAD) system research and development. AI-based systems can thus help the designer at a higher level of abstraction in the design process than existing CAD systems. In this chapter, possible approaches to achieve intelligent CAD systems are outlined. The advantages and disadvantages of current programming paradigms for ICAD applications are described. Finally, the role of geometry in engineering design system development is emphasised.

1.1 Development of Computer-Aided Design

Human beings have long been involved in the design process but the importance and the meaning of design is not fully understood even in the computer age. In the early 1970s, leading international management consultants concluded that

> since most of the expenditure in any organization occured in the manufacturing end of the business, and less than 5% at the design end, CAD would contribute little overall improvement and any investment would not be remunerated [Llewelyn 1989].

```
        RECORD = positive,
        delete(H,OBLIST,OBLIST1),
        find_v(H,Vk),line(LINES,C,F),reverse(LINES,LINE1),
        assert_nth([known],LEN,OBLIST1,OBLIST2),
        find_ov(LINES,LEN,Vk,Vo),
        p_vertex(LINE1,Vo,OBLIST2,OBLIST3), write(OBLIST3),
        wwrite(deduced(seg,Pn:LEN),const),
        LEN1 is LEN + 1,
        check_unify(Pn,P1,T,LEN1,OBLIST3,OB,RECORD));
      (LEN1 is LEN + 1,
      check_unify(Pn,P1,T,LEN1,OBLIST,OB,RECORD))).
check_unify_ps(Pn,P1,P2,[HL],LEN,[H|T],OB):-
      (subtract(P1,HL,SUB1),
       subtract(P2,H,SUB2),
       length(SUB1,0),
       length(SUB2,0),
       line(LS,C,F),
       reverse(LS,LS1),
       join(LEN,1,LS,V),
       p_vertex(LS1,V,[H|T],OB));
       OB = [H|T].
check_unify_ps(Pn,P1,P2,[H1,H2|T],LEN,OBLIST,OB):-
     ((subtract(P1,H1,S1),
       subtract(P2,H2,S2),
       length(S1,0),
       length(S2,0),
       line(LS,C,F),reverse(LS,LS1),
       LEN1 is LEN + 1,
       join(N1,N2,LS,V),
       p_vertex(LS1,V,OBLIST,OBLIST1),
       wwrite(deduced(v,Pn:V)),
       check_unify_ps(Pn,P1,P2,[H2|T],LEN1,OBLIST1,OB));
     (LEN1 is LEN + 1,
       check_unify_ps(Pn,P1,P2,[H2|T],LEN1,OBLIST,OB))).
known([]).
known([H|T]):-
      H = [known], know(T),!.
find_v([v$V|T],V).
find_v([H|T],V):-
      not H = _$_, find_v(T,V).
```

```
find_ov([l(ID,V1,V2)|T],LEN,V,OV):-
      find_ov(T,LEN,V,OV).
find_ov([l(LEN,V1,V2)|T],LEN,V,OV):-
      ((V = V1,OV = V2);
      (V = V2,OV = V1)).
make_varable(LIST,LIST1):-
      delete(vSV,LIST,LIST2),
      LIST1 =[vS_|LIST2].
join(N1,N2,LINES,V):-
     join(N1,N2,LINES,[],N1N2),
     join(N1N2,V).
join(N1,N2,[],N1N2,N1N2).
join(N1,N2,[l(N,_,_)|T],NN,N1N2):-
     join(N1,N2,T,NN,N1N2).
join(N1,N2,[l(N,V1,V2)|T],NN,N1N2):-
     (N = N1;N = N2),
     join(N1,N2,T,[l(N,V1,V2)|NN],N1N2).
join([l(N1,V1,V2),l(N2,V3,V4)],V):-
     ((V1 = V3,V = V3);
     (V1 = V4,V = V4);
     (V2 = V3,V = V3);
     (V2 = V4,V = V4)).
```