LOGIC PROGRAMMING IN ELECTRICAL MACHINE DESIGN

by

Mukolera, J.R.

A thesis submitted for the degree of Doctor of Philosophy

of the University of London.

December 1986

Department of Electrical Engineering

Imperial College of Science and Technology

Exhibition Road, London SW7 2BT.

To Shubi Mugisha

"My love is of a birth as rare
As 'tis for object strange and high:
It was begotten by despair
Upon impossibility.

Magnanimous Despair alone
Could show me so divine a thing,
Where feeble Hope could ne'er have flown
But vainly flapt its tinsel wing."

Definition of Love
Andrew Maervel (1621 - 1678)

ABSTRACT

The purpose of this work is to demonstrate that it is possible to build a CAD system which makes use of captured expert knowledge in the design of electrical machines. The greater part of the effort was expended in devising a scheme which would make the area of application of the system as wide and as general as possible.

It has to be emphasized that such a system will not replace a design expert; it is meant to play the role of a 'design assistant' by interactively advising and also guiding the designer through the design process.

The thesis starts by presenting a case for the need for knowledge based systems in the design of electrical machines and the way in which logic programming could be used to implement them. Prolog, an interactive programming language based on logic, is used to illustrate the implementation.

The presentation introduces related topics on a relatively elementary and general level. This, it is hoped, will enable also those unfamiliar with logic programming to gain an appreciation for its potential uses. Prolog's strengths and weaknesses are discussed in some detail.

An account is then given for the planning and implementation of a knowledge based CAD system, explaining the many trade-offs necessary to achieve a working program. Some background on electrical machine design is also briefly introduced.

After describing the technical features of the system the thesis identifies important benefits to be derived, and examines the future of development tools for intelligent design systems particularly the so called expert system shells.

The simplicity and use of the system is exemplified by the way it is successfully applied to various aspects of the design of specifically direct current machines.

ACKNOWLEDGEMENTS

I owe a great debt to my supervisor, Professor. E M Freeman for his enthusiasm, guidance and support in the course of my research. I am grateful to Dr. C F Bryant for many fruitful discussions and to Miss Nada El-Khatib for her useful commentaries on earlier draft of the dissertation. I am also grateful to the staff of the Department of Computing at Imperial College for assisting me in the use of Prolog.

I would like to thank the British Council for their financial support and the University of Dar Es Salaam, Tanzania, for securing me the scholarship in the first place as well as granting me a study leave of absence.

I am especially indebted to my family for their unerring support and understanding.

CONTENTS

Title page 1

Dedication 2

Abstract 3

Acknowledgements 5

Contents 6

Notation 11

CHAPTER 1 : A CASE FOR LOGIC PROGRAMMING APPLIED TO ELECTRIC MACHINE DESIGN 14

1.1 Introduction 14

1.2 The Situation 15

1.2.1 Design Activities 15

1.2.2 Design Process 17

1.3 Problems 19

1.4 Possible Solutions 22

CHAPTER 2 LOGIC PROGRAMMING 24

2.1 Introduction 24

2.2 Data Types 25

2.2.1 Syntax 25

2.2.2 Data Structures 26

2.3 Derivation 28

2.4 Runing Logic Programs 29

2.5 Reading Prolog Programs 32

2.6 Other Aspects of Prolog 32

2.6.1 Non-deterministic 32

2.6.2 Recursion 32

2.6.3 The Cut 38

2.7 Advantages of Prolog 38

CHAPTER 3 RATIONALIZATION OF A KNOWLEDGE-BASED COMPUTER DESIGN PROGRAM 39

3.1 Scope 39

3.2 Problem Characteristics 41

3.3 System Requirements 42

3.4 Approach Towards the Construction of a Design System 43

3.4.1 Representation of the Design Process 45

3.4.2 Organization of Domain Knowledge 47

3.5 Issues Addressed in Devising a Design Program 47

3.5.1 Solving a Conjunction of Interacting Goals 48

3.5.2 Looping 48

3.5.3 Discrete Parameters 49

3.5.4 Approximate (Inexact) Parameters 50

CHAPTER 4 SYSTEM ANALYSIS 51

4.1 Introduction 51

4.2 Development Environment 51

4.2.1 Hardware 51

4.2.2 Software 51

4.3 Program Structure 53

4.4 Key Ideas on Which the Program Design is Based 55

4.4.1 Control of Search 55

4.4.1.1 Best First Search Approach 55

4.4.1.2 Means-end Analysis 57

4.4.2 Domain Independent Strategy 57

4.4.3 Partial Design Approach 57

4.4.4 Alternative Designs Got Through Backtracking 59

4.4.5 Modules 69

4.4.6 Symbolic Solver 69

4.4.7 User Interaction 71

4.4.8 Planning vs. Design 74

4.5 Data Structure 74

4.6 Logic of Program 75

4.6.1 Problem Specification 76

4.6.2 Problem Solution 77

4.6.2.1 Looking for an Existing Design 77

4.6.2.2 Modifying an Existing Design 78

4.6.2.3 Designing From Scratch 78

4.7 The Inference Engine 79

4.7.1 Basic Components 79

4.7.2 Solving a Parameter Goal 80

4.7.2.1 Parameters With Numerical Values 81

CHAPTER 5 APPLICATION OF THE SYSTEM TO D.C. MOTOR DESIGN 88

5.1 Formulae, Rules (Heuristics), Curves and Tables for Armature Design 88

5.2 A Listing of Existing Designs 100

5.3 Symbols for and Relationships Between Parameters 102

5.4 Translation of d.c. Motor Design Knowledge 109

5.5 Design Problem Examples 135

5.5.1 Starting From Scratch 135

5.5.2 Modifying an Existing Design 141

5.6 A Session With the Design System 146

CHAPTER 6 OTHER ASPECTS 152

6.1 Introduction 152

6.2 Observations 152

6.3 Shells 154

6.4 Prolog vs. Fortran, etc. 155

6.5 Symbolic Computations of Algebraic Equations 158

6.6 Missing Quantities 159

6.7 Standards 160

CHAPTER 7 DISCUSSION, FUTURE WORK AND CONCLUSION 174

7.1 Discussion 174

7.2 Future Work 175

7.2.1 Explanations 176

7.2.2 Speed 177

7.2.3 Checks for Inconsistency and Completeness 177

7.2.4 Interfaces to Other Packages 178

7.2.5 Simplification of Data and Rule Input 178

7.2.6 Application to Other Areas 179

7.2.7 Robustiness 179

7.2.8 Integration in a CAE System 179

7.2.9 Redesign 180

7.3 Conclusion 181

REFERENCES 182

APPENDIX A 186

APPENDIX B 202

NOTATION

Symbols

Aa	armature surface area
Ac	cross sectional area of copper wire
As	net slot area under pole
At	net tooth area under pole
Ay	longitudinal area of yoke
Bg	apparent gap flux density
Bt	flux density in teeth
By	flux density in yoke
Cps	conductors per slot
Dc	actual wire size
De	armature external diameter
Ds	shaft diameter
Eff	efficiency
Ell	electric loading
Hc	available depth for copper with insulation
Hs	slot height
Hy	yoke height
I	full load armature current
Ksf	stacking factor
Lc	length of armature core
Mps	armature peripheral speed

N	number of turns
Pc	maximum copper loss allowed
Pi	total iron loss
Pin	input power
Po	output power
Ps	dissipated power from the armature surface
Pt	iron loss in teeth
Py	iron loss in yoke
Rpm	speed in revolutions per minute
Rps	pole enclosure, i.e., ratio pole face arc to pole pitch
Rt	ratio ampere conductors per cm to sq. cm. per amp
S	number of slots
Sel	specific electric loading
Sg	conductivity of copper
T	temperature
Tc	net coil to core clearance along slot width
Tid	total slot insulation thickness along slot depth
Tiw	total slot insulation thickness along slot width
Tow	pole pitch
V	supply voltage
Vt	total volume of teeth
Vy	total volume of yoke
Wc	available width for coil
Ws	slot width
Wt	tooth width
Z	number of conductors
_po	number of poles

_ps	specific power dissipated
_pt	teeth specific loss
_py	yoke specific loss
Dc£	diameter of copper (estimate)
Φ	flux per pole

Conventions

In-line comments (/*)

'/*' preceding text in Prolog example sessions and program listings indicates an in-line comment.

Underscore (_)

Underscore is used to join together two words or more which form part of the same entity. For example 'Parameter_value'.

<cr>

The symbol <cr> indicates an action whereby the user presses a RETURN key on the terminal.

In the examples showing program runs, user input is underlined and the program responses and prompts are shown in bold type.

Chapter 1

A CASE FOR LOGIC PROGRAMMING APPLIED TO MACHINE DESIGN

1.1 Introduction

By its very complex nature, the design of electrical machines has to be, and indeed it has always been, an iterative one. Most of it involves solving for unknown parameters given some known parameters, using equations describing the relationships between them. These processes are naturally amenable to computer processing. So, over the years, attempts have been made to introduce computer programs to aid designers.

Such earlier design tools were aimed at relieving a designer of routine and often repetitive calculations. In industrial organizations, until now, they existed as independent fragments each one used to solve a specific problem. Essentially, the user enters numbers and after the program has run, it displays values of some pre-defined parameters. On inspection, the user decides to accept the results or else try to enter another set of inputs with the hope of getting better results. Programs of this type do what has been called design analysis, as opposed to design synthesis[1].

Gradually over the years, more sophisticated tools were introduced by which specific design procedures were built in the program[2][3]. The design parameters

in this case are essentially grouped into two categories. There is one category for those parameters whose values could be entered by the user interactively. Basic parameters such as the main dimensions are the quantities usually entered by the designer, the choice of which is based on experience. The other category is a group of parameters derivable by the program. It consists of physical parameters such as reactances, and performance figures (e.g. efficiency) which in this context are also treated as parameters.

1.2 The Situation

The question is why do we need a logic programming language? In order to answer this question, we need to look at the kind of design activities that take place from which , hopefully, we could identify areas in which logic programming would be of use. The first part of this presentation is concerned with this aspect. A case is presented for logic programming.

1.2.1 Design Activities

Electrical machine design may be divided into four main areas of activity according to the levels of the difficulties involved.

(a) Designing an electrical machine similar to a previous design with minor design changes; such as changing stator diameter or stator length.

(b) Minor developments required, such as,

- special torque levels,
- unusual voltage or frequency,

- those aimed at improving the performance of existing machinery in terms of,
  : higher power factor
  : higher efficiency
  : lower noise levels
- tackling specific problems such as,
  : for thyristor controlled drive systems, efforts are geared towards overcoming problems associated with the use of semiconductor power converters.
  : commutation, in the case of commutator machines, particularly traction motors.

(c) Major developments.

- for example, raising machines to several higher ratings involving configuration changes and/or different cooling techniques. As an example, if means were found to improve the 'black band' performance, this could lead in turn to high ratings, requiring improved cooling and perhaps different lamination shapes.

(d) Development of an entirely new line,

- for special applications
- aimed at lowering production costs using whatever is available for achieving this; for example, searching for less expensive production costs – more items of a given quantity and better fabrication and machining methods.
- due to the introduction of new materials and manufacturing processes such as the introduction of robotics.

1.2.2 Design process

(a) Preliminary design involving selection of overall forms and functional requirements.

(b) Preliminary component design where components are selected and elaborated.

(c) Detailed design in which further refinements take place.

(d) Analysis, evaluation and optimisation to verify the various aspects of design.

There is still a great deal more to be done to achieve satisfactory computer design systems and it is not difficult to see why this should be so. The complexity of the design process leaves very little room for manouvre. The nature of the problem can be partially represented by a graph as shown in Fig. 1.1

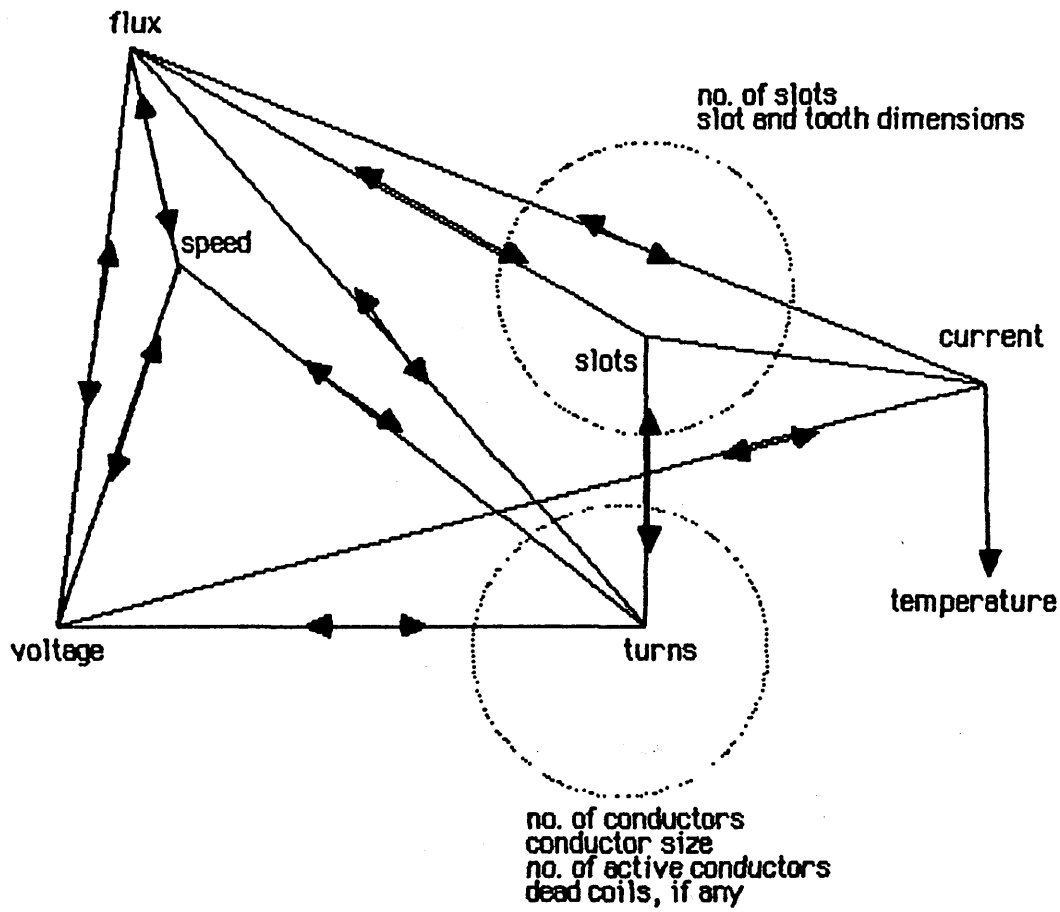


Fig. 1.1 A Simplified Network Representation of Interdependence Relationships Between Parameters

There are several basic equations describing a machine, each of them affecting to a different degree, most of the performance characteristics. Several may be implicitly or explicitly specified; such as efficiency, temperature rise and, in the case of induction motors, pull out torque and power factor, etc.

Normally an estimate on values on some parameters is made as a starting point. Adjustments are then made to meet the specifications. The number of possible ways in which the original estimate may be altered is considerable. In addition, the logical sequence of carrying out the adjustments and the order in which the various frequently conflicting performance specifications should be met, present a major problem.

1.3 Problems:

The design of electromechanical devices is a complex business; an art by itself the expertise of which can be attained after a very long period of experience. An expert designer has, among other things, rules of thumb (recorded and unrecorded) which guide him through the design. Unfortunately human experts are getting scarcer ! Something must be done to preserve the accumulated expertise. This is a particularly serious problem extending over the whole field of heavy electrical engineering where we see now most young trainees in electrical engineering opting for subjects in electronics and computers.

The problem is compounded by strict external constraints. Among them are requirements for a machine to carry a reasonable price, to match mechanical and electrical features intended for and to withstand the environment in which it is to work; coupled with an ever more exacting performance specification.

There is an additional difficulty posed by the manufacturing environment/infrastructure. Equipment owned by manufacturing firms will have been accumulated over years and consequently the sizes vary in a haphazard manner with no logical consistent steps. Hence the final design should have dimensions for a machine which, apart from meeting the performance specifications, correspond to dies and tools available in the factory.

Each of the design activities previously discussed is geared to solving a problem of an electrical or mechanical nature. However, any form of solution that may be adopted in any one of them is bound to have some effects on mechanical as well as electrical parameters. It is interesting to note that in practice a mechanical problem, for example, may be tackled with a total lack of regard for the effect the solution might have on electrical quantities and vice versa. A complete design of electrical machines exists in the form of two separate distinct design activities, one being on mechanical aspects and the other being electrical including electromagnetics. Normally the procedure is to start with the latter followed by the former such that each of these is dealt with fairly independently of the other. This cycle is repeated several times until agreement is reached. All this is done while taking into account of the thermal aspects and the coupling that occurs. The exception here is permanent magnet machine in which the magnetic properties are strongly dependent on temperature.

Not only does this make the transfer of data among engineers working on the same project complex and error prone, but may also cause unresolved conflicts in the entire design group due to insufficient coordination of activities and the lack of crucial information. The result of this will be higher development costs and longer delays. On the other hand, if the whole process, starting from the conceptual stage through to the final optimal design, could involve active participation of all parties, the above mentioned problems could be avoided.

In fact manufacturers would like to automate their design schemes to meet competition and to cope with ever increasing demands for better machines. This leads in turn to the incorporation of heuristic knowledge in computer programs; a differently and, until recently, intractable problem.

A design which meets all given performance specifications is not necessarily the best possible. A fully optimal program will calculate a number of possible designs all of which meet the given performance specification. These are stored in memory and the best design is chosen according to some specified criterion. In different cases, different criteria may apply, for example least cost, best material used, best possible efficiency (not just the specified one). As yet there is little published on electrical machine design systems which perform optimisation automatically [4], [5].

Another area which is becoming increasingly significant is systems integration. It arises from the need to integrate design with manufacture, sales and tendering.

It is important, therefore, that support and assistance be available to help with problems involving areas such as design, sales, manufacture and marketing.

1.4 Possible solutions

These problems may be substantially overcome by development of new or judicious application of known modern design aids:

(a) 'User friendly' interfaces to packages capable of sophisticated numerical analysis of electromagnetic devices are starting to emerge [6]. Some of these are now based on expert system techniques.

(b) Provision of continually updated graphic displays of preliminary and, eventually, final designs of objects.

(c) Development of a central data base into and out of which data employed by various designers is transferred.

(d) Provision for an environment where mechanical design work can be done concurrently with electrical design. This way it may be possible to reach to an optimal design in the minimum time.

(e) Provision of a design system that is able to control and monitor design tasks.

A lot of effort has been put in developing numerical techniques with success [7] aimed at aiding a designer to get more accurate answers. Efficient solid modellers are starting to emerge . The integration of different types of design data is currently a research issue in its own right.

The recent developments in logic programming languages seem to offer the potential for developing more interesting applications [8], [9] [10]. This is because, it is argued, it is easily possible to include intelligent design tasks and features which hitherto have been either difficult or impossible to incorporate into design systems written in classical languages like Fortran, C and Pascal.

Logic programming could be employed to build the so called 'knowledge based' design system to capture expert knowledge.

The use of expert systems is not new in itself. These are programs that exploit knowledge to solve problems. The early prominent systems include R1 [11] for configuring VAX computer systems, PROSPECTOR [12] for mineral exploration, DENDRAL [13] for mass-spectrum analysis and MYCIN [14] for medical diagnosis. They were written in functional languages which perform symbolic computations.

Since then, more systems have been built mainly borrowing ideas from the early systems. In the area of design, some work is being done in VLSI design [15] and IC layouts [16].

To-date, similar systems have been or are being built in Prolog, a logic programming language [17]; but to the author's knowledge, there is no working knowledge based system for design. Our objective is to attempt to build one.

Obviously this kind of 'expert system' envisaged is not one that would replace a human expert simply because it is not possible. We therefore need to know what can be captured and coded.

Chapter 2

LOGIC PROGRAMMING

2.1 Introduction

A knowledge based system derives its power from the knowledge it contains. The knowledge, of course, must be coded in the machine. And the ease of coding it to some extent depend on the language of implementation.

It is a good idea that if you are going to use these new methods you have to choose a language which makes the whole process very straight forward and very easy. While there are those who suggest that you could use languages like C and Pascal, in fact if you do so, this might take you a considerably longer time. It is generally agreed that languages which do symbolic computations are most suitable. One such language involves logic programming.

As the name implies, logic programming entails computer programming using a language whose foundations are based on some kind of logic. In this case Prolog, (an acronym for PROgramming in LOGic), will be used to implement the necessary algorithms. Consequently, the key concepts involved and definitions necessary to make this thesis self contained are introduced briefly in this chapter. There is an excellent introduction to the language by Clocksin and Mellish [18].

2.2 **Data types**

Prolog provides a uniform data structure called a **term** out of which all data and programs are constructed. A term is either an atom or a structure. The simplest data type is an **atom** which could be a **constant** or a **variable**. Constants are further subdivided into **numbers** and **names**. Thus a clause, a predicate, a goal with its arguments and the arguments themselves are terms.

A term of the form [H|T] is treated as a list whose head is H and tail is T. An empty list is [].

2.2.1 Syntax

Constants are numbers consisting of a string of digits optionally containing a decimal point. Non-numerical constants must consist of a lower case letter, alone, or followed by any combination of letters, numbers and the underline character (_).

Variables are all atoms starting with an underline and all atoms beginning with a capital letter followed by any combination of digits, lower case letters and/or underline.

2.2.2 Data structures

The basic data structure in Prolog is a **horne clause**. A set of horne clauses constitute a program. There are two types of clauses, namely, an **implication** and an **assertion.**

Implications are of the form,

B :- A1, A2, A3, ..., An ;

read as,

B is implied by A1, A2, A3, ..., An.

or as,

B is true if it can be proved that A1, ... ,An are true.

B is known as the **head** of the clause and A1, A2, etc., its **tail** or **body**. The head and tail of the clause are often referred to as **goal** and **conditions** respectively. When the A[i] (i=1,n) are missing, the clause is an assertion; a statement of fact which unconditionally holds to be true.

Generally knowledge is represented in the form of assertions, usually referred to as **facts**; and implications, also called **rules**, which describe relationships between objects. A relationship between objects is defined by a relation name called a **predicate**, with the objects that are related as its arguments.

All this may appear a bit abstract. To make things clear let us consider some illustrative examples related to aspects of electrical machine design.

Suppose we are told there is a known design of a direct current motor for which the number of parallel paths in the armature winding equals to two and that the type of winding is lap. In Prolog this may be represented as facts in the form,

I. paths(2).

II. winding(lap).

Every component of these two structures is an atom. That is to say, the set {paths,2,winding,lap} is composed of atoms of which one element is a number; the rest being atomic names. Predicate names here are "paths" and "winding".

One of the fundamental relationships in electric machinery is an equation of electromotive force (emf) across the armature windings. The parameters involved are voltage, number of turns, speed, number of poles, flux and number of paths. A list of these parameters in Prolog would be written as,

III. [voltage,turns,speed,poles,flux,paths].

Carrying on with the emf equation example further, a Prolog assertion,

IV. employ(emf, voltage(Value),find).

may be interpreted as 'to find the <Value> of the voltage, employ an emf equation'. Note here that, following the syntax described earlier, 'Value' is a variable name.

This example demonstrates the point that rules and facts are subjective. They are subject to individual interpretation. When designing large knowledge based systems different interpretations of clauses will inevitably cause problems and lead to confusion. It is important always to maintain a single interpretation of clauses associated with each predicate.

The structures in I, II and III as well as their constituents are terms.

It is always a good idea when designing d.c. machines first to consider, among other things, the use of wave type of windings. Such a choice, however, implies that the number of paths can only be equal to two. One form in which this rule could be coded in prolog would be,

IV. winding(wave) :- paths(2).

If it is known that there are two paths, then the proposition that the type of winding is wave, would be proved to be true by the rule above. I, II and IV would be referred to as clauses.

2.3 Derivation

A process of finding a solution is known as a derivation. Prolog's derivation mechanism uses a pattern matching operation known as **unification**. When a program is invoked, by entering a goal, the unifier seeks for a match of the goal with the head of a clause from the set of clauses available. On successful matching, the system proceeds similarly to solve for the conditions. When a condition fails, the system backtracks to the previous condition. In doing so, any

instantiations made as a result of matching the head are undone and a search is made for alternative match.

2.4 Runing Logic Programs

At any time, the Prolog system has a data base of clauses (which are logical theorems) from which to prove goals entered by the user. A problem is posed to the system and its solution initiated when the user runs the program by entering a goal (which is a logical formula) in response to a prompt "?" ; where "?" is a system predicate for the goal the query is about. The system executes the program by searching for a proof of the formula.

As an example, consider the following clauses:

```
emf_eqn_variable(voltage).
emf_eqn_variable(X) :- parameter(X).
parameter(speed).
parameter(turns).
parameter(flux).
```

The first clause is an assertion to be read as, 'voltage' is a variable of an emf equation. The second clause is a rule which states that some 'X' is a variable of an emf equation if that 'X' can be proved to be a parameter. An alternative interpretation is to say that every parameter is a variable of an emf equation. The last three clauses are assertions which name objects known to be parameters. Thus 'speed', 'turns' and 'flux' are parameters.

In the rest of this section illustrative examples will be presented with commentaries on any special features.

] ?emf_eqn_variable(turns). /* a query is entered

Yes /* a match was found so it displays "yes"

] ?emf_eqn_variable(poles).

No /* no match is available

] ?emf_eqn_variable(Variable). /* mention one variable of an emf equation

Variable = **voltage** /* the variable recognized first is "voltage"

More? yes /* system asks if another solution is required; user answers "yes"

Variable = **speed** /* ... so subsequent derivation finds "speed"

More? yes

Variable = **turns**

More? yes

Variable = **flux**

More? yes /* more solutions required by the user

No further solutions /* system reports it cannot find any more;

?No /* the last failure gives rise to failure of the original goal; hence the system
/* displays "No"

? /* the program terminates and the system prompts for more goal(s) input

This example serves to illustrate Prolog's non-deterministic nature. A query may give rise to several answers. A procedure which generates more than one solution to a given goal is said to be **non-deterministic**.

Suppose data (knowledge base) also includes an assertion that the voltage is also a parameter resulting in a new set of rules and facts as,

```
emf_eqn_variable(voltage).
emf_eqn_variable(X) :- parameter(X).
parameter(speed).
parameter(turns).
parameter(voltage).
parameter(flux).
```

Posing the same query as before:

```
] ?emf_eqn_variable(One_of_them).
One_of_them  = voltage
More? yes
One_of_them  = speed
More? yes
One_of_them  = turns
More? yes
One_of_them  = voltage
More? yes
One_of_them  = flux
More? yes
No further solutions
```

'voltage' appears twice; and this is undesirable. Prolog has no way of remembering whether or not something has been answered before. Nor has it the ability to tell whether a rule being entered is already implied and it is necessary to write extra rules to check against such an eventuality.

2.5 Reading Prolog Programs

Most commonly, Prolog clauses are read procedurally. They may also be written and read declaratively. Consider the concatanation of two lists by appending one list to another. One possible program which would do this is:

```
append ( [H|X], Y, [H|Z] ) :-
    append ( X, Y, Z ).
append ( [ ], Y, Y ).
```

Decleratively it states that a list with the head H and tail Z results from appending a list Y to a list with head H and tail X, if Z results from appending Y to X. Moreover, the result of appending a list Y to the empty list is Y itself.

2.6 Noteworthy aspects of Prolog

2.6.1 Non-deterministic

Prolog is non-deterministic in the sense that if you ask a question you may get several answers.

2.6.2 Recursion

The term recursion is used to describe a property whereby a procedure calls itself within itself. When used together with Prolog's built in list processing capability, simple and powerful programs can be written. It can be viewed in very simple terms as a way of effecting a 'do loop' as is usually done in Fortran except that no upper limit is set. Perhaps it is best to show what it entails by a simple example.

Assume it is required to determine the average value of air gap flux in a direct current machine using the emf equation,

$$V = 2N \frac{Rpm}{60} \frac{_pa}{P} \Phi$$

where,

V = voltage equal to 120 volts,

N = number of turns, equal to 98,

Rpm = speed in revolutions per minute, equal to 900 r.p.m.

_pa = number of paths, equal to 2 and

P = number of poles, equal to 4.

Φ = air gap flux

The problem might be to show, within Prolog, that these values are known, in order to proceed to determine the value of flux using the emf equation. This could be achieved using a recursive program as follows:

```
(1) knowns([],[]).
(2) knowns([X1(Y1)|X],[Y1|Y]) :-
(3)     known(X1(Y1)),
(4)     knowns(X,Y).
```

The procedure assumes the existence of facts of the form,

```
known(Parameter(Parameter_value))
```

which actually name the parameters that are known. For the example considered now, the corresponding set of facts are,

```
known(voltage(120)).
known(speed(900)).
known(poles(4)).
known(paths(2)).
known(turns(98)).
```

To find out if the values of a given list of parameters are known, a query,

knowns(List_of_parameters, List_of_values_of_parameters)

is posed. As a way of showing what actually happens, a listing of the trace of program execution is given. Numbers contained in the comment lines refer to line numbers appearing in the definition of predicate "knowns".

```
] ?knowns([poles(P),turns(N),voltage(V),paths(_pa),speed(Rpm)],Values).
/* first time procedure "knowns" is called: (line number 2)
***Calling GOAL
knowns([poles(P),turns(N),voltage(V),paths(_pa),speed(Rpm)],Values)
***Calling GOAL known(poles(P))
***Succeeding GOAL known (poles(4))
/* procedure "knowns" called again (at line number 4); the solution starts again at
/* line number 2
***Calling GOAL
knowns([turns(N),voltage(V),paths(_pa),speed(Rpm)],X)
***Calling GOAL known(turns(N))
***Succeeding GOAL known(turns(98))
/* Again "knowns" is called from line number 4; and the solution process starts at
```

/* line number 2;

***Calling GOAL knowns([*voltage(V)*,paths(_pa),speed(Rpm)],X)

***Calling GOAL known(*voltage(V)*)

***Succeeding GOAL known(*voltage(120)*)

***Calling GOAL knowns([*paths(_pa)*,speed(Rpm)],X)

***Calling GOAL known(*paths(_pa)*)

***Succeeding GOAL known(*paths(2)*)

***Calling GOAL knowns([*speed(Rpm)*],X)

***Calling GOAL known(*speed(Rpm)*)

***Succeeding GOAL known(*speed(900)*)

***Calling GOAL knowns([],X)

/* Now a point has been reached when the first argument of predicate "knowns" turns

/* out to be an empty list. A pattern which matches this last goal, line number 1, is

/* included for such an eventuality. It is often referred to as a terminal case.

***Succeeding GOAL knowns([],[]) /* instantiates variable X to "[]"

***Succeeding GOAL knowns([*speed(900)*],[900])

***Succeeding GOAL knowns([*paths(2)*,speed(900)],[2,900])

***Succeeding GOAL

knowns([*voltage(120)*,paths(2),speed(900)],[120,2,900])

***Succeeding GOAL

knowns([*turns(98)*,voltage(120),paths(2),speed(900)],[98,120,2,900])

/* At the end of the solution; variables will have assumed their respective values.

***Succeeding GOAL

knowns([*poles(4),turns(98),voltage(120),paths(2),speed(900)*],

[4,98,120,2,900])

/* Values of all variables that appeared in the original goal will be displayed:

Values = [4,98,120,2,900] /* List of the values of parameters

Rpm = 900

_pa = 2

V = 120

N = 98

P = 4

More? yes

/* When there is no more solution, it backtracks to the previously achieved goals;

/* undoing them as it backtracks

```
***Re-trying GOAL
knowns([poles(4),turns(98),voltage(120),paths(2),speed(900)],
                    [4,98,120,2,900]
        )
***Re-trying GOAL
knowns([turns(98),voltage(120),paths(2),speed(900)],[98,120,2,900])
***Re-trying GOAL
knowns([voltage(120),paths(2),speed(900)],[120,2,900])
***Re-trying GOAL knowns([paths(2),speed(900)],[2,900])
***Re-trying GOAL knowns([speed(900)],[900])
***Re-trying GOAL knowns([],[])
***Failing GOAL knowns([],X)
***Re-trying GOAL known(speed(900))
***Failing GOAL known(speed(Rpm))
***Failing GOAL knowns([speed(Rpm)],X)
***Re-trying GOAL known(paths(2))
```

```
***Failing GOAL known(paths(_pa))

***Failing GOAL knowns([paths(_pa),speed(Rpm)],X)

***Re-trying GOAL known(voltage(120))

***Failing GOAL known(voltage(V))

***Failing GOAL knowns([voltage(V),paths(_pa),speed(Rpm)],X)

***Re-trying GOAL known(turns(98))

***Failing GOAL known(turns(N))

***Failing  GOAL
knowns([turns(N),voltage(V),paths(_pa),speed(Rpm)],X)

***Re-trying GOAL known(poles(4))

***Failing GOAL known(poles(P))

***Failing GOAL
knowns([poles(P),turns(N),voltage(V),paths(_pa),speed(Rpm)],Valu
es)
No further solutions
] ?
```

2.6.3 The Cut

Prolog contains one useful non-logical feature known as a cut denoted by a symbol "!". When invoked as one of the conditions, it immediately succeeds and the system continues to derive other conditions. Now if failure occurs to the extent that backtracking brings the solver back to the cut, then in this situation, the use of the cut has the effect of signalling the solver to abandon the procedure containing the

alternative solution for the goal that invoked the failed procedure.

2.7 Advantages of Prolog

We shall use Prolog because,

- of its inherent feature of expressive power; closer to the way
  the expert understands knowledge in the form of rules.
- Prolog programs are easily extensible.
- it provides a relational data base for free.
- list processing is implemented in it automatically but not so
  in other languages with the exception of Lisp.
- it supports : evidential reasoning
  : explanation
  : constraint propagation
- declerative - relationships can thus be included in the program without stating how they are to be used.

These advantages will become more clear as we progress in this text.

Chapter 3

RATIONALIZATION OF A KNOWLEDGE-BASED COMPUTER DESIGN SYSTEM

3.1 Scope

Due to the expanse of the subject, it is only possible to consider a small portion of the overall design. The type of design tasks to be tackled here, are those which deal with d.c. armature design.

This may involve making minor changes to existing designs so that major dimensions are known and therefore fixed. It means sufficient design details would be available or sought for a machine whose specifications are about the same, but not quite exactly, as the one desired. The idea is to modify the existing design to meet new requirements.

Alternatively, it may be necessary to carry out the design starting from scratch. Fig. 3.1 shows a general design program lay out which is envisaged.

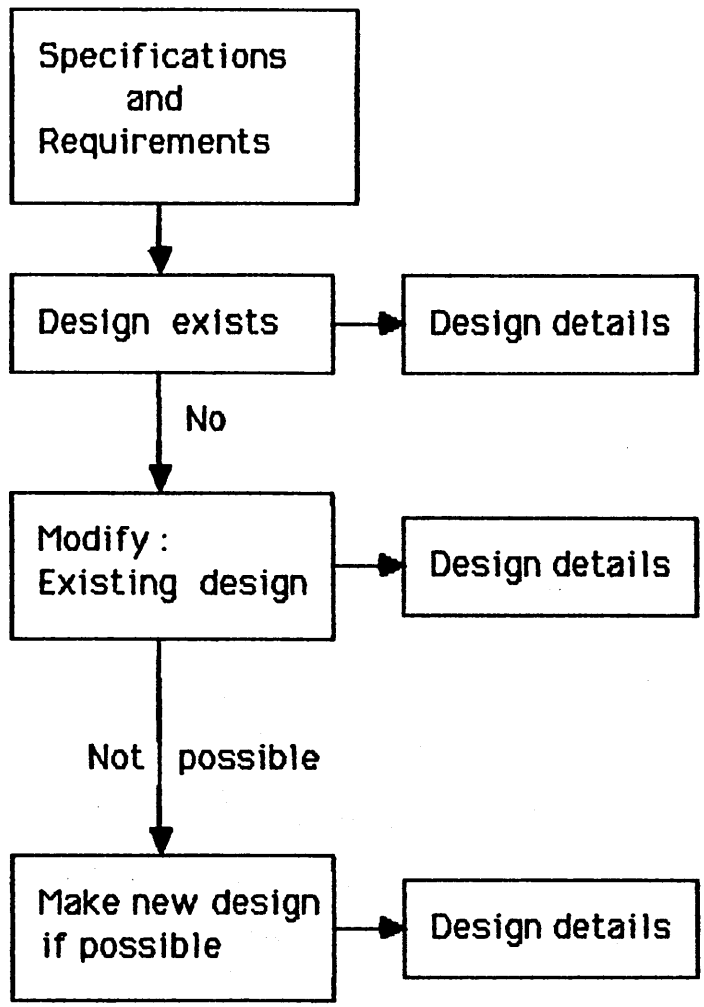


Fig. 3.1 Lay out of the task

3.2 Problem Characteristics

(a) Certain fixed quantities

There will be quantities which once specified, should not be changed by any program (e.g. number of poles, etc.) ; but in other designs this may not be the case.

(b) Hierarchical behavior

Certain parameters should not be changed unless other parameters have been changed appropriately.

(c) Interdependency relationships among parameters

Changes to the values of certain parameters must be accompanied by appropriate changes to other parameter values due to a stiff interdependence that may be existing among them. A problem arises when two or more interdependent parameters appear in a goal.

(d) Inexact derivations

Some desired quantities in a goal may not be obtainable exactly so that, for instance, instead of 50 kW may be it turns out to be 51.5 kW.; a figure acceptable from an engineering point of view.

(e) Extensive data

There will be a large amount of data. This will ofcourse increase as many designs are added.

3.3 System requirements

When considering the design of the system, there are several important requirements to be borne in mind.

As well as tailor it to accommodate the problem characteristics mentioned in section 3.1, the design system envisaged is aimed at playing the role of a design assistant at the top level to perform consistency checks at all design stages. For example, to check that a mechanical design modification does not affect the magnetic circuit design. Another example would be ensure that throughout the design appropriate standards and codes of practice are adhered to.

The designer will expect to find in the system, as part of expert knowledge, rules of thumb such as those related to unbalanced magnetic pull, noise levels, skewing effect and heating and cooling of machines.

The system must be able to accommodate fixed, commonly used dimensions and shapes such as available copper wire diameters, slot shapes and frame sizes.

The system should be capable of learning what constitutes a good or bad design as the data base expands.

Artificial intelligence techniques alone are not adequate for the overall design process. Connections must be provided to allow access to existing analytical tools. Thus facilities will be needed to interface the system to special purpose programs (written in imperative languages like Fortran) used to derive the necessary quantities or evidence it requires.

3.4 Approach Towards the Construction of a Design System

Obviously, in order to carry out design work, one must have the knowledge on how to go about it. In this context, knowledge encompasses all aspects of information, documented or undocumented, a designer has to know. Now, as we wish some of the design tasks to be performed by the computer automatically, first and foremost we need to have some means of coding knowledge into it. And for the purpose of this work knowledge will be grouped into two semi-distinct categories, namely, knowledge of the design process and domain knowledge. Domain knowledge would essentially be a data base containing all bits of information required to carry out the design process. It encompasses basic and empirical (heuristic) rules governing relationships between physical constants, machine parameters, dimensions, standards, etc. Knowledge of the design process consists of logical steps normally taken in carrying out the design. The chart below is an example of the steps that would be taken in modifying a design of an armature of a d.c. motor for a different speed and supply voltage.

Obtain Dimensions

|

|

Find Air gap flux

|

|

Find the Number of Conductors

|

|

Find the Number of Slots

|

|

Find Conductor Size

|

|

Find Current & Power

An example of a design process:

Procedure for modifying an existing design

3.4.1 Representation of the Design Process : The Knowledge Manipulator

The whole process of design essentially entails establishing a complete set of parameters. They include dimensions, number of quantities (eg. number of turns, number of slots), types of quantities (eg. lap type of winding), operational attributes (eg voltage, temperature, speed) and performance attributes (eg. efficiency). The basic links among these parameters are the mathematical descriptions of the relationships between them. Some of the mathematical relations apply globally in the sense that they are required to hold under all circumstances. Other relations are applicable to particular situations.

It is therefore prudent to consider the design as a succession of operations which establish parameters that conform to the mathematical relationships between them. However, due to the complex relations involved in electrical machine design, at some stages it is often necessary to make educated guesses to some of the parameters and later on establish their actual values.

The basic operation in design may thus be described as follows. In order to establish a parameter, either use a mathematical description, relevant to the situation, or choose some appropriate value.

The most promising formalism at this stage, useful to our particular application which would encapsulate this general idea, appears to be that using the concept of state transitions. The design process is considered as a series of actions each of which transforms one state into another starting from the initial state to the final state. At any one instant, a state is given by a set of the actual values of all design parameters. A change of state takes place if the value of one or more

parameters changes. An initial state would consist of data from a previous design, when the task is to modify an existing design. The final state will consist of new design data as a result of modifying the initial design to meet the given specifications. Actions correspond to the steps taken in carrying out the design.

Thus if state_0 is the initial state and state_n the final state, then if the actions are denoted by a symbol '-->', the design process may be represented by the graph,

state_0 --> state_1 --> state_2 --> .... --> state_n.

The graph should not be taken to mean that the actions are always taken serially. For when one carries out a design task, one goes round and round in order to get the required design. What is shown above, is a description which one would give of the successful actions which were made in arriving to the solution of the goal(s); effectively a series of operations.

As it is shown later in the text, the architecture of the design system hinges on this basic concept of state transitions effected by executing a plan of actions. One of the earlier planning system based on this approach is Warren's WARPLAN.

WARPLAN - A Planning System

We have drawn very heavily on the work by David Warren [19]. His planner program WARPLAN will be used as a starting block to our design programs. WARPLAN is essentially a knowledge manipulator (the inference engine). It uses facts and rules relevant to a problem at hand, the domain knowledge, to solve goals, which in our case are the specifications. Taking the design example, part of the domain knowledge names the parameters whose values cannot be altered. There is another part

of the domain knowledge which states, for each goal, a list of actions that can be made to achieve it; each action having a set of conditions necessary for it to take place. Also included within the domain knowledge will be indications of situations which should not be allowed to occur because they cannot happen. We will not concern ourselves too much with the details of the implementation since the programs are described in details in reference [19].

A complete program of WARPLAN is included in Appendix A

3.4.2 Organization of Domain Knowledge (Representation)

The over-all scheme draws ideas from semantic networks formalism where objects are represented as nodes and relationships between objects, effectively the knowledge, as labelled arcs terminating at nodes. The domain knowledge base is written declaratively. It includes groups of equations which form part of the basic concepts of the subject area. Some of the knowledge is obtained from text books; but the most important part comes directly from the design expert. Bits of the complex interactions can be represented and used in the inference control program.

3.5 Issues Addressed in Devising a Design Program

In this section an account is given of some of the issues that were considered in building the computer program for design tasks.

3.5.1 Solving a Conjunction of Interacting Goals

As indicated before the starting program is WARPLAN. Initially, there was really no choice; WARPLAN was available and was found a most suitable planning system. It transpired that the knowledge representation scheme employed in WARPLAN is a very good one. Its other main advantage, the method used for dealing with interactions, does not directly concern us here.

3.5.2 Looping

Frequently one encounters a set of interdependent parameters such that if one of them has to be changed, then another parameter of those remaining has to be changed in order to satisfy the relationship between them. During the earlier stages of program specification this was wrongly believed to be a form of goal interaction. For as it turns out, the problem is that of looping.

The case in point is best illustrated by an example. Suppose a specification includes voltage and speed for a rotating electric machine. In attempting to establish the voltage, the system would find out that one way of doing so, is through the use of an emf equation. Using the basic scheme described earlier, it would deduce a pre-condition that either the number of turns or speed or both be changed in order to accommodate the specified voltage. For the purpose of argument, suppose the system actually recognizes, which should be the case, that the number of turns

are to be changed. It is immediately clear this would give rise to an endless loop of derivation. For to determine the number of turns requires a knowledge of the values of voltage and speed; the parameters which are to be established in the first place.

A construct was conceived which would avoid looping as well as take into account of interdependence relationship among parameters. It was achieved by arranging the systems to make use of the specifications whenever appropriate. Specified parameters are percieved to have their values to be weakly bound to them so that references to the parameters associates their values with the specifications.

Ideally, attributes of a goal should not be used unless the goal itself has been solved. However, it is not possible to accommodate this idea for the type of problems contemplated. Imagine, as part of the specification, a machine is required for a supply voltage of 240 volts. At some point in the process of derivation, it may be necessary to know the value of the voltage. Although it may have not been established then, as a temporary measure, a value of 240 volts could be used.

Note also that in general an action made to solve for a goal will not result in the rest of the goals being solved. But it gives rise to some other definite or real state.

3.5.3 Discrete parameters

Some parameters can only assume discrete values. A table would normally be given of the possible values of such parameters. In general we can consider a situation where a group of parameters can only attain their respective values which are discrete. A set of such values would usually be associated with some kind of identification. A table of Brown & Sharp copper wire gauge provides a good example.

Round copper wire is available in definite, prescribed diameter sizes. Each diameter has a corresponding cross-sectional area, thickness of insulation, resistance per unit length for various temperature rises, and gross diameter of wire when it is covered with insulation. These constitute a group of parameters referred to above; the identification in this case being the gauge number.

We can deduce the value of a parameter from a table if we already know the value of the identifier or of another parameter in the table which in turn will have been previously derived. Alternatively, from an estimate on the value of a parameter, perhaps obtained by calculation, we assert the actual value to be that nearest to the estimated value.

3.5.4 Approximate (Inexact) Parameters

It may well be that prescribed values of some parameters cannot be established exactly. Assume, for instance, it is desired to design a motor to develop 50 kW of output power. It is not practicable by any means to arrive at a design for which the power would be exactly 50 kW. At best, designers seek to produce designs for which final values of these type of parameters closely match those prescribed within acceptable engineering limits.

Chapter 4

SYSTEM ANALYSIS

4.1 Introduction

In this chapter an account is given of the main techniques which were employed in building the design system. A description of the system itself, is also given.

4.2 Development Environment

4.2.1 Hardware

A Prime 2250 machine using Primos operating system (version 19.301k.) was used.

4.2.2 Software

The University of Salford Lisp/Prolog system; version 60, will be used. Salford Prolog is a part of Salford Lisp/Prolog system; written in Fortran77. As the name implies, Salford Lisp/Prolog system provides a common environment for Prolog and Lisp languages. One can switch between Lisp and Prolog modes by issuing an appropriate command; as a goal in Prolog, or a function call in Lisp. Lisp functions can be called within Prolog. Also, an interface is provided for with routines written in

Fortran77 (for Salford University Fortran77 compiler). For the purpose of completeness,, a list of built-in Lisp functions and Prolog predicates is included in Appendix B. Detailed explanations of their use are in reference [20].

The system allows different modes which makes it possible to choose the desired syntax. It has a provision to write a program so that one can interrupt the session with the computer and return to continue with it afterwards. It has also an on line help facility.

4.3 Program block structure

A general overview of the structure of the program is given in Fig.4.1.

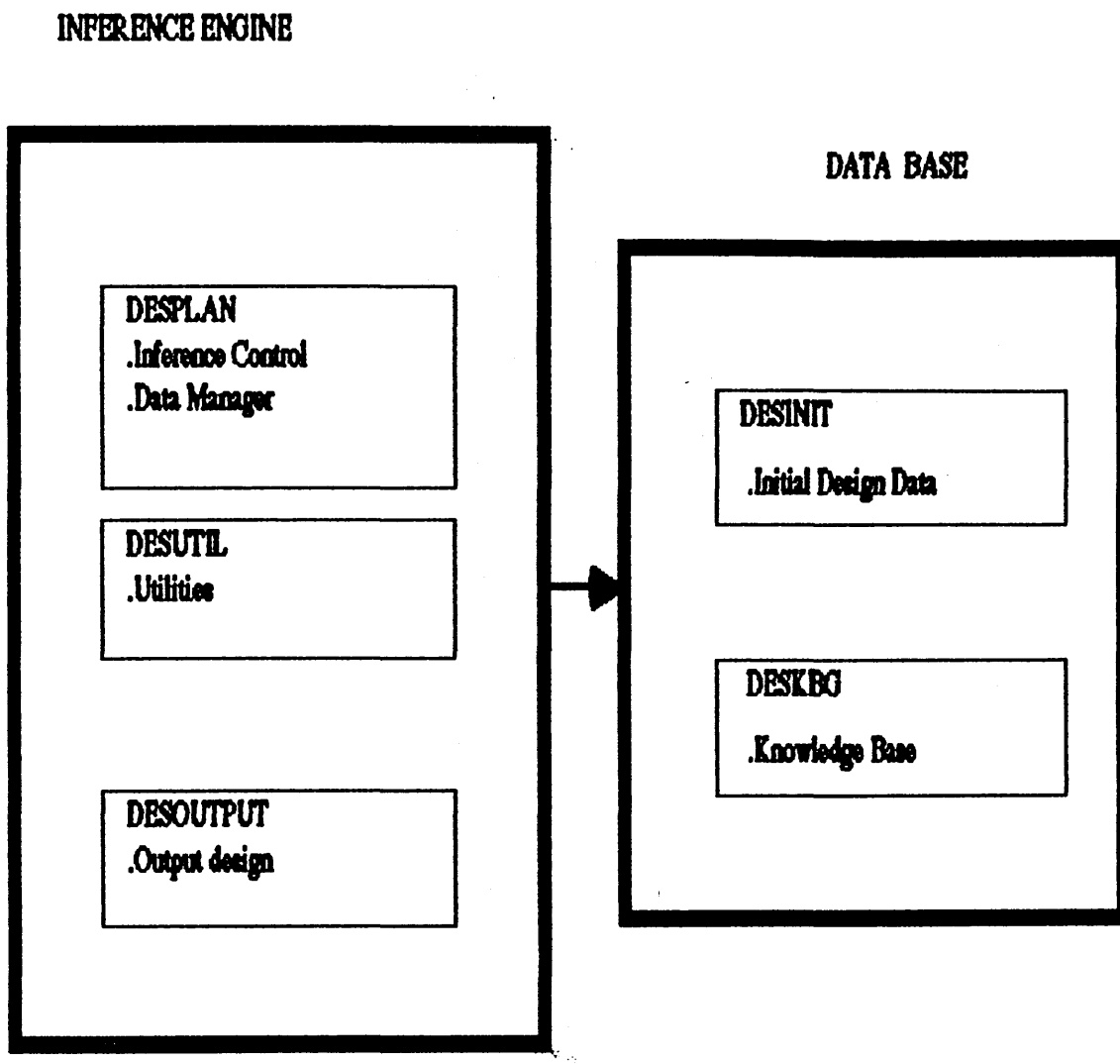


Fig. 4.1 The structure of the program.

The program is actually divided into five modules:

DESPLAN contains the control program and data base manager. DESUTIL contains definitions of predicates used in DESPLAN, DESOUTPUT and DESKGB. DESOUTPUT displays the final design parameters. DESINIT contains information on existing designs. DESKBG contains the actual knowledge for the design of electrical machines. It can be extended to include rules pertaining to such information as the category and/or subcategory of machine to be designed (eg.: d.c or a.c. motor, a generator, etc.), classification (eg.: a series or shunt d.c. machine, or in the case of induction motors, if it has a squirrel-cage or a wound rotor) and standards.

The structure of the overall program hierarchy is shown in Fig. 4.2

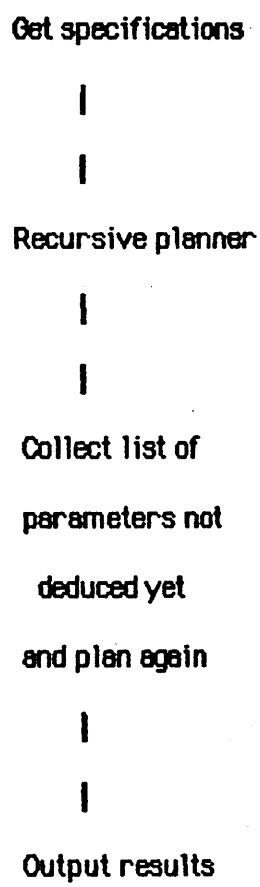


Fig. 4.2 Structure of Overall Program Hierarchy

4.4 Key Ideas on which the Program Design is Based

4.4.1 Control of Search

Prolog means working on search patterns. Generally, search would be made for all possible solutions. The set of possibilities is known as a 'search space'. Obvious fruitless searches ought to be avoided. Reduction, or pruning as is normally referred to, of the search space will be achieved by adopting a best first search approach, establishing a priority ordering of goals by setting up an hierarchy of objects.

4.4.1.1 Best First Search Approach

When there are more than one possibilities of solving a problem naturally it is always desired the best choice to be taken initially followed by the second, and so on in a descending order of preference. This is almost exclusively implemented through goal ordering, a method which takes advantage of Prolog's depth first search pattern. If there are several answers to a goal, Prolog deduces the solutions in the order the rules and assertions were entered. Thus with a careful ordering, the best solutions may be obtained first. A few examples of the uses of this technique will demonstrate its applicability.

Consider writing the following design rule: "There are basically two distinct type of armature winding for a direct current machine, namely, lap and wave type of windings. When designing an armature winding, the first choice should be the use of a wave winding (since a wave winding is the cheapest). If it is not possible to do so, a lap winding may be considered."

The knowledge as to the type of winding that may be chosen could be represented by the facts,

```
choose(winding(wave)).
choose(winding(lap)).
```

Posing a query : ?choose(winding(Type)). ,

gives the first solution as,

Type = wave ,

and when queried for more answers, it gives the second solution as,

Type = lap.

An opposite order of solution is effected by arranging the fact concerning the choice of a lap type of winding to appear first in position.

Goal ordering has been extensively used in building the main manipulating program. But its use in the knowledge data used by the main program should be avoided as much as possible. Because data is intended to increase with time, ordering it might prove difficult although not impossible.

4.4.1.2 Means-end Analysis

In this case, in order to reduce the search space, only those entities which satisfy some given goal are considered. For instance, one can employ this technique to prevent goals corresponding to user-defined and built-in programs from being treated as parameter goals and vice versa.

4.4.2 Domain independent strategy.

The thought of a system capable of handling different types of design areas is very attractive. In particular, it is desirable to be able to use the system to design various types of electrical machines. This concept could be extended further to include other design areas such as mechanical designs.

4.4.3 Partial Design Approach

The program design is such that the problem may involve either to establish a situation where a particular parameter attains a specific value, or it could be the other way round where in a given state, an unknown parameter is to be determined. Thus, each of the parameters involved in a design problem, its value could be a number or a name as the case may be, or it could be a variable.

This feature mainly arises from the desire for the system to be able to carry out what may be termed as partial or incomplete designs where, given a set of specifications and requirements, the system would come out with an incomplete set of design parameters. It would only have generated a set of parameters sufficient to solve the original goals. When more details are required the system would have then to be queried further accordingly.

As an example, suppose the problem is to modify an existing design for a machine rated 120 volts, 22 kW and 900 rpm to a new design for a machine to run at 950 revolutions per minute with a supply voltage of 220 volts. On successful completion of the design task the system will have found values of other extra parameters which were necessary to know in order to establish the specified parameters. They are flux, number of paths, number of poles, number of slots, conductors per slot and the number of turns. This set of parameters is by no means complete. So the user may wish to get more details. He may wish to know the full load current, for example. The goal to be solved would then involve the current; its unknown value entered as a variable.

This is a very useful feature. If we take an extreme case where a large knowledge base exists, it is possible for some problems the system to generate several alternative designs. Reasonable as they may be chances are that some of them may not measure to a designers liking. At some point when an undesired parameter emerges a designer should be able to tell the system not to proceed further; and instead look for an alternative design, or before that, try to find out values of one or more parameters not yet determined for the suspended design.

The idea fits in very well with normal design practice. For a preliminary design is in effect a partial design.

4.4.4 Alternative designs are got through backtracking

It was briefly mentioned earlier, that when at a certain stage Prolog fails to get a solution, it goes one step back (backtracks), to try to find another way of solving the previous goal. If it succeeds to get a solution, it tries again to solve the last failed goal. This is the feature which will be used to direct the design system look for alternative designs. The idea is, when the system successfully arrives to a design, it is made to fail which in turn forces it to look for alternative solutions (design), if any. How backtracking takes place and the way it is used to effect searches for alternative designs will be demonstrated by an example.

Consider the following rules and facts:

```
(1) solve(voltage(V),speed(Rpm),power(Po)) :-
(2)     values(voltage(V),speed(Rpm),power(Po)).
(3) known(power(22)).
(4) known(voltage(115)).
(5) maybe(power(25),voltage(100)).
(6) values(voltage(220),speed(1200),power(20)).

(7) values(voltage(110),speed(1500),power(Po)) :-
(8)     maybe(power(Po),voltage(100)).
```

```
(9) values(voltage(V),speed(Rpm),power(Po)) :-
(10)    known(voltage(V)),
(11)    known(power(Po)),
(12)    Rpm = 900.
```

The rules and fact above have been entered without paying attention to the order they appear, intentionally. They constitute an and/or tree structure.

Suppose, in addition, the following clauses are used to generate and display solutions related to the above rules and facts:

```
(13) give_a_design( ) :-
(14)    solve(voltage(V),speed(Rpm),power(Po)),
(15)    disp(V,Rpm,Po),
(16)    write(Do you another or alternative design ? (y/n) ),
(17)    read(Answer,LINE),
(18)    Answer = n.
```

```
(19) disp(V,Rpm,Po) :-
        write(The voltage equals to ),
        write(V),
        write( volts),
        nl,
        write(The speed  equals to ),
        write(Rpm),
        write( rpm),
        nl,
        write(The power  equals to ),
        write(Po),
        write( kW),
        nl.
```

Entering a goal "give_a_design" will result in displaying values of voltage, speed and power followed by a question whether more solutions are needed. Answering "y" will cause another solution to be sought and displayed if it exists. This is because when "y" is entered, the last condition " Answer = n" for predicate "give_a_design" fails since the variable "Answer" will have been instantiated to "y" . As a result, the program backtracks. It cannot resatisfy "write" as it only succeeds once, so it backtracks further back and tries to resatisfy "solve". A listing of the record of program run conveys the idea best.

] ?give_a_design.

The voltage equals to 220 volts

The speed equals to 1200 rpm

The power equals to 20 kW

Do you another or alternative design ? (y/n) no /* Answer = n, no

/* backtrack

Yes

] ?give_a_design.

The voltage equals to 220 volts

The speed equals to 1200 rpm

The power equals to 20 kW

Do you another or alternative design ? (y/n) yes /* Answer = y , so

/* backtrack

The voltage equals to 110 volts

The speed equals to 1500 rpm

The power equals to 25 kW

Do you another or alternative design ? (y/n) yes /* Answer = y , so

/* backtrack

The voltage equals to 115 volts

The speed equals to 900 rpm

The power equals to 22 kW

Do you another or alternative design ? (y/n) yes /* Answer = y, so

/* backtrack

No /* oops! no other ways of arriving to a solution

In order to show exactly how the solution proceeds, a trace of how it executes, is given below. A number preceeded by a semicolon corresponds a number in brackets inserted in the definition of predicates associated with the facts and rules for this example.

```
] ?give_a_design.
***Calling GOAL give_a_design( )  :13
***Calling GOAL solve(voltage(X),speed(Y),power(Z)) :14
***Calling GOAL values(voltage(X),speed(Y),power(Z)) :2
***Succeeding GOAL values(voltage(220),speed(1200),power(20)) :6
***Succeeding GOAL solve(voltage(220),speed(1200),power(20)) :2
***Calling GOAL disp(220,1200,20) :15
***Calling GOAL write(The voltage equals to )
The voltage equals to
***Succeeding GOAL write(The voltage equals to )
***Calling GOAL write(220)
220       /* writes "220"  in achieving the goal "write(22)"
***Succeeding GOAL write(220)
***Calling GOAL write(  volts)
 volts
***Succeeding GOAL write(  volts)
***Calling GOAL nl( )
                          /* here a new line is generated by goal "nl( )"
***Succeeding GOAL nl( )
***Calling GOAL write(The speed  equals to )
```

```
The speed   equals to
***Succeeding GOAL write(The speed   equals to )
***Calling GOAL write(1200)
1200
***Succeeding GOAL write(1200)
***Calling GOAL write( rpm)
 rpm
***Succeeding GOAL write( rpm)
***Calling GOAL nl()

***Succeeding GOAL nl()
***Calling GOAL write(The power   equals to )
The power   equals to
***Succeeding GOAL write(The power   equals to )
***Calling GOAL write(20)
20
***Succeeding GOAL write(20)
***Calling GOAL write( kW)
 kW
***Succeeding GOAL write( kW)
***Calling GOAL nl()

***Succeeding GOAL nl()
***Succeeding GOAL disp(220,1200,20)  :15
***Calling GOAL write(Do you another or alternative design ? (y/n) )
:16
Do you another or alternative design ? (y/n)
```

```
***Succeeding GOAL write(Do you another or alternative design ?
(y/n) )
***Calling GOAL read(X,LINE)
no   /*  Answer is no more required
***Succeeding GOAL read(n,LINE)
***Calling GOAL =(n,n)  :18    /* variable "Answer" has been
                               /* intentiated to "n"
***Succeeding GOAL =(n,n)      /* success, so . . .
***Succeeding GOAL give_a_design()  /* the original goal succeeds
Yes
] ?give_a_design.
***Calling GOAL give_a_design()   :13
***Calling GOAL solve(voltage(X),speed(Y),power(Z))   :14
***Calling GOAL values(voltage(X),speed(Y),power(Z))  :2
***Succeeding  GOAL  values(voltage(220),speed(1200),power(20))
:6
***Succeeding GOAL solve(voltage(220),speed(1200),power(20))
***Calling GOAL disp(220,1200,20)   :15 /* has not been traced to avoid a
                                        /* lot of listing. The trace above
***Succeeding GOAL disp(220,1200,20)    /* is sufficient

***Calling GOAL write(Do you another or alternative design ? (y/n) )
:16
Do you another or alternative design ? (y/n)
***Succeeding GOAL write(Do you another or alternative design ?
(y/n) )
```

```
***Calling GOAL read(X,LINE) :17
yes
***Succeeding GOAL read(y,LINE)
***Calling GOAL =(y,n) :18   /* "Answer" instantiated to "y"
***Failing GOAL =(y,n)     /* failure - y is not equal to n; backtrack to "disp"!
***Re-trying GOAL disp(220,1200,20) /* "disp" calls "write" and "nl" . . .
***Failing GOAL disp(220,1200,20)    /* which cannot be resatisified
                                        /* so backtrack to "solve" . . .
***Re-trying GOAL solve(voltage(220),speed(1200),power(20)):14
/* . . . and in turn backtrack to "values" . Finds another way - :7
***Re-trying GOAL values(voltage(220),speed(1200),power(20))
***Calling GOAL maybe(power(X),voltage(100)) :8
***Succeeding GOAL maybe(power(25),voltage(100))
***Succeeding GOAL values(voltage(110),speed(1500),power(25))
          /* success!
***Succeeding GOAL solve(voltage(110),speed(1500),power(25))
          /* success!
***Calling GOAL disp(110,1500,25)
***Succeeding GOAL disp(110,1500,25) /* display results

/*  Similarly, the third design is obtained:

***Calling GOAL write(Do you another or alternative design ? (y/n) )
Do you another or alternative design ? (y/n)
***Succeeding GOAL write(Do you another or alternative design ?
(y/n) )
```

```
***Calling GOAL read(X,LINE)
yes
***Succeeding GOAL read(y,LINE)
***Calling GOAL =(y,n)
***Failing GOAL =(y,n)
***Re-trying GOAL disp(110,1500,25)
***Failing GOAL disp(110,1500,25)
***Re-trying GOAL solve(voltage(110),speed(1500),power(25))
***Re-trying GOAL values(voltage(110),speed(1500),power(25))
***Re-trying GOAL maybe(power(25),voltage(100))
***Failing GOAL maybe(power(X),voltage(100))
***Calling GOAL known(voltage(X))
***Succeeding GOAL known(voltage(115))
***Calling GOAL known(power(X))
***Succeeding GOAL known(power(22))
***Calling GOAL =(X,900)
***Succeeding GOAL =(900,900)
***Succeeding GOAL values(voltage(115),speed(900),power(22))
***Succeeding GOAL solve(voltage(115),speed(900),power(22))
***Calling GOAL disp(115,900,22)
***Succeeding GOAL disp(115,900,22)
***Calling GOAL write(Do you another or alternative design ? (y/n) )
Do you another or alternative design ? (y/n)
***Succeeding GOAL write(Do you another or alternative design ?
(y/n) )
***Calling GOAL read(X,LINE)
```

yes

```
***Succeeding GOAL read(y,LINE)
***Calling GOAL =(y,n)
***Failing GOAL =(y,n)
***Re-trying GOAL disp(115,900,22)
***Failing GOAL disp(115,900,22)
```

/* At this stage all possible solutions have been obtained. The system will backtrack
/* back to the original goal "give_a_design" and consequently fail:

```
***Re-trying GOAL solve(voltage(115),speed(900),power(22))
***Re-trying GOAL values(voltage(115),speed(900),power(22))
***Re-trying GOAL known(power(22))
***Failing GOAL known(power(X))
***Re-trying GOAL known(voltage(115))
***Failing GOAL known(voltage(X))
***Failing GOAL values(voltage(X),speed(Y),power(Z))
***Failing GOAL solve(voltage(X),speed(Y),power(Z))
***Failing GOAL give_a_design()
No
```

Notice that in the last part because it failed to get more solution, it backtracked to the original goal and in the process undoing all the previous intantiations. This feature is used frequently in the main program. But it can also be a problem if one wants to remember some aspects prior to the failure.

4.4.5 Modules:

The idea of building a system based on modules is an attractive one. For it is then possible to build a general design program for electrical machines and other supplementary programs each of which could be picked up by the main program depending on the nature of the problem at hand such as 'black band' calculations.

4.4.6 Symbolic solver:

A symbolic solver has two main advantages over routines written in algorithmic languages. Firstly, multiple terms can be computed so that given an equation involving several parameters, a solution for any parameter can be obtained in terms of the other independent parameters. With Fortran77, Pascal or C language, a different subroutine would be required for each parameter to be solved. Secondly, if values of some of the independent parameters are not known, the solver will yield a solution for a parameter as a function of some unknown quantities; something which would yield an error in algorithmic languages. Thirdly, the solver is not restricted to any particular equation. Hence is the elimination of the requirement to write separate routines associated with each equation applicable in the design. Fourthly, it is possible, elegantly to introduce limits within which equations apply. A demonstration of the potential of a symbolic solver follows below.

The goal,

solveqn(Equation, Variable, Solution).

is posed, to get a <Solution> by solving an <Equation> with respect to a given <Variable>. A listing of a sample of program runs illustrate the point:

.

] ?solveqn(98 = 12*x,x,Y). /* Equation: 98 = 12*x

/* the variable is "x" (must be lower case letter)

Y = 8.16666666667 /* the solution is "Y" (must be upper case)

More? yes

No further solutions

/* the variable whose the solution is sought can appear on the left hand.

] ?solveqn(12*x = 98,x,Y). /* solve for X: 12*X = 98

Y = 8.16666666667

More? yes

No further solutions

] ?solveqn(65 = x*5/2+80,x,Y). /* solve for X: 65 = X*5/2+80

Y = -6

More? yes

No further solutions

] ?solveqn(25 = x+x,x,Y).

No /* this case has not been included so it fails

] ?solveqn(25 = x+ 12.5,x,Y).

Y = 12.5

More? yes

No further solutions

It is seen the solver can manage very simple algebraic equations simply because that is what it was intended for. It can be extended to handle complex algebraic equations.

4.4.7 User Interactions

During the design process a point may be reached when the system fails to establish the value of a parameter. In order to continue with the design, it requests the user to volunteer an answer. Before giving an answer, the user may wish first to get some information. For example, of what may have been achieved. After a series of interactions, he would then go back to the original query and answer it accordingly.

Putting it in other words, the program is required to be able to suspend its task, when the need arises, and continue with the task sometime later. This is achieved by a combination of predicates 'ask', 'h' and 'r'.

```
ask(X,Pre_read_goal_list) :-
kkc(Pre_read_goal_list),
read1(X).

read1(In) :-
   read(In,'LINE'),
   result(In).
```

```
result(h) :-
   !,
   result1(h).

r :-
    %('COMMAND_LOOP'()).

result(In).

result(_) :-
    current_goal(G,2),
   G =.. [Pred|_],
   abort(Pred).

result1(h) :-
    %('COMMAND_LOOP'('a)),
    write('Now returning back :'),nl,
    fail.

kkc(List) :-
   apply(Goal,Goal,List).

kkc(List) :-
   kkc(List).
```

As an illustration consider a clause below whose predicate is "a"

```
a :-
  ask(X,[write('Please give values :')]),
  c(X).

c(2).
```

An excerpt of the interaction illustrates the concept of suspending a task:

```
] ?a.

Please give values :h.    /* type "h" command to suspend the program
] ?var(X)       /* having suspended the program, we can do anything
X  = X          /* eg: here the query is to find out if "X" is a variable; which it is.
More? yes
No further solutions
] ?r.    /* "r" causes a return to the previous program run.
Yes
Now returning back:
Please give values :2.        /* success
Yes
```

4.4.8 Planning vs. Design

In many ways planning is similar to design with a different areas of emphasis. A planner gives, as output, an enumeration of the sequence of actions that ought to be carried out to achieve a set of given goals. In contrast, the outcome of a design process is expected to be a listing of the design details. That is to say, the interest is not so much in knowing the actions, but more so in getting all the facts which hold in the final state resulting from the actions taken.

4.5 Data Structures

It can't be emphasized too strongly that the design of the data structures is possibly the most important activity of all. It is important to appreciate that the nature of the problem to some extent influences the form of data structures that can be used in the representation of knowledge. During program development, it was often necessary to make a careful balanced judgement on the choice of the form of data structures to adopt. Of necessity, the procedure has been an iterative one.

In the case of the actual design parameters it was found most suitable to represent them by unary predicate names whose arguments correspond to parameter values. Thus an indication that the voltage is 230 volts is represented by a term voltage(230).

The main program uses various forms of data to represent knowledge. They are introduced in the relevant sections. One which should be mentioned here, is "links" relation name. It attains an assertion of the form:

links(Relation_name,Related_parameters,Relationship).

The interpretation is: some <Relation_name> links a set of <Related_parameters> according to a given <Relationship>. The relation name can be a name given to an equation, a curve or a table. The relationship would then be the equation, curve or table.

While the data structures described in here perform reasonably well, it is not claimed to be the best possible. There is much work to be done here. Indeed, any future work could well start with the aim of improving the data structures.

4.6 Logic of Program

The top level interaction cycle is of the following form:

-display of options : initiation/command level

-task/problem invocation : goals & queries or desired parameters

-Problem solution : derivation of Prolog goals/ generation of a plan of actions

-output of results

It effectively works on two levels which are (i) problem specification and statement of requirements and (ii) problem solution.

4.6.1 Problem Specification

This is the point of starting at which the designer specifies the main performance criteria of the machine required. The user enters the specifications in any order. For any design task, however, there is always a defined order of evaluating the parameters. Thus on completion of the specifications, the parameters are sorted according to the design expert's priority rating. The program predicate "insort" puts the goals in the correct order before passing them to the planner. It uses assertions in the data base which actually name the parameters together with their relative priorities as an extra argument.

The data base will thus contain assertions of the form,

parameter(X(Y),Z).

where, X is the parameter name, Y is the parameter value of X and Z is the priority number. The priority of X decreases as Z increases.

Imagine there are three facts for the relation "parameter" , namely,

parameter(1 ,voltage(Y)).
parameter(100,current(A)).
parameter(130,power(W)).

a query : insort(power(P)&current(C)&voltage(110) , X). ,
gives **X = voltage(110)&current(C)&power(P)** as the answer.

This priority rating can be altered by the design expert at any time.

4.6.2 Problem Solution

Although problem solution commonly refers to finding a set of design parameters which conform to given specifications and requirements, it is also taken to encompass ordinary Prolog derivations.

4.6.2.1 Looking for an existing design

Normally in an industrial setting, there are many existing designs, some dating back a considerable time. Thus whenever specifications are entered for the desired machine, the first step is to find out if a design exists which meets operation and performance needs. It is based on the rule: from a list of available designs, search if for the conjunction of specified parameters, their values are acceptable as possible solutions to the specifications. The value of a parameter from an existing design is a possible solution with respect to a specified parameter if the two values are the same. This is so in the case of parameters whose values only attain symbolic names. For parameters whose values are numbers, it is unrealistic to expect to match a value of a parameter of an existing design to some specification, because there will always be deviations between numbers. Accordingly, the test allows for deviations of numbers within defined limits. If as part of the specification the output power of a motor is required to be 30 kilo watts, then an existing design is sought where the power is, say between 30 and 33 kilo watts. An assertion of the form,

kind(Parameter_name,Min,Max)

is used to establish the limits of the value of a <Parameter_name> to be between <Min> and <Max> expressed as percentage of specified value.

4.6.2.2 Modifying an existing design

If none of the existing designs matches the specifications, an immediate thing to do is to find out if among the existing designs there is at least one suitable for modification so as to meet the requirements. A suitable design will be identified according to specified rules. In our case it is based on values of output coefficients. Upon identification, the design is then modified appropriately. The modification proceeds in a similar manner as is done in the case of a new design which is described in details below.

4.6.2.3 Designing from Scratch

When neither of the above attempts produce positive results, then the design has to be made starting from scratch in two steps. The first involves making a preliminary design. This is when values of the main parameters are approximately determined. The second step is where the actual values are established.

4.7 The Inference Engine

A design is complete when all the design parameters are established. This is achieved by application of the appropriate general and proprietary formulae and rules of thumb. All types of rules will be collectively referred to as **design rules**. At any stage of the design the system has to determine selectively which design rule to employ. And the program which carries out this task is known as an **inference engine**.

For the system described here, as far as the inference engine is concerned, there is no distinction in the manner in which it handles rules for a design involving modification of a known design as opposed to designing from scratch. The difference lies in the rules themselves. Thus in the following analysis of the program flow, the operations which will be described are to be taken to be applicable to these two categories of design. Exceptional cases will be clearly indicated.

4.7.1 Basic Components

Design is viewed as a network composed of nodes and links between nodes. There are goals and actions. The goals are achieved through actions. An action is said to be successful on fulfilling the pre-conditions attached to it and proving that the value of the parameter goal is acceptable. And a derived parameter is always taken to be acceptable unless indicated otherwise.

As was mentioned earlier, it is based on a planning approach. The entry point of a planner is a call to the goal,

plans(Set_of_specified_goals, start).

This sets out to solve a <Set_of_specified_goals>, one at a time recursively. The goal,

solve(Agoal,_,_,_)

initiates the solution process.

It may well be that the current goal which is to be solved was in fact derived earlier while in the process of solving previous goals. Or it may be given explicitly that a particular parameter must unconditionally always attain a specific value. When this is so, then the solution is construed to have been obtained. Otherwise a search is made for a possible action which would lead to the solution of a goal. Evaluation of a parameter involves initially fulfilling the conditions which must be satisfied, determining its value and checking if it is acceptable.

4.7.2 Solving a Parameter Goal

There are two kinds of queries that are possible with regard to parameters. One is where it is required to establish a state in which a specified value of a parameter holds to be true, and the other one is for a case when the value of a parameter is unknown and so needs to be determined.

4.7.2.1 Parameters with numerical values

If the goal is to establish or find the numerical value of a parameter, often it means looking for and using an appropriate equation into which the parameter in question is one of the independent variable.

Inevitably, the operations depend on there being equations each of which describes the relationship between sets of parameters. The objective is then, for any given equation, to get values of the corresponding parameters which make the equation to hold. In this regard, various actions have been identified and described below.

The approach used in the definition of the actions is basically the same. Differences come in to cover the special circumtances associated with each case. On account of this, a thorough description of the first action will be given. When necessary, other actions will be treated briefly only for the purpose of pointing out their peculiarities.

(a) Action : change

In order to establish a parameter, look for the applicable relationship and satisfy the conditions necessary for the action to succeed.

Essentially, it models the way one goes about to establish a specified value of a parameter. Because the value of a parameter goal is specified, it means the value of some other parameter in the equation has to be changed accordingly to accommodate the specification.

To start with, a possible equation to use is identified; making sure it has not been used before otherwise it gives rise to a circular definition. A parameter that can be changed is then sought. It must not be a specified parameter. Other parameters affected by the changes are identified.

The next step is to determine and fulfil the preconditions that must be met before the action takes place.

First pick up the details about the relation name which was identified in the previous stage. They are, a conjunction of related parameters and a description of the relationship between the parameters. In this case the relationship is an equation. So if the relation name is "emf" , the corresponding relationship is an emf equation.

What follows after this is to collect together parameters which are not known, combine them with a conjunction of parameters affected by the change such that before the relevant equation is solved, the unknown parameters are first determined; and after it is solved, the affected parameters, if any, are updated to accommodate the new situation.

Action : compute

In this case, the value of a parameter for which the goal is sought, is unknown; it is to be determined. A straight forward action to do so is to look for an appropriate equation and compute for the unknown value of the parameter. This action can succeed only if the values of all parameters in the equation being used, other than the goal parameter, are known.

(b) Action : find

This action also simulates the manner in which an unknown parameter can be found. However, it relaxes restrictions imposed by the 'compute' clause. Here, the values of the rest of the parameters, excluding the goal parameter, need not be known in advance. Instead allowance is given for possibilities to use other means which eventually lead to a knowledge of the values. A numerical value of a parameter defined by an equation in terms of other parameters, may be determined by first 'find'ing values of the other parameters; yet unknown. The value of the desired parameter is then obtained by solving the resulting algebraic equation. A schematic diagram of what this entails is shown in Fig. 4.3

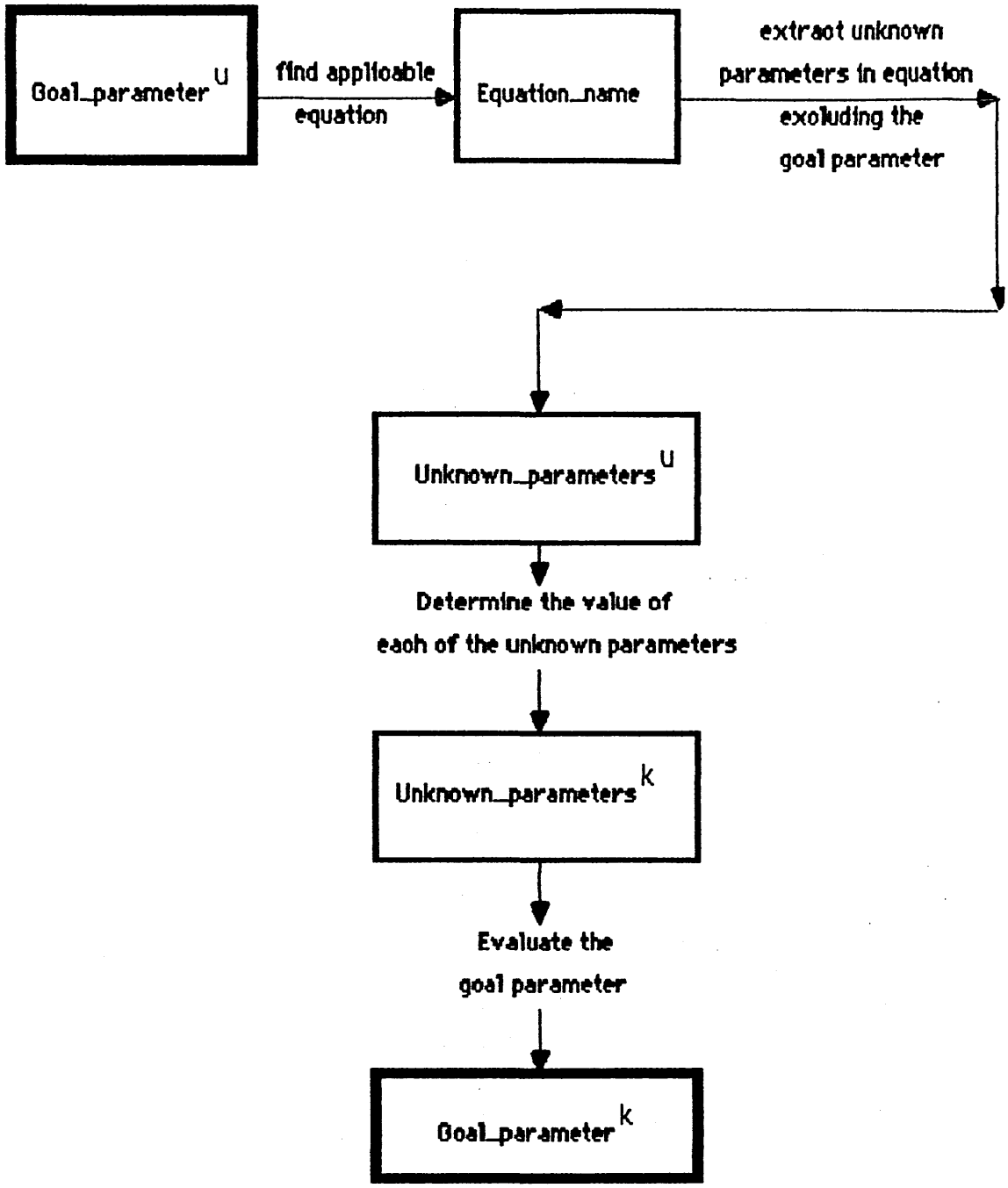


Fig. 4.3 Schematic diagram for action "find"

Superscripts "u" and "k" denote unknown and known quantities respectively.

Although a parameter may appear in an equation, it does not necessarily always mean it can be obtained using any of the actions above. For instance, to determine the number of poles one does not use an emf equation. Similarly, not every parameter in an equation can be changed to accommodate one which is specified. If, for example, a goal involves establishing a specified voltage using an emf equation, the number of paths would never be considered a candidate parameter to be changed in order to make the equation to hold.

For these reasons, it is necessary for any given equation, to indicate actions which are possible for each parameter. This has been done by including assertions of the form,

employ(Relation_name, Parameter_name(Parameter_value), Action_name).

read as: the <Parameter_value> for a <Parameter_name> may be determined by employing a relationship defined by a <Relation_name> in doing an action associated with <Action_name>.

Hence, an option that the voltage may be computed using an emf equation would be written as,

employ(emf, voltage(V), compute).

This approach will be used to include all types of actions. The resulting benefit for its use is a considerable reduction of the search space. Because with these types of facts present, it is possible to arrange for the design system to avoid having to try solution methods that are known never to succeed.

An assertion employ(<equation_name>,<parameter>,find) means an equation identified by an <equation_name> may be used to 'find' the value of a <parameter>. This assertion may be included in the knowledge base cautiously, that is, on condition it is known beyond doubt that the parameter in the equation would not lead to a loop in the process of deducing its value. Thus the best way to determine whether the inclusion of such an assertion is appropriate is to (check whether) determine that in doing so the potential for looping does not occur.

Other actions

Among the other actions that are possible are using tables and graphs, or simply making a choice depending on certain circumstances. Making a choice also applies to non-numeric parameter values.

In order to make the program faster when trying to solve for a parameter by computing the appropriate equation we could simply arrange so that some of the conditional parameters as well as being required to be 'known', they may be optionally be chosen. However the need to be consistent in keeping a record of every action made, the possibility of choosing a parameter should be allowed for by including such parameter in the conjunction of conditions to be fulfilled for the 'compute' action to succeed. Care has to be taken so that, in turn, the conditions attached to the selection of a parameter do not lead to a loop.

The idea of a known parameter needs clarification. The value of a parameter is said to be known if either it is explicitly known in which case possibly it has been solved as a goal or stated it must always be so or specified - in a goal.

If it happens that there are no suitable actions for solving a goal parameter, as a last resort it asks the user for possible values. When this happens, it may mean the value of the parameter is either indeterminate or, if it was determined, it was found to be unacceptable or inconsistent in relation with the set of parameters found then.

Chapter 5

APPLICATION OF THE SYSTEM TO D.C. MOTOR DESIGN

While the system can be applied to a wide range of design problems, here we concentrate on one particular example of design of an armature of a d.c. machine. In this chapter we demonstrate how the system is used to arrive at the design. We will go about this by first enumerating formulae, graphs, tables and rules pertaining to d.c. motor design. This knowledge will then be translated into the system and subsequently used to carry out various design tasks. For each set of specifications an indication of how the designer proceeds with the design task will be given. It will be followed by a listing of the record of the session on how the user interacts with the system while conducting the design.

5.1 FORMULAE, RULES (& HEURISTICS), CURVES AND TABLES FOR ARMATURE DESIGN

Figures and rules used here have been chosen for the purpose of demonstration. It should be made clear too, that one could easily have rules which have different values, with different sets of conditions in, from those indicated.

Most of the rules have been obtained from various sources including ref. [21], from personal experience and from discussions with people who have done design work previously.

When it is necessary to design a d.c. motor from scratch, curves such as those shown in Fig. 5.1, Fig. 5.2 and Fig. 5.3 are used to determine approximate values of the main dimensions. It should be appreciated that all loading values quoted are a function of time. Furthermore, some of the ranges given are peculiar to a particular set of conditions.

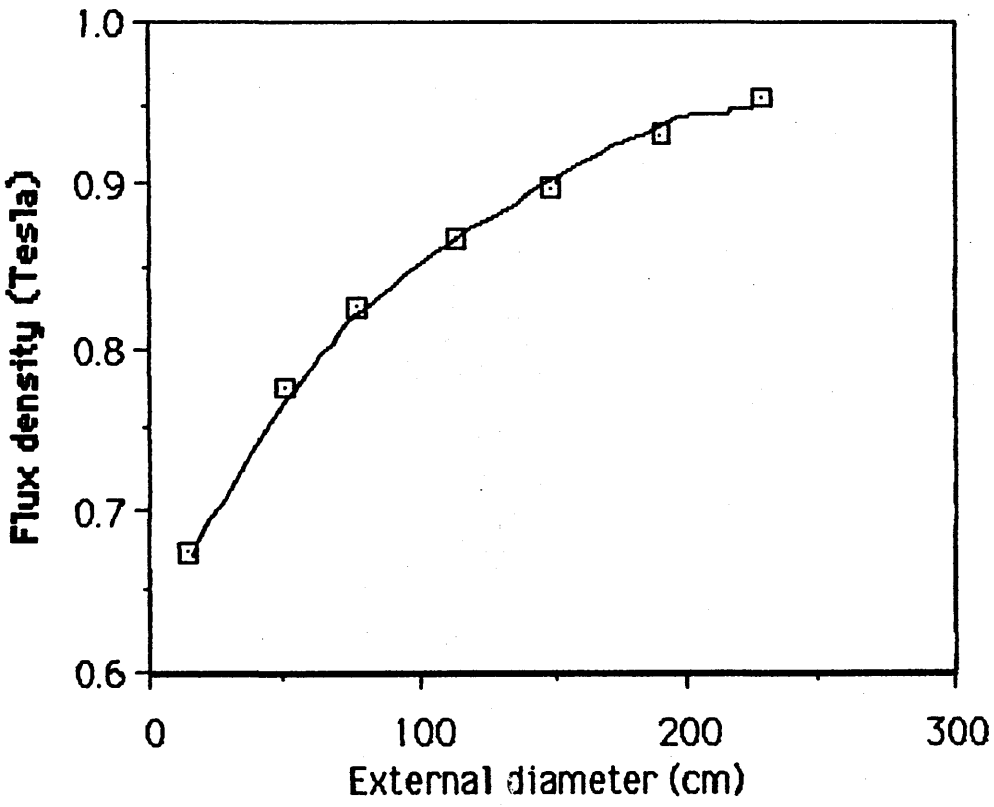


Fig. 5.1 Curve giving relation between apparent gap density and armature diameter

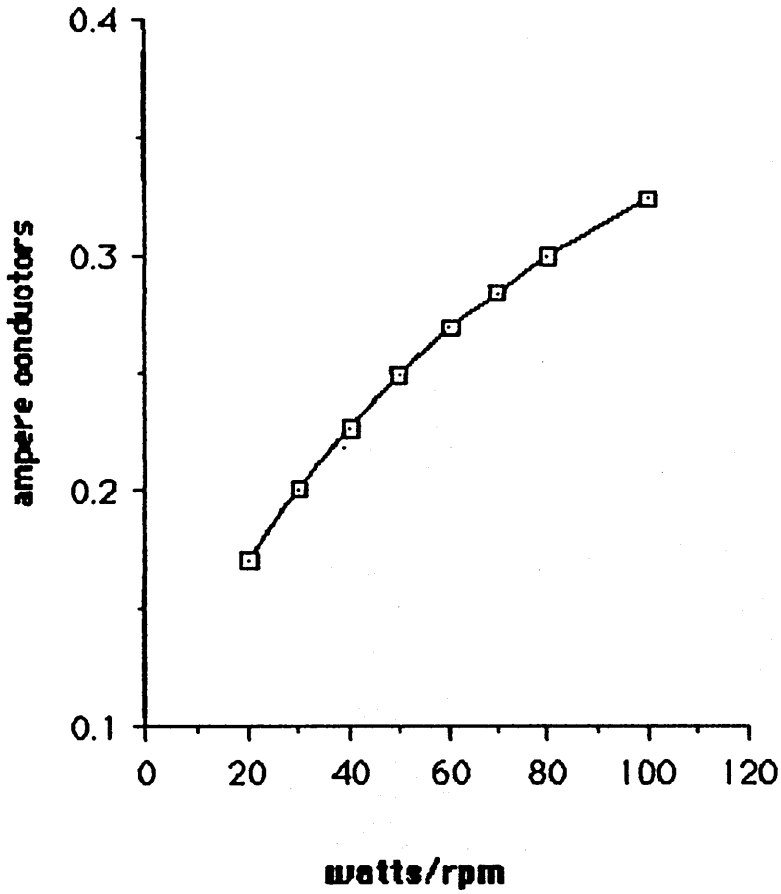


Fig. 5.2 Curve giving relation between electric loading and power per unit speed

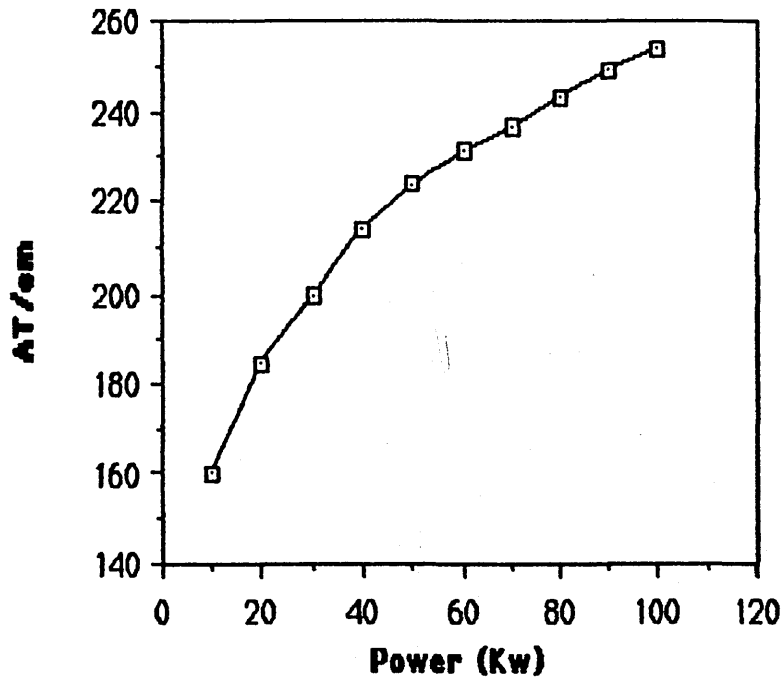


Fig. 5.3 Curve giving relation between specific electric loading and output power

The above curves are used to obtain approximate length and external diameter of armature core using the equations:

$$De = \frac{Hl}{\pi Sel} \qquad E1$$

$$Lc = \frac{Pin}{Rpm} \frac{60.8}{Bg\ Rps\ Sel\ De^2} \qquad E2$$

emf equation:

$$V = Z\Phi \frac{Rpm}{60} x \frac{_po}{_pa} \qquad E3$$

where,

V = induced emf in the armature windings, assumed to be equal to terminal voltage so that voltage drop across armature windings is neglected.

Z = number of armature conductors

Φ = air gap flux per pole (Wb)

Rpm = armature speed in revolutions per minute

_po = number of poles

_pa = number of paths

Apparent air gap flux per pole, Φ:

$$\Phi = Bg\ Rps\ Tow\ Lc \qquad E4$$

Turns:

- for a series winding,

$$N = \frac{k_po}{2} \pm 1 \qquad E5$$

where,

N = number of turns

k = an integer

Slots :

- for a series winding,

$$S = \frac{k_po}{2} \pm 1 \qquad E6$$

where,

S = number of slots

- for a multiple winding with equalisers,

$$S = \frac{k_po}{2} \qquad E6_b$$

The number of slots should be such that the slot pitch lies between 2 and 4 cm. In addition, the number of slots can have values from a table of standard punches.

. . . . H1

Armature ampere-turns per pole, AT_p :

$$AT_p = \frac{ZI_a}{2_po} < 7500 \qquad \text{H2}$$

The ratio,

$$\frac{\text{maximum tooth width}}{\text{slot width}}$$

= 1.1 for large machines (H3)a

= 1.0 for small machines (H3)b

In the context of this thesis, a small machine is one whose rating lies between 10 and 200 kW; a large machine has its rating lying between 200 kW and 1000 kW.

Commutator diameter ,

- for large machines,

  = 0.6 x (armature diameter) (H4)a

- for small machines,

  = 0.75 x (armature diameter) (H4)b

Flux density is taken from the following table:

cycles per second	tooth flux density (T)	core flux density (T)
30	2.32	1.55
40	2.17	1.32
60	1.94	1.66

If the problem involves designing armatures with a given stator, then in order to use the same poles, yoke and armature parts, it is necessary that the flux per pole be the same in each case.

..... (L1)

Commutator segment size:

(i) The thickness of one copper and one mica segment taken together should not be very small. Otherwise a commutator with a large number of segments is liable to develop high bars.

(ii) With a two turn series winding half the number of segments are required but the reactance voltage is twice as large so that the machine will probably require high resistance brushes; this will have to be decided after the current rating has been found and the reactance voltage determined.

Reactance voltage:

Not to exceed 0.7 x (voltage drop per pair of brushes) for machines which operate at loads from no-load to 25% overload with brushes fixed in neutral position.

Not to exceed the voltage drop per pair of brushes for machines with the brushes shifted so as to help commutation.

Winding:

A series winding is the cheapest since it requires half as many coils and commutator bars as are required for the one-turn multiple winding. The first attempt for the type of winding should always be a series type. If it is not possible to do so, a multiple winding may be considered or else a decision be made for a new design of armature punching.

Slots size:

If the tooth density is to be kept the same then the slot area per pole must remain unchanged. If the core density is to remain unchanged then the depth of slot must be kept the same.

If slot dimensions have to be changed, they can only have values given in the tables of standard punches and dies.

Conductor size:

In order to get the largest possible rating out of the machine, the size of conductor should be the largest that can be got into the slot without making the coil too tight a fit.

Possible conductor sizes are given in the table which follows.

Note that designers might be using tables with non-SI units and probably all their experience is in terms of non-SI units; values do not have to be in SI units. But the system should work in SI units.

Each column is headed by a unique capital letter with the following meaning:

A = B&S gauge number

B = diameter(in)

C = diameter (cm)

D = cross section area in square inches

E = cross-sectional area in circular mills

F = cross-sectional area in square millimetres

G = diameter double cotton covered

H = insulation (mils on diameter

I = ohms per 1000 ft. at 15 degrees centigrade

J = ohms per 1000 ft. at 60 degrees centigrade

Table of copper wire gauge:

A	B	C	D	E	F	G	H	I	J
0000	.46000	11.684	.16619	211600	107.19	.472	12	.048119	.056969
000	.40964	10.404	.13179	167810	85.013	.421	12	.060677	.071838
00	.34680	9.266	.10452	133080	67.433	.377	12	.076511	.090583
0	.32495	8.252	.082931	105590	53.482	.337	12	.096428	.11416
1	.28930	7.348	.065733	83694	42.406	.301	12	.12166	.14403
8	.12849	3.264	.012967	16510	8.3673	.140	12	.61673	.73016
9	.11443	2.906	.010284	13094	6.6325	.126	12	.77760	.92061
10	.10189	2.588	.0081536	10382	5.2603	.144	12	.98078	1.1612
11	.09074	2.304	.0064637	8233.7	4.1692	.103	12	12.2366	1.4640
20	.03196	.813	.00080223	1021.4	.51912	.042	10	9.9682	11.80
24	.02010	.511	.00031738	404.01	.20508	.030	10	25.202	29.838

Ampere rating:

The current in each conductor should be such that, the ratio,

$$Rt = \frac{\text{amp. cond. per in.}}{\text{cir. mills per amp.}}$$

shall not exceed the value found from the curve below.

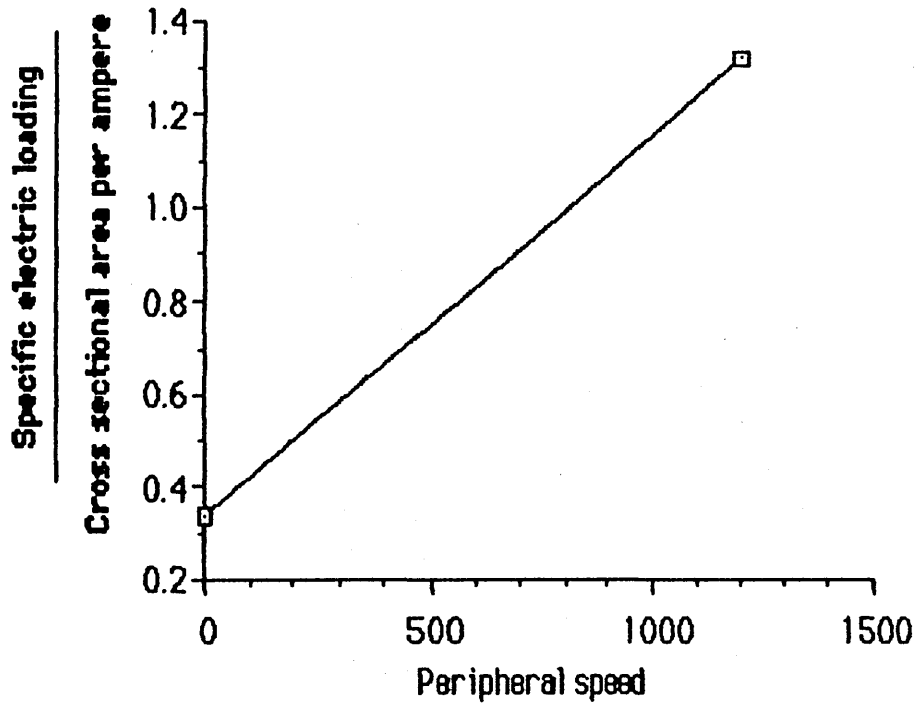


Fig. 5.4 Curve showing allowable armature copper losses in relation to its peripheral speed

Armature current, I_a:

From above, full load armature current is given by,

$$I_a^2 = \frac{\pi \, De \, Ac \, p a^2 \, Rt}{2N} \qquad \text{Eq}$$

5.2 A listing of existing designs

For the purpose of demonstration, we shall assume there are three existing designs whose details are,

Design number 1		
Armature external diameter	31.75	cm
Length of armature core	17.78	cm
Voltage	120	volts
Power	22.38	kW
Current	208	A
Speed	900	rpm
Number of poles	4	
Flux	0.0206	Wb
Number of paths	2	
Number of slots	49	
Number of turns	98	
Conductors per slot	4	

Slot height	2.3368	cm
Slot width	1.016	cm
Shaft diameter	4.0	cm

It is not necessary to quote all the above, details of a design may be different as below:

Design number 2

Frame size	132S	
Voltage	460	volts
Speed	1750	rpm
Power	11.0	kW
Current	31	A
Maximum speed	3200	rpm
Efficiency	0.795	
Moment of inertia	0.11	kgm^2
Weight	140	Kg

There is a stock for this machine

5.3 Symbols for and relationships between parameters

In the following, a list of symbols for parameter names, parameter values and the relationships between them will be given. Only relationships involving curves, tables and equations will be treated here. But before doing so, there is a remark to be made.

Whenever use is made of any of these relationships, each associated parameter, in general, could have its value either known or unknown depending on the stage a design has reached and the manner it was carried out. Therefore, when a relationship is given, it is to be understood as a statement of the link existing between the quantities involved. Wherever possible, following each relationship, will be an indication on how some of the parameters contained in it, are related to yet another set of parameters. When no such details are given, it means there are other relationships for such parameters which are not in the form of tables, equations or curves.

If one traces what actually happens during the design process, one finds that all relationships stem from the need to get solutions for voltage and current. The former usually involves an emf equation, and the latter involves one or more relationships which take into account, among others, of heat limits for the machine required. Current is also intimately related to voltage through the number of turns and conductor size. For this reason, we shall start with the basic relationship involving current and, in a top down fashion, introduce the rest of the parameters; and their relationships.

Each relationship will begin with a relation name in brackets and the relationship, in that order. Parameter names as they are known by the system, appear last enclosed between parentheses. Symbols representing values of parameters are explained wherever appropriate.

```
eqnlc       Lc   =  (Pin/Rpm).(608/(Bg.Rps.Sel.De^2)*100
                    :length of armature core                {lsc}

mloading    Bg   <= curve_mloading(De,Bg)
                    :apparent gap flux density              {bsg}

seloading   Sel  <= curve_seloading( Pin/1000 , Sel )
                    :specific electric loading              {sel}

eqnde       De   =  Ell/(pi.Sel)
                    :armature external diameter             {dse}

eloading    Ell  <= curve_eloading( [Pin/Rpm] , Ell )
                    :electric loading                       {ell}

amps2       I**2 =  pi*De*Ac*_pa*_pa*Rt/(2*N)
                    :full load armature current             {current}

            De   =  armature external diameter              {dse}

            _pa  =  :number of paths                        {paths}
```

```
                N    =  :number of turns                          {turns}

spload          Rt   <= curve_spload(Mps,Rt)
                        :Ratio ampere conductors per inch
                         to circ.mils. per amp                    {rst}

               Mps   =  :peripheral velocity                      {mps}

bswire          Ac   <= wire_gauge_table(Dc,Ac)
                        :cross sectional area of copper wire      {asc}

bswire          Dc   <= wire_gauge_table(Dct,Dc)
                        :actual wire size                         {dsc}

eqndct          Dc£  =  (2/cps)*(Wc+Hc/2)
                        :diameter of copper estimate              {dsc£}

               Cps   =  :conductors per slot                      {cps}

eqnhc           Hc   =  Hs - Tid
                        :available depth for copper
                         with insulation                          {hsc}

               Hs    =  :slot height                             {hss}

               Tid   =  :total insulation thickness along
                        :slot depth                               {tsid}
```

eqnwc	Wc	=	Ws – Tiw – Tc	
			:available width for coil	{wsc}
	Ws	=	:slot width	{wss}
	Tiw	=	:total slot insulation thickness along slot width	{tsiw}
	Tc	=	:net coil to core clearance along slot width	{tsc}
eqnfi	F	=	Bg.Rps.Tow.Lc	
			:flux per pole	{flux}
	Rps	=	0.7 (approx)	
			:pole enclosure, i.e., ratio pole face arc to pole pitch	{rsps}
eqnpo	Po	=	Pin.Eff	
			:output power	{power}
eqnpin	Pin	=	V*I	
			:Input power	{psin}
	V	=	:supply voltage	{voltage}

```
                Eff   =  :efficiency                          {efficiency}

                Rpm   =  :speed in revolutions per minute      {speed}

eqntow          Tow   =  pi.De/_po
                         :pole pitch                           {tow}

                _po   =  :number of poles                      {poles}
```

Alternative means of finding full load armature current:

```
amps            I**2  =  Pc.As.Ksf.(S/Z).(_pa/Z).(_pa/Lc).Sg
                         :full load armature current           {current}

                _pa   =  :number of paths                      {paths}

                 Z    =  :number of conductors                 {conductors)

                Lc    =  :length of core                       {lsc}

                Sg    =  :conductivity of copper               {sigma}

                Ksf   =  :stackig factor                       {kssf}

                 S    =  :number of slots                      {slots}


eqnpc           Pc    =  Ps - Pi
```

```
                        :max. copper loss allowed                {psc}

eqnps       Ps  =  _ps.Aa
                        :dissipated power from surface           {pss}

eqnsps      _ps =  0.0245(1+0.00127Mps).T
                        :specific power dissipated               {psps}

eqnaa       Aa  =  (pi).De.Lc.E-4
                        :armature surface area                   {asa}

            T   =  :temperature                                  {temperature}

eqnmps      Mps =  πDe.Rpm
                        :armature peripheral speed               {mps}

            Rpm =  :revolutions per minute                       {speed}

eqnpi       Pi  =  Pt + Py
                        :total iron loss                         {psi}

eqnpy       Py  =  _py.Vy
                        :iron loss in yoke                       {psy}

eqnpt       Pt  =  _pt.Vt
                        :iron loss in teeth                      {pst}
```

```
loss    _py   <= loss_curve(By,_py)
                 :yoke specific loss                    {pspy}

loss    _pt   <= loss_curve(Bt,_pt)
                 :teeth specific loss                   {pspt}

eqnby   By    =  (fhi/2)/Ay
                 :flux density in yoke                  {bsy}

eqnbt   Bt    =  fhi/At
                 :flux density in teeth                 {bst}

eqnay   Ay    =  Hy.Lc.KsfE-4
                 :longitudinal area of yoke             {asy}

eqnhy   Hy    =  (De/2)-Hs-Ds/2
                 :yoke height                           {hsy}

        Ds       :shaft diameter                        {dss}

        Hs    =  :slot height                           {hss}

eqnat   At    =  Wt.Lc.S/PE-4
                 :net tooth area under pole             {ast}

        P     =  :number of poles                       {poles}
```

```
eqnwt      Wt   =  [π(De-2Hs)-SWs]/S
                   :tooth width                          {wst}

           Ws   =  :slot width                           {wss}

eqnvt      Vt   =  [pi(De-Hs)Hs-SHtWs]Lc.E-6
                   :total volume of teeth                {vst}

eqnvy      Vy   =  pi(Ds+Hy)HyLc.E-6
                   :total volume of yoke                 {vsy}

eqnas      As   =  Ws.Hs.S/PE-4
                   :net slot area under pole             {ass}
```

5.4 **Translation of d.c. motor design knowledge**

The way you choose to represent data determines how easy or otherwise it is to invent and implement algorithms to achieve a goal. By reason of necessity any representation regime is related to the language of implementation and the kind of operations that are to be performed on the data. Perhaps the best thing to do at the moment is to take a design parameter as a basic data object. A parameter is represented by a term having a unary functor. The parameter name corresponds to the functor name; the parameter value being the functor argument. In this context, every design attribute is regarded as a parameter. Examples of some typical parameters are:

- a fact that number of turns is 345 is represented as turns(450)

- a voltage of 240 is represented as voltage(240)

- a wave type of winding is indicated as winding(wave)

For simplicity, each curve is stored as a series of line segments. A segment is represented in the form,

<*Curve_name*>(<*Abscissa_value*>,<*Ordinate_value*>,<*Slope*>,<*Intersection*>).

Each line segment starts from a given point having coordinates, (<*Abscissa_value*>,<*Ordinate_value*>) and drawn with a given <*Slope*> in the direction of decreasing values of the abscissa. It terminates at the next coordinate point which marks the beginning of the next segment. The of ordinate where it meet a line drawn parallel to the segment passing through the point, (<*Abscissa_value*>,<*Ordinate_value*>), is the point of <*Intersection*>.

```
/* curve for the specific loading

spload(0000.0, 0.34,   _     , _ ).
spload( 1200.0, 1.32, 8.166666E-04, 0.34).


/* Representation of the curve for the magnetic loading

mloading(25.4,0.674419,_,_).
mloading(31.75,0.682171,0.122077E-2,0.6434).
mloading(50.8,0.775194,0.48831EE-2,0.5727).
mloading(76.2,0.8255,0.198E-2,0.6744).
```

```
mloading( 101.6,0.868,0.167E-2,0.6976).
mloading( 127,0.89147,0.9155E-3,0.7752).
mloading( 147.32,0.899,0.381E-3,0.7752).
mloading152.4,0.903,0.7629E-3,0.786).
mloading( 177.8,0.918,0.6103E-3,0.81).
mloading( 203.2,0.93,0.4577E-3,0.837).
mloading( 228,0.93,0.6101E-5,0.929).
```

```
/* curve for output coefficient
```

```
co(0.004, 0.50,  _ ,   _ ).
co(0.010, 1.80, 216.667, - 0.3667).
co(0.020, 2.15, 35.000,  1.4500).
co(0.040, 2.50, 17.500,  1.8000).
co(0.060, 2.75, 12.500,  2.0000).
co(0.090, 3.00, 8.333,  2.2500).
co(0.100, 3.10, 9.998,  2.1000).
co(0.200, 3.50, 4.000,  2.7000).
co(0.500, 4.00, 1.667,  3.1667).
co( 1.000, 4.50,  1.000,  3.5000).
```

```
/* curve for specific electric loading
```

```
seloading( 10, 157.48,_,_).

seloading( 20, 181.1,2.36,133.86).

seloading( 30, 196.8,1.57,149.6).

seloading( 40, 210.6,1.377,155.5).

seloading( 50, 220.47,0.98,171.25).

seloading( 60, 228.3,0.787,181.1).

seloading( 70, 233.3,0.49,198.8).

seloading( 90, 243.11,0.492,198.8).

seloading( 100, 250,0.6889,181.1).


/* curve for elecric loading


eloading( 20, 17000.0,  _  ,  _  ).

eloading( 30, 20000.0, 300.000, 11000.0).

eloading( 40, 22666.7, 266.666, 12000.0).

eloading( 50, 25000.0, 233.334, 13333.3).

eloading( 60, 27000.0, 200.000, 15000.0).

eloading( 70, 28333.3, 133.332, 19000.1).

eloading( 80, 30000.0, 166.668, 16666.5).

eloading( 100, 32333.3, 116.666, 20666.7).
```

```
/* Curve for iron loss at 30 for frequency of 30 c/s
/* [Ref.: Puchstein Fig.6, pp 39]
/* The curve is approximately represented by a straight line
/* loss(Arg_1, Arg_2, Arg_3, Arg_4):
/* Arg_1 is x-coordinate value,  (flux density in Tesla)
/* Arg_2 is y-coordinate value, (power loss in watts per cubic metre)
/* Arg_3 is the slope of the line
/* Arg_4 is value of y when x=0.

loss(0.619963, 30512.2,  _ ,  _ ).
loss(2.634840, 286814.0, 127205, - 48350).
```

```
/* BROWN AND SHARP GAUGE: COPPER WIRE

/* Note: This is not a coplete list - only a sample is taken

/* format: bswire(A, B, C, D, E, F, G, H, I, J) where

/* A = B&S gauge number

/* B = diameter(in)

/* C = diameter (cm)

/* D = cross section area in square inches

/* E = cross-sectional area in circular mills

/* F = cross-sectional area in square millimetres

/* G = diameter double cotton covered

/* H = insulation (mils on diameter

/* I = ohms per 1000 ft. at 15 degrees centigrade

/* J = ohms per 1000 ft. at 60 degrees centigrade

bswire( a,.46000,11.684,.16619  .211600,107.19 ,.472,12, .048119,  .056969).

bswire( b,.40964,10.404,.13179  .167810,85.013 ,.421,12, .060677,  .071838).

bswire( c,.36480, 9.266,.10452  .133080,67.433 ,.377,12, .076511,  .090583).

bswire( 0,.32495, 8.252,.082931 .105590,53.482 ,.337,12, .096428,  .11416).

bswire( 1,.28930, 7.348,.065733 , 83694,42.406 ,.301,12, .12166 ,  .14403).

bswire( 8,.12849, 3.264,.012967 , 16510, 8.3673,.140,12, .61673 ,  .73016).

bswire( 9,.11443, 2.906,.010284 , 13094, 6.6325,.126,12, .77760 ,  .92061).

bswire(10,.10189, 2.588,.0081536, 10382, 5.2603,.144,12, .98078 ,  1.1612).

bswire(11,.09074,2.304,.0064637 , 8233.7 ,4.1692 ,.103, 12, 1.2366,  1.4640).

bswire(20,.03196, .813,.00080223, 1021.4 , .51912,.042, 10, 9.9682, 11.802).

bswire(24,.02010, .511,.00031738,  404.01, .20508,.030, 10,25.202 , 29.838).
```

```
/* Relationships applicable when looking for a preliminary design.

links(mloading,
bsg(Bg)&dse(De),
graph).

links(eloading,
ell(Ell)&psin(Pin)&speed(Rpm)& Rt is Pin/Rpm,
graph).

links(seloading,
sel(Sel)&psin(Pin)& Rt is Pin/1000,
graph).

/* flux(Fi) : flux per pole
links(eqnfi,
flux(Fi)&bsg(Bg)&rsps(Rps)&lsc(Lc)&tow(Tow),
Fi = Bg*Rps*Lc*Tow
   ).

/* dse(De) : Armature external diameter
links(eqnde,
dse(De)&ell(Ell)&sel(Sel),
De =  Ell/(pi*Sel)
   ).
```

```
/* psin(Pin) : Input power in Watts
links(eqnpin,
psin(Pin)&voltage(V)&current(I),
Pin = V*I/1000
    ).

/* tow(Tow) : Pole pitch
links(eqntow,
tow(Tow)&dse(De)&poles(_po),
Tow = pi*De/_po
    ).


/* lsc(Lc) : length of armature core
links(eqnlc,
lsc(Lc)&psin(Pin)&speed(Rpm)&bsg(Bg)&rsps(Rps)&sel(Sel)&dse(De),
Lc = (Pin/Rpm)*60.8E7/(Bg*Rps*Sel*De**2)
    ).

choose(efficiency(0.9)).
choose(rsps(0.7)).
choose(poles(2)).
choose(poles(4)).
choose(poles(6)).
choose(poles(8)).
choose(poles(10)).
```

```
choose(poles(12)).
choose(poles(14)).
choose(poles(16)).
choose(poles(18)).

reject(poles(_po)) :-
    apply(known(Parameter),Parameter,[dse(De),lsc(Lc)]),
  - in_between(pi*De/(_po*Lc), 1.1, 1.7).

employ(graph(mloading),bsg(Bg),use).
employ(graph(eloading),ell(Ell),use).
employ(graph(seloading),sel(Sel),use).
employ(eqnfi,flux(F),find).
employ(eqnde,dse(De),find).
employ(eqnpin,psin(Pin),find).
employ(eqntow,tow(Tow),find).
employ(eqnlc,lsc(Lc),find).

:- consult('mloading.crv').
:- consult('eloading.crv').
:- consult('seloading.crv').
```

A DESIGN KNOWLEDGE BASE

(DESKBG)

```
/*    A GENERAL KNOWLEGE BASE FOR ROTATING ELECTRICAL MACHINES

can(select(slots(S)), poles(_po)&choose(slots(S),1) ) .

start_with(choose(slots(S),1)).

can(select(paths(_pa)) , winding(Type) &  choose(paths(_pa),Type)).

/* For a simple series (wave) winding the number of armature parallel paths equals
to 2:
choose(paths(2) , wave).

/* For a simple multiple (lap) winding the number of armature parallel paths equals
to the
/* number of pole pairs:
choose(paths(_pa), lap) :-
    known(poles(P)),
   _pa is P/2 .

add(Parameter, select(Parameter)).
```

```
can(select(Par(Parval)), true) :-
  var(Parval),
   is_arg( 1 ,choose,Par(Parval_1),Clause_head_1),
  substitute(Clause_head_1 ,Parval_1 ,Parval,Clause_head),
  Clause_head.

add(hss(Hs), keep(hss(Hs))).
add(wss(Ws), keep(wss(Ws))).
can(keep(hss(Hs)), adapting(Des_no)&des(Des_no,hss(Ht))&Hs is Ht/2.54).
can(keep(wss(Ws)),
      adapting(Des_no)&slots(S)&des(Des_no,slots(S))&des(Des_no,wss(Ws))).
can(select(wss(Ws)),
    adapting(Dno)&slots(S)&des(Dno,slots(S1))&
    des(Dno,wss(_ws))& Ws is (_ws/2.54)*S1/S).

choose(winding(wave)).
choose(winding(lap)).
choose(kssf(0.925)).
choose(sigma(5.8E7)).
choose(tsiw(0.08)). /* dimensions in inches !!
choose(tsc(0.04)).  /*   - || -
choose(tsid(0.06)).  /* dimensions in inches !!

choose(slots(36)).
choose(slots(37)).
choose(slots(41)).
choose(slots(49)).
```

```
choose(slots(53)).

choose(slots(65)).

choose(slots(72)).

choose(slots(81)).

choose(slots(96)).

choose(slots(99)).

choose(slots(128)).


links(emf,

voltage(V)&turns(N)&speed(Rpm)&flux(_wb)&paths(_pa)&poles(_po),

V = 2*N*_wb*(Rpm/60)*(_po/_pa) ).


links(eqnpo,

power(Po)&psin(Pin)&efficiency(Eff),

Po = Pin * Eff

   ).


links(amps,

current(I)&psc(Pc)&ass(As)&kssf(Ksf)&

turns(N)&slots(S)&paths(_pa)&lsc(Lc)&sigma(Sg),

I**2 = Pc*As*Ksf*(S/N)*(_pa/N)*(_pa/Lc)*Sg*100/4

   ).


links(eqnpc,

psc(Pc)&pss(Ps)&psi(Pi),

Pc = Ps - Pi

   ).
```

```
links(eqnps,
pss(Ps)&psps(_ps)&asa(Aa),
Ps = _ps*Aa
    ).

links(eqnsps,
psps(_ps)&mps(Mps)&temperature(T),
_ps = 0.0245*(1+0.00127*Mps)*T*144*10.764
    ).

links(eqnaa,
asa(Aa)&dse(De)&lsc(Lc),
Aa = pi*De*Lc/10000
    ).

links(eqnmps,
mps(Mps)&dse(De)&speed(Rpm),
Mps = pi*De*Rpm/100
    ).

links(eqnpi,
psi(Pi)&pst(Pt)&psy(Py),
Pi = Pt + Py
    ).
```

```
links(eqnpt,
pst(Pt)&pspt(_pt)&vst(Vt),
Pt = _pt*Vt
    ).

links(eqnpy,
psy(Py)&pspy(_py)&vsy(Vy),
Py = _py*Vy
    ).

links( loss, pspy(_py)&bsy(By), graph).

links( loss, pspt(_pt)&bst(Bt), graph).

links(eqnby,
bsy(By)&flux(Fi)&asy(Ay),
By = (Fi/2)/Ay
    ).

links(eqnbt,
bst(Bt)&flux(Fi)&ast(At),
Bt = Fi/At
  ).

links(eqnay,
asy(Ay)&hsy(Hy)&lsc(Lc)&kssf(Ksf),
```

```
Ay = Hy*Lc*Ksf/10000
  ).

links(eqnhy,
hsy(Hy)&dse(De)&hss(Hs)&dss(Ds),
Hy = De/2 - Hs - Ds/2
    ).

links(eqnat,
ast(At)&wst(Wt)&lsc(Lc)&slots(S)&poles(P),
At = Wt*Lc*S/P/10000
    ).

links(eqnwt,
wst(Wt)&dse(De)&hss(Hs)&wss(Ws)&slots(S),
Wt = (pi*(De-2*Hs)-S*Ws)/S
    ).

links(eqnvy,
vsy(Vy)&dss(Ds)&hsy(Hy)&lsc(Lc),
Vy = pi*(Ds + Hy)*Hy*Lc/1000000
    ).

links(eqnvt,
vst(Vt)&dse(De)&hss(Hs)&slots(S)&wss(Ws)&lsc(Lc),
Vt = (pi*(De-Hs)*Hs-S*Hs*Ws)*Lc/1000000
    ).
```

```
links(eqnas,
ass(As)&wss(Ws)&hss(Hs)&slots(S)&poles(P),
As = Ws*Hs*S/P/10000
    ).

/* Facts and rules applicable to determine the current rating given:
/* (a) A curve of peripheral velocity versus the ratio of specific electric loading to
/*      conductor current density
/* (b) A table of sizes of wire available

links(amps2,
current(I)&dse(De)&dsc(Dc)&asc(Ac)&paths(_pa)&rst(Rt)&turns(N),
/* here dsc(Dc) is included so that asc(Ac) may be found
/* it is not possible to know asc(As) before dsc(Dc) in table.
I**2 = pi*(De/2.54)*Ac* _pa * _pa *Rt/(2*N)
    ).

links('eqndsc£',
'dsc£'(Dc)&cps(Cps)&wsc(Wc)&hsc(Hc),
Dc = (2/Cps)*(Wc+Hc/2)
    ).

links(eqnwc,
wsc(Wc)&wss(Ws)&tsiw(Tiw)&tsc(Tc),
Wc = (Ws/2.54) - Tiw - Tc
```

```
    ).links(eqnhc,
hsc(Hc)&hss(Hs)&tsid(Tid),
Hc = Hs - Tid
    ).
```

```
/* Format for table of wire gauge

links(bswire,
a1(A1)&dsc(Dc)&a3(A3)&a4(A4)&asc(Ac)&
a6(A6)&a7(A7)&a8(A8)&a9(A9)&a10(A10),
table
    ).

links(sploed,rst(Rt)&mps(Mps), graph).

  start_with(choose(Agoal,Anything)).


affects(turns(N),slots(S)).

reject(dummy).

kind(power, 0.85, 1.15).
kind(Par_name, 0.9, 1.10).  /* catch all
/* Note: 'voltage' can't be included otherwise it becomes inconsistent
/*       since 'voltage' appears in "assent" clause above.

require(maximum(power,Design_no)) :-
apply((des(Design_no,Par),asserta(always(Par))),Par,
                [flux(Fi),dse(De),lsc(Lc),poles(_po),dss(Ds)]
      ).
```

```
employ(emf,voltage(V),change).

employ(emf,turns(N),change).

employ(emf,speed(Rpm),change).

employ(amps,current(I),find).

employ(eqnpc,psc(Pc),find).

employ(eqnps,pss(Ps),find).

employ(eqnsps,psps(_ps),find).

employ(eqnaa,asa(Aa),find).

employ(eqnmps,mps(Mps),find).

employ(eqnpi,psi(Pi),find).

employ(eqnpt,pst(Pt),find).

employ(eqnpy,psy(Py),find).

employ(eqnby,bsy(By),find).

employ(eqnbt,bst(Bt),find).

employ(eqnay,asy(Ay),find).

employ(eqnhy,hsy(Hy),find).

employ(eqnat,ast(At),find).

employ(eqnwt,wst(Wt),find).

employ(eqnvy,vsy(Vy),find).

employ(eqnvt,vst(Vt),find).

employ(eqnas,ass(As),find).
```

```
employ(eqnpo,power(Watts),_).
employ(eqnpo,power(Watts),find).  /* 'find' needs explicit mention
employ(eqnpo,psin(Kw),_).
employ(eqnpo,psin(Kw),find).      /* 'find' needs explicit mention

employ(amps2,current(I),find).
employ('eqndsc£', 'dsc£'(Dc),find).
employ(eqnwc,wsc(Wc), find).
employ(eqnhc,hsc(Hc), find).

employ(graph(loss),pspy(_py),use).
employ(graph(loss),pspt(_pt),use).
employ(graph(sploed),rst(Rt),use).

employ(table(bswire),dsc(Dc),use).
employ(table(bswire),asc(Ac),use).
parameter(211,rsps(Rps)).
parameter(211,bsg(Bg)).
parameter(211,ell(Ell)).
parameter(211,psin(Pin)).
parameter(211,efficiency(Eff)).
parameter(211,sel(Sel)).
parameter(211,tow(Tow)).

parameter(1,voltage(V)).
parameter(20,turns(N)).
parameter(10,speed(Rpm)).
```

```
parameter( 100,flux(_wb)).
parameter( 100,paths(_pa)).
parameter( 100,poles(_po)).
parameter( 250,current(C)).
parameter( 120,slots(S)).
parameter( 260,cps(Cps)).
parameter( 500,power(_watts)).
parameter( 110,winding(_type)).

parameter( 249,lsc(Lc)).
parameter( 249,sigma(Sg)).
parameter( 249,kssf(Ksf)).
parameter( 249,psc(Pc)).
parameter( 249,pss(Ps)).
parameter( 249,psps(_ps)).
parameter( 249,asa(Aa)).
parameter( 249,dse(De)).
parameter( 249,temperature(T)).
parameter( 249,mps(Mps)).
parameter( 249,psi(Pi)).
parameter( 249,psy(Py)).
parameter( 249,pst(Pt)).
parameter( 249,bsy(By)).
parameter( 249,bst(Bt)).
parameter( 249,asy(Ay)).
parameter( 249,hsy(Hy)).
parameter( 249,dss(Ds)).
```

```
parameter(249,hss(Ht)).
parameter(249,ast(At)).
parameter(249,wst(Wt)).
parameter(249,wss(Ws)).
parameter(249,vsy(Vy)).
parameter(249,vst(Vt)).
parameter(249,ass(As)).
parameter(250,pspy(_py)).
parameter(250,pspt(_pt)).

parameter(249,dsc(Dc)).
parameter(249,'dsc£'(Dc)).
parameter(249,wsc(Wc)).
parameter(249,tsiw(Tiw)).
parameter(249,tsc(Tc)).
parameter(249,hsc(Hc)).
parameter(249,tsid(Tid)).
parameter(249,asc(Ac)).
parameter(249,rst(Rt)).
```

asa means 'Armature surface area' & 'sq. cm.' => 'F5.2'.

asc means 'Cross sectional area of winding wire' & 'sq. cm.' => 'F7.1'.

ass means 'Net slot area under pole' & 'sq. cm.' => 'F5.2'.

ast means 'Net tooth area under pole' & 'sq. cm.' => 'F5.2'.

asy means 'Area of yoke in azimuthal plane' & 'sq. cm.' => 'F5.2'.

bsg means 'Apparent gap flux density' & 'Tesla' => 'F6.4'.

bst means 'Flux density in the teeth' & 'Tesla' => 'F6.4'.

bsy means 'Yoke flux density' & Tesla => 'F6.1'.

conductors means 'Number of armature conductors' & number => 'F4.0'.

cps means 'Armature conductors per slot' & number => 'F3.0'.

current means 'Full load current' & 'A.' => 'F4.0'.

dsc means 'Diameter of winding wire' & 'cm.' => 'F6.2'.

'dsc£' means 'Diameter of copper wire, estimate' & 'cm.' => 'F6.2'.

dse means 'Armature external diameter' & 'cm.' => 'F5.2'.

dss means 'Shaft diameter or internal yoke diameter' & 'cm.' => 'F5.2'.

efficiency means 'Efficiency' & number => 'F4.2'.

ell means 'Electric loading' & 'ampere conductors' => 'F5.0'.

flux means 'Flux' & 'Wb.' => 'F6.4'.

hsc means 'Available depth for copper

hss means 'Slot height' means 'Yoke height' & 'cm.' => ' means 'Length of arm

poles means 'Number of poles' & number => 'F3.0'.

power means 'Output power' & 'Kw.' => 'F3.0'.

psc means 'Maximum copper loss allowed' & 'W.' => 'F5.0'.

psi means 'Total armature iron loss' & 'W.' => 'F4.0'.

psin means 'Input power' & watts => 'F6.0'.

psps means 'Power dissipated per unit surface area' & 'W/sq.cm' => 'F5.0'.

pspt means 'Teeth specific iron loss' & 'W/cubic cm.' => 'F5.0'.

pspy	means 'Yoke specific iron loss' & 'W/cubic cm.' => 'F5.0'.
pss	means 'Dissipated power from armature surface' & 'W.' => 'F5.0'.
pst	means 'Iron loss in armature teeth' & 'W.' => 'F5.0'.
psy	means 'Iron loss in armature yoke' & 'W.' => 'F4.0'.
rsps	means 'Pole enclosure' & number => 'F4.2'.
rst	means 'Ratio ampere conductors per in. to cir. mils per amp' & number => F4.2'.
sel	means 'Specific electric loading' & 'ampere conductors per cm.' => 'F5.0'.
sigma	means 'Conductivity of winding wire' & 'mho/m' => 'E6.4'.
slots	means 'Number of armature slots' & number => 'F4.0'.
speed	means 'Speed' & rpm => 'F5.0'.
temperature	means 'Temperature rise of embeded condutors' & 'deg. Cent.' => 'F3.0'.
tow	means 'Pole pitch' & 'cm.' => 'F5.2'.
tsc	means 'Net coil to core clearance along slot width' & 'cm.' => 'F5.3'.
tsiw	means 'Total slotong slot width' & 'cm.' => 'F5.2'.
N	means 'Number of turns' & number.
vst	means 'Total volot' & 'cm.' => 'F5.3'.
wst	means 'Tooth width' & 'cm.' => 'F5.3'.

```
/*  The existing designs:

 des( 1,dse(31.75)).   /* external diameter (cm)
des( 1,lsc( 17.78)).   /* length of core  (cm)
des( 1,voltage( 120)).  /* (V)
des( 1,power(22.38)).  /* (kW)
des( 1,current(208)).  /* (A)
des( 1,speed(900)).    /* (rpm)
des( 1,poles(4)).
des( 1,flux(0.206E-1)). /* (Wb)
des( 1,paths(2)).
des( 1,slots(49)).
des( 1,turns(98)).
des( 1,cps(4)).   /* (conductorsper slot)
des( 1,hss(2.3368)).  /* slot height (cm)
des( 1,wss( 1.016)).  /* slot width (cm)
des( 1,dss(4.0)).  /* shaft diameter (cm)
 des( 2,frame('132S')).
des( 2,voltage(460)).
des( 2,speed( 1750)).
des( 2,power( 11.0)).
des( 2,current(31)).
des( 2,speedmax(3200)).
des( 2,eff(79.5)).
des( 2,minert(0.11)).  /* moment of inertia
des( 2,weight( 140)).
```

```
 des(3,frame('160L')).
des(3,current(103)).

/* stock(X) : Design number <X> is in stock
stock(dummy).
```

5.5 Design Problem Examples

In the following, the normal manual design process for a d.c. armature is outlined. The intention is to pave the way to the knowledge based example which follow. Two cases are considered. One is concerned with modifying a suitable existing design to meet the specification. The other deals with making a design, starting from scratch.

The idea is to convey the impression how the system reasons out to arrive at a design; a replica of the manual design.

5.5.1 Starting from Scratch

First a preliminary design is carried out, followed by a more detailed design procedure.

Specification:

It is required to design an armature for a d.c. motor according to the following specifications: 120 volts, 30 hp, 900 r.p.m.

First Stage:

Establish, using simple hand calculator methods, the approximate dimensions of the armature.

Voltage given	E =	12000	volts
Speed given	Rpm =	900	rpm
Power output given	Po =	22380	watts
Probable efficiency	Efe =	90%	
Full load current I = Po/(Eff x E) = 208	I =	208	amperes
Power input	Pin =	25000	watts
Ampere conductors : from Fig. 5.2	Ell =	0.19E5	
Ampere conductors per in : from Fig. 5.3	Sel =	189	AT/cm

Armature diameter De = Ell/(π.Sel)

= 0.19E5/(π x 189) = 32	De =	32	cm
Apparent gap density, from Fig. 5.1	Bg =	0.68	Tesla
Pole enclosure - ratio	Rpe =	0.7	

Frame length

$$Lc = 25000 \times \frac{60}{900} \times \frac{1}{0.68 \times 0.7 \times 189\pi^2 \times 0.32^2}$$

= 18.3 Lc = 18.3 cm

Poles : use rule :

(pole pitch)/(frame length) = 1.1 to 1.7

i.e. $\pi De/(_po.Lc) = 1.1$ to 1.7

for _po = 2 ; = 2.75 /* too high

for _po = 4 ; = 1.37 /* within range

for _po = 6 ; = 0.91 /* too low

thus take 4 poles _po = 4

Pole pitch Tow = π.De/_po = πx32/4 Tow = 25 cm

Flux per pole ϕ = Bg.Rpe.Tow.Lc

= 0.68x0.7x25x10^{-2}x18.3x10^{-2}= 0.02178 ϕ = 0.0217 Wb

Total number of conductors, Z :

$$Z = \frac{E}{\phi} \cdot \frac{60}{Rpm} \cdot \frac{_pa}{_po}$$

$$= \frac{120 \times 60 \times 2}{0.0278 \times 900 \times 4}$$

= 181.818 Z = 182

Detailed design

Having established the preliminary starting set of values, the design can proceed to a more detailed design to produce final design values.

External diameter - from preliminary design De = 32 cm

Probable number of conductors

from preliminary design Z£ = 182

Winding:

The minimum number of slots per pole = 12

Thus minimum number of total slots = 48

But Slots = k . _po/2 ± 1 for a series winding

Nearest suitable number is 49	S =	49	
Coils per slot Cps = Z£/S = 3.71	Cps =	4	
Coils or turns N = Cps.S/2 = 98	N =	98	
Active coils or turns = 97	Nac =	97	
Winding : one turn series			
Active conductors	Zac =	194	
Ampere per conductor Ic = Ia/_pa = 208/2 = 104	Ic =	104	

Ampere conductors per inch, Sel

$$Sel = Z.Ic/(\pi.De)$$

$$= 194 \times 104/(\pi \times 32) = 201 \qquad Sel = 201 \text{ AT/cm}$$

Peripheral velocity, Mps:

$$Mps = \pi.De.Mps$$

$$= \pi \times 32 \times 10^{-2} \times 900 = 904.8 \qquad Mps = 904.8 \text{ m/min}$$

$$Rt = \frac{\text{ampere conductors per cm.}}{\text{sq. cm. per ampere}} = 8.29 \times 10^{4}$$

Sq. cm. per ampere = Sel / Rt = 0.00242	Ai =	0.00242	sq. cm.
Section of conductor Ac = Ai.Ic = 0.00242 x104			
= 0.252	Ac =	0.252	sq. cm.

From table, a wire of about 0.584 cm. diameter would be needed

Slot pitch $32\pi/49 = 2.05$	2	cm
Probable slot width 2/2 = 1 cm (H3: page 94)	1	cm

With this slot size, the round wire above is

not suitable. So strip wire will be chosen.

Put two conductors in the width of the slot

Width of slot insulation		0.162	cm
Clearance between coil and core		0.102	cm

The coil is insulated with tape as follows:

(a) taping each strip with one layer 0.015 cm. thick

(b) taping the coils which make one group with

the same thickness of tape

Total insulation thickness along slot width		0.091	cm
Available width for copper = 0.4-0.064-0.04-0.036		0.66	cm
Make the strip width to be - avoiding tightness		0.30	cm
Depth of conductor 0.252/0.345	hsc =	0.827	cm

Slot depth:

- Depth of each conductor = 0.827
- Insulation thickness on each conductor: 0.015 x 2 = 0.03 cm
- Insulation of conductors in a group: 0.015 x 2 = 0.03 cm
- Depth of slot insulation = 0.025 x 6 = 0.15
- Thus depth of each insulated coil = 1.0 cm
- Total depth of coils in slot = 1.0x 2 = 2.0 cm
- Thickness of stick in top of slot = 0.508 cm

Necessary depth of slot = 1.0 + 0.508 cm	wss =	1.5	cm

Diameter at bottom 32-(2x1.5) = 29 cm

Slot pitch at bottom of slot π x 29/49 = 1.86 cm

Minimum tooth width = 1.86 - 1.0 = 0.86 cm

Frame length - from preliminary design Lc = 18.3 cm

Centre vent ducts: rule:

number of ducts {Nd} to be such that,

Lc/Nd < 7.63 cm; each duct of width 0.952 cm Nd = 2

Gross iron in frame length {Lgc}

Lgc = Lc - 0.952xNd = 18.3 - 1.9 = 16.4 Lgc = 16.4 cm

Stacking factor for armature laminations Kssf = 0.9

Net iron in frame length {Lnc},

Lnc = Kssf.Lgc = 0.9x16.4 = 14.76 cm Lnc = 14.76 cm

Minimum tooth area per pole (49/4)x0.7x0.86x14.76

= 108.85 sq. cm

Flux per pole for 194 conductors,

(120/194).(60/900).(2/4) = 0.0206 ϕ = 0.0206 Wb

Thus max. tooth flux density: Btm,

Btm = ϕ /Min. tooth area

= 0.0206/(108.84 x 10^{-4}) = 1.89 Btm = 1.89 Tesla

This is rather too low a value;

we would like to raise it to 2.32 Tesla

So shorten core length to 18.3 x (1.89/2.32)=14.91

that is, Lc = 14.91 Lc = 15 cm

Maximum tooth flux density = 2.32 Tesla

Flux density in the core ,assumed 1.32 Tesla

Core area (0.0206/2)/1.32 x 10^4 = 78.03 sq. cm

Gross iron in frame length 15-1.9 = 13.1	Lgc =	13	cm
Net iron 13 x 0.9	Lnc =	11.7	cm
Core depth 78/11.7 = 6.667 cm		6.67	cm

The above is lengthy but considered necessary to compare with the knowledge based equivalent designs given below.

5.5.2 Modifying an existing design

Another example to be considered is modification of an existing design. Given a d.c. motor, the design of which is detailed below, it is required to design the armature for a different voltage and speed. The aim is, to determine the number of turns for the armature winding, the conductor size, number and size of armature slots, current and power ratings.

Details of the design to be modified are as follows,

Rated _power	22.38	kW
Rated -voltage	120	volts
Rated -current	208	amperes
Rated -speed	900	r.p.m.
Number of poles	4	
Armature		
External diameter	31.75	cm
Frame length	17.78	cm

Slots:	
number	49
size	1.016 x 2.3368 cm
Conductors	
number per slot	4
size	0.3048 x 0.8128 cm
Coils	98, 1 dead
Winding,	wave , i.e.:
	1 T.S. (one turn series)
Commutator	
Diameter	24.130 cm
Face	17.145 cm
Bars, number	97

Specification :

The motor is required to have a supply voltage of 500 volts and a normal running speed of 950 r.p.m.

Solution by an expert:

We would look through data base and use appropriate formulae or techniques and decide on the dimensions of the machine required.

Somehow empirically we identify a frame size of an existing stock but realise that the voltage is different and maybe the armature current is different.

1. Find Z, the number of armature conductors using the emf equation.

   By substituting in the values, a figure of 765.78 is obtained.

2. Choose the cheapest winding; a series winding is generally the cheapest.

3. Winding

3.1 For a one-turn series winding

Determine the number of slots:

Here we apply formulae, rules and heuristics given in section 6.2 which are [F2, F3, H1a]

The nearest suitable number of slots is 49

- conductors per slot = 765.78/49 = 15.626 ~ 16
- thus no. of coils = (16 x 49)/2 = 392
- since the number of active turns must be odd for a one turn series winding, the winding will be composed of 391 active turns and an extra single dead winding.

  * the actual number of active conductors will be 391 x 2 = 782 conductors.

Find maximum conductor size that can be got into slot without making the coil too tight a fit:

Taking round conductors,

width of slot = 1.016 cm

width of slot insulation 0.203 cm

clearance between coil and core = 0.102 cm

available width for copper and insulation on conductors 0.711 cm

depth of slot 2.337 cm

depth of slot insulation 0.152 cm

available depth for copper and its insulation 2.184 cm

The nearest suitable diameter for round wire which can fit reasonably well in the net slot space, would be that corresponding to wire gauge number 11(using B&S wire gauge) whose diameter and cross sectional area are .2304 centimeters and 0.0417 square centimeters respectivelly.

armature peripheral velocity = π *De*Speed

$$= \pi(31.75)*950 = 9.48 \times 10^4 \text{ cm/min}$$

from Fig. 5.4:

for peripheral velocity of 9.48×10^4 cm/min, then:

$$Rt = \frac{\text{ampere conductors per cm.}}{\text{sq. cm. per ampere}} = 8.68 \times 10^4$$

$$I^2 = \pi \times De \times Ac \times _pa^2 \times Rt /(2N)$$

$$= \pi \times 12.5 \times 0.0417 \times 2^2 \times 8.68 \times 10^4/(2 \times 391) = 42.97^2$$

$$= 43 \times 43$$

Thus full load current is 43 amperes.

The design has thus been modified to meet the new specification

5.6 A Session With the Design System

In this section the above specifications will be run through the basic design program written using Prolog. The program is known locally as "An Assistant Designer of Electric Machiones". For brevity, this is reduced to ADFMA.

A typical session to run specifications used for manual designs above, would proceed as follows:

Please note that the user input is underlined

Computer responces and prompts are in bold type

OK, prolog desplan

&&&&&&&&&&&&&&&&&&&&&& A D F M A &&&&&&&&&&&&&&&&&&&&&&

welcome to a design session with

an Assistant Designer of Electromagnetic Machines

] ?carryon.

adapt opts To modify a design

new opts To start a completely new design

save opts To save the state of the system for subsequent restart

give opts To give design details - will be prompted for goal(s)

store opts To store design details obtained sofar

Please enter option :

Type 'help' for a list of options available

option>give

Please enter your specifications for the machine desired :

(form : <parameter name> = <parameter value>)

specs >voltage=120 /* the user now enters the specifications

specs >speed=900

specs >power=22380

specs >end

There is no existing design to meet the specifications

Cannot even find a suitable design for modification

The only possibility remaining is designing from scratch

Do you wish to make a completely new design ? yes

```
start;
select(efficiency(0.9));
find(eqnpo,psin(24866.6666667));
use(graph,eloading,ell(19288.8888889));
use(graph,seloading,sel(188.766026667));
find(eqnde,dse(32.5131266772));
use(graph,mloading,bsg(0.685897076469));
select(rsps(0.7));
```

```
compute(eqnlc,lsc(17.5339734293));
select(poles(4));
find(eqntow,tow(25.5460281034));
find(eqnfi,flux(0.215060534813E-1));
select(winding(wave));
select(paths(2));
select(slots(49));
change(emf,voltage(120),turns(98),slots(49));
change(eqnpo,power(22400),psin(24866.6666667),true).
```

Pole enclosure	= 0.70
Apparent gap flux density	= 0.686 Tesla
Electric loading	= 19289 ampere conductors
Input power	= 24867. watts
Efficiency	= 0.90
Specific electric loading	= 189. ampere cond. per cm.
Pole pitch	= 25.55 cm
Terminal (rated) voltage	= 120. V.
Number of turns	= 98.
Speed	= 900. rpm
Flux	= 0.0215 Wb.
Number of paths	= 2.
Number of poles	= 4.
Number of armature slots	= 49.
Armature conductors per slot	= 4.
Output power	= 22.4 kW.
Armature type of winding	= wave
Length of armature core	= 17.53 cm
Armature external diameter	= 32.51 cm

Do you need more possible designs (Y/N) ? n

adapt opts To modify a design

new opts To start a completely new design

save opts To save the state of the system for subsequent restart

give opts To give design details - will be prompted for goal(s)

store opts To store design details obtained sofar

Please enter option :

Type 'help' for a list of options available

option>adapt 1

Following are details of the design to be modified:

Armature external diameter	= 31.75 cm.
Length of armature core	= 17.78 cm.
Terminal (rated) voltage	= 120. V.
Output power	= 22. Kw.
Full load current	= 208. A.
Speed	= 900. rpm
Number of poles	= 4.
Flux	= 0.0206 Wb.
Number of paths	= 2.
Number of armature slots	= 49.
Number of turns	= 98.
Armature conductors per slot	= 4.
Slot height	= 2.33 cm.
Slot width	= 1.02 cm.
Shaft diameter or internal yokediameter	= 4.00 cm.

Please enter your specifications for the machine desired :

(form : <parameter name> = <parameter value>)

specs >voltage=500

specs >speed=950

specs >end

Please give other specific requirements, if any:

req > power is maximum

req > end

start;

select(winding(wave));

select(paths(2));

select(slots(49));

change(emf,voltage(500),turns(392),slots(49)).

Terminal (rated) voltage	= 500. V.
Number of turns	= 392.
Speed	= 950. rpm
Flux	= 0.0206 Wb.
Number of paths	= 2.
Number of poles	= 4.
Number of armature slots	= 49.
Armature conductors per slot	= 16.
Armature type of winding	= wave

Do you need more possible designs (Y/N) ? n

adapt opts To modify a design

new opts To start a completely new design

save opts To save the state of the system for subsequent restart

give opts To give design details - will be prompted for goal(s)

store opts To store design details obtained sofar

Please enter option :

Type 'help' for a list of options available

option>end

] ?g

OK,

Obviously more designs would be available in an industrial design office - thus further facilitating the application of the system.

Chapter 6

OTHER ASPECTS

6.1 **Introduction**

In the course of the planning and implementation of this system; the need for a control program to be independent of specific domains continued to be a major consideration and the development of software base of utility programs was of paramount importance to this.

6.2 **Observations**

Currently, writing an expert system is more of an art than a science. The science is emerging but it is taking time.

An element of luck is responsible for the use of Salford Lisp/Prolog package. At the beginning of the project, although it was suggested by Prolog proponents that one could do any thing with it, it was nonetheless not clear what this would entail, and so it was difficult beforehand to specify the requirements. The initial experience was with a Prolog version implemented by a group at Imperial College running on microcomputers; appropriately called Micro-Prolog. It has a syntax similar to Lisp, so that rules and fact are list structures. Ultimately it was found unsuitable for this

project because it had no provision for calling and passing arguments to and from routines written in other languages. Later, a Prime computer was acquired supporting the Salford Lisp/Prolog system which by coincidence turned out to meet most of the basic needs.

Building a knowledge based system in one language can be very difficult if not impossible in view of the diverse requirements of the system builder. On some occasions it was not possible to implement some elements of the design process using Prolog. Certain functional constructs were best realized in Lisp [22].

All this leads to a very important conclusion that the most useful quality of a development environment is one which hosts several different programming languages. For then one would be able to concentrate on knowledge acquisition instead of spending time and effort in improvising schemes to circumvent limitations imposed by a single language.

When considering the building of a system for doing design, it is not a good idea to think of building it piece by piece, by considering one aspect at a time; the bottom up approach. The architecture of the basic inference system has to be constructed taking into account the entire design process. In essence a top down approach is necessary.

6.3 Shells

Depending upon the approach taken, building a knowledge based system from scratch can be technically very demanding (and expensive) with prohibitive lead times. A large proportion of the time was spent in devising the top level control program. The author believes this time could be considerably reduced if the control program was available in the first place. It is hoped that such design shells will be commercially available before very long, but as yet, they have still to emerge.

For application areas other than design, packages are now available which make it possible to build what are commonly referred to as expert systems within very short times. A well known vendor recently claimed that on average a first time user starts entering rules within three to four days' use of their product. This might well be true, provided the shell matches the user's problem domain. The point is, almost invariably, that each type of shell now in the market, will be matched to a certain type of application. Most shells available are intended for such problem domains as diagnostics. For most of them, problem deduction evolves around the use of "IF . . . THEN . . . ELSE . . ." rules. As yet the author has no knowledge of shells specifically tailored for design application areas. General planning systems do exist however. A selection of such systems together with areas to which they have been applied, is shown in Table 6.1.

Area of application	Planner
Robot control	STRIPS
Simple program generation	HACKER
Mechanical engineers apprentice supervision	NOAH
House Building and electrical turbine overhaul	NONLIN
Experiment planning in molecular genetics	MOLGEN
Electronic circuit design	NASL
Journey planning	OPM
Job shop scheduling for turbine production	ISI-II
Aircraft carrier mission planning	SIPE, AIRPLAN
Naval Logistics	NONLIN
Voyager spacecraft mission sequencing	DEVISER

Table 6.1 A sample of planning systems and areas to which they have been applied.

6.4 Prolog vs. Fortran, etc.

At the outset, one could argue that nearly all aspects incorporated into the Prolog program could be done with traditional languages like Fortran. Indeed one could easily say it is possible to implement anything using any of the common high level languages. However, by using a logic programming language such as Prolog, it is possible to write the various relationships and heuristics much faster than is possible in a procedural language like Fortran. Because knowledge, in the form of facts and rules,

can be written in Prolog in the way people think about about them, it is a lot faster and easier to write them.

The design system achieves more than establishing the design parameters. It offers a data base environment so that sales and marketing personnel could also use them. Even with the the developed design system just as it is, it can be used by a sales staff to respond to customers' enquiries. For example, suppose a customer wanted to know if a machine is available which meets his specification. The sales person would enter the specifications, and the system would respond accordingly; as shown below.

] ?carryon.

adapt opts To modify a design

new opts To start a completely new design

save opts To save the state of the system for subsequent restart

give opts To give design details - will be prompted for goal(s)

store opts To store design details obtained sofar

Please enter option :

Type 'help' for a list of options available

option>give

Please enter your specifications for the machine desired :

(form : <parameter name> = <parameter value>)

specs >voltage=120

specs >speed=900

specs >power=22380

specs >end

There is an existing design that meets the specifications :

Design identifier : 1

Armature external diameter	**= 31.75 cm.**
Length of armature core	**= 17.78 cm.**
Terminal (rated) voltage	**= 120. V.**
Output power	**= 22. kW.**
Full load current	**= 208. A.**
Speed	**= 900. rpm**
Number of poles	**= 4.**
Flux	**= 0.0206 Wb.**
Number of paths	**= 2.**
Number of armature slots	**= 49.**
Number of turns	**= 98.**
Armature conductors per slot	**= 4.**
Slot height	**= 2.337 cm.**
Slot width	**= 1.016 cm.**
Shaft diameter or internal yoke diameter	**= 4.00 cm.**

There is no machine in stock for this design

The program could be devised so that only the essential information is given.

The point being made here is that the same data is used by a designer and sales person. It could also be used by policy makers. Better still it could be used by other engineers working on the same project. It increases an expert designer's productivity by releasing him from routine tedious jobs. The level of expertise required is lowered using such a system, so that less experienced persons could do a better job.

The most notable difference is the ability to improve the expertise of the system incrementally. One can simply keep adding expertise and experience.

6.5 Symbolic computation of algebraic equations

In design we cannot avoid algebra. With traditional procedural languages an algebraic equation having, for example, five independent parameters would have to be programmed by writing a separate routine for evaluating the value of one of the parameters when values of the remaining parameters are given. But with Prolog, a properly written symbolic solver is all that is needed. The beauty of such a solver is its generality in being applicable to all types of algebraic equations. It is interesting to note that the scheme could be extended to mathematical differentiation and integration.

There are qualitative things an expert can say, relating to what makes a good or bad design. If set out as a set of rules, this knowledge can be easily coded into Prolog.

6.6 Missing quantities

When an interactive program is written in algorithmic languages the entities which would be user inputs in response to program interrogations, are always specified. As a result, continuing with our example in electric machine design, there would be a distinct set of parameters whose values can be entered by the user interactively. Furthermore, there would be only one sequence of assigning values to the individual parameters. Thus if a program was organized so that the user entered his specifications in the order voltage, speed and power, then every time the program is run, the query/response interaction would proceed in the same order, that is, speed, voltage and power. Notice also that if any parameter other than the three above were not specified, it would cause the program to abort; indicating an error has occurred. This is because for the parameter concerned, it was not explicitly written that its value could be entered by the user.

A properly written Prolog program on the other hand, does not categorize parameters as inputs or otherwise; only when it fails to find a solution will it ask for a clue. By this technique, the user is free to specify any set of parameters. If one or more values of parameters still need to be given, the user will be required to give them. The important feature here to note is that there is nowhere in the the program where it is specifically stated that its value be requested from the user.

The provision of what has been termed "a query the user" [23] facility is undoubtedly a significant step towards a simpler man-machine interaction. It is a construct unique to languages that perform symbolic computations where if a solution cannot be obtained, the user is asked for further data.

This facility has been included in the inference engine as one way of solving for a goal parameter. It is evoked when all other methods of solution have failed. It also gives the user an option to say that the value is unknown or, interpreted in another way, to signal the system to look for other means of solving the original goal.

6.7 Standards

There are other areas of application of logic programming to machine design. One of these is the use of standards. While not yet implemented in the main design system, the author has examined the application of logic programming to "standards" as a separate design study.

During the design process, it is possible to take into account of national, international and proprietary standards coded in a form very much similar to the way they are normally stated. By way of example, we demonstrate below how easily it can be done, by considering the British Standard specification BS4999, Part 41, paragraph 41.3 specifying the starting kVA and locked rotor torque of 3-phase cage induction motors. [24].

It states:

> The starting performance (locked rotor, rated voltage conditions) of single speed, 3-phase, 50 Hz cage induction motors designed for direct-on-line starting shall be associated with six different designs of motor, A, B, C, D, E and F as classified in table 41.3.1, and shall be in accordance with the appropriate values specified in tables 41.3.1, 41.3.2 and 41.3.3. Motors having starting characteristics other than those covered by designs A to F in table 41.3.1 shall be classified as design G.

Tables 41.3.1, 41.3.2 and 41.3.3 are reproduced below.

Table 41.3.1 Classification of starting characteristics of 3-phase cage induction motors

Design	Starting kVA (locked rotor) not to exceed	Locked rotor torque not to be less than
A	Column 1 of table 41.3.2	Values in table 41.3.3
B	Column 2 of table 41.3.2	Values in table 41.3.3
C	Column 2 of table 41.3.2	Not less than twice per unit (p.u.) torque, up to 630 kW rated output
D	Column 3 of table 41.3.2	Values in table 41.3.3
E	Column 3 of table 41.3.2	0.75 p.u. values in table 41.3.3
F	Column 1 of table 41.3.2	1.5 p.u. values in table 41.3.3
G	Subject to agreement	

Table 41.3.2 Starting (locked rotor) kVA for 3-phase cage induction motors

Rated output (kW)			Ratio † starting (locked rotor) kVA / rated output kW		
			Column 1	Column 2	Column 3
Over 1.0	up to	2.5	In excess of ratios given in column 2	10.5	
2.5	up to	6.3		9.8	
6.3	up to	16		9.2	
16	up to	40		8.7	
40	up to	100		8.2	6.2
100	up to	250		7.8	6.0
250	up to	630		7.6	5.8
630	up to	1 600		7.4	5.6
1 600	up to	4 000		7.2	5.4
4 000	up to	10 000		7.0	5.2

† To obtain the ratio of starting (locked rotor) current to rated load current, multiply this ratio by per unit efficiency and power factor at rated load.

Table 41.3.3 Locked rotor torque for 3-phase cage induction motors

Rated output (kW)	Locked rotor torque (per unit values)					
	Number of poles					
	2	4	6	8	10	12
Over 1.0 up to 2.5	2	2	1.75	1.5	1.5	1.15
2.5 up to 6.3	2	2	1.75	1.4	1.3	1.15
6.3 up to 16	2	2	1.75	1.4	1.2	1.15
16 up to 40	1.6	1.7	1.6	1.4	1.15	1.1
40 up to 100	1.25	1.4	1.3	1.25	1.15	1.1
100 up to 250	1.0	1.1	1.2	1.2	1.1	1.1
250 up to 630	0.8	0.8	0.8	0.8	0.8	0.8
630 up to 1600	0.6	0.6	0.6	0.6	0.6	0.6
1600 up to 4000	0.5	0.5	0.5	0.5	0.5	0.5
4000 up to 10000	0.5	0.5	0.5	0.5	0.5	0.5

One way of representing the above standard specification in Prolog, would be as follows:

```
/* Code for Table 41.3.1 - Classification of starting characteristics of 3-phase
/* induction motors. Two predicates are used:
 /* (1) type(Arg1, Arg2, Arg3, Arg4, Arg4, Arg5, Arg6)
/* Arg1 = Letter for the design type
/* Arg2 = Actual (calculated) value of the torque for a machine in question
/* Arg3 = Actual (calculated) value of locked rotor kVA
/* Arg4 = Specified minimum torque as given in Table 41.3.3
/* Arg5 = Appropriate values of column 3 in table 41.3.2
/* Arg6 = Appropriate values of column 2 in Table 41.3.2

 type('A',X,Y,Z1,Z2,Z3) :-
     column(1,Z1,Z2,Z3),
    X >= Y.

 type('B',X,Y,Z1,Z2,Z3) :-
     column(2,Z1,Z2,Z3),
    X >= Y.

 type('D',X,Y,Z1,Z2,Z3) :-
     column(3,Z1,Z2,Z3),
    X >= Y.
```

```
type('E',X,Y,Z1,Z2,Z3) :-
    column(3,Z1,Z2,Z3),
   X >= Y*1.50 .

type('F',X,Y,Z1,Z2,Z3) :-
    column(1,Z1,Z2,Z3),
   X >= Y*1.50 .

type('G',X,Y,Z1,Z2,Z3).

/* (2) column(Arg1, Arg2, Arg3, Arg4)
/*  Arg1  =  Relevant column number
/*  Arg2  =  Actual (calculated) locked rotor kVA
/*  Arg3  =  Locked rotor kVA in column 3, Table 41.3.2
/*  Arg4  =  Locked rotor kVA in column 2, Table 41.3.2

column(1,X,Y,Z) :-
   X >= Z.

column(2,X,Y,Z) :-
   number(Y),
   X > Y,
   X < Z.

column(2,X,Y,Z) :-
   X < Z.
```

```
column(3,X,Y,Z) :-
number(Y),
   X < Y.


/* Table 41.3.3 : Locked rotor torque for 3-phase cage induction motors
/* locktor([Elm1, Elm2, Elm3,...,Elmn])
/* "locktor" has one argument which is a list. The first and second elements are the
/* upper and lower bounds of relevant power range. They are followed by several
/* two-elements sub-lists of the form [Number_of_poles, Minimum_starting_torque]

locktor([1  , 2.5,[2,2]  ,[4,2]  ,[6,1.75],[8,1.5] ,[10,1.5]  ,[12,1.15]]).
locktor([2.5, 6.3,[2,2]  ,[4,2]  ,[6,1.75],[8,1.4] ,[10, 1.3] ,[12,1.15]]).
locktor([6.3, 16 ,[2, 2] ,[4,2]  ,[6,1.75],[8,1.4] ,[10, 1.2] ,[12,1.15]]).
locktor([16 , 40 ,[2,1.6],[4,1.7],[6,1.6] ,[8,1.4] ,[10, 1.15],[12,1.1]]).
locktor([40 ,100 ,[2,1.25],[4,1.4],[6,1.3] ,[8,1.25],[10,1.15] ,[12,1.1]]).

/*  Code for Table 41.3.2

lockvars(1  , 2.5 , na  , 1.05E1).
lockvars(2.5, 6.3 , na  , 9.8).
lockvars(6.3, 16  , na  , 9.2).
lockvars(16 , 40  , na  , 8.7).
lockvars(40 , 100 , 6.2 , 8.2).
lockvars(100, 250 , 6.0 , 7.8).
```

```
/* Using the above rules and facts, one could extract any information needed.  Sample
/* programs that could help in that have been written:


/* Specify  minimum pull out torque for a machine for which the output power and
/* the number of poles are known.
pull_out_torque :-
    write('Enter value of power '),
    read(Y),
    write('Enter number of poles '),
    read(Z),
    torat1(Y,Z).


torat1(X,Y) :-
    locktor([Z|[_x|_y]]),
    X > Z,
    X < _x,
   member([Y,_z],_y),
   write('locked rotor torque (per unit values) is '),
    write(_z),nl.
```

```
/* Establish the design category of a machine, given its output power, the number of
/*  poles,  calculated per unit locked torque and kVA

design_type :-
    write('Give value of power :'),
    read(Y),
    write('Give number of poles :'),
    read(Z),
    write('Give p . u . locked torque :'),
    read(_x),
    write('Give p . u . starting kVA :'),
    read(_y),
    locktor([_z|[X1|Y1]]),
    Y >= _z,
    Y < X1,
    member([Z,Z1], Y1),
    lockvars(_z,X1,_x1,_x2),
    type(_y1,_x,Z1,_y,_x1,_x2),
    write('The specifications correspond to design type '),
    write(_y1),nl.
```

Following here are some program runs for extracting the required information from the specifications.

] ?design_type.

Give value of power :10.

Give number of poles :6.

Give p . u . locked torque :1.82.

Give p . u . starting kVA :9.4.

The specifications correspond to design type A

Yes

] ?design_type.

Give value of power :10.

Give number of poles :2.

Give p . u . locked torque :1.82.

Give p . u . starting kVA :9.4.

The specifications correspond to design type G

Yes

] ?pull_out_torque.

Enter value of power 30.

Enter number of poles 4.

locked rotor torque (per unit values) is 1.7

Yes

Another example is BS 5000 : Part 10 : 1978 - Section 6 : specification for rated output and shaft number for totally enclosed fan-ventilated or airstream rated cage induction motors [25]. For brevity, a sample of the specification concerning frame size is given below

Frame number	3000 rpm	1500 rpm	1000 rpm	750 rpm
D56	0.09 and 0.12	0.06 and 0.09	-	-
D63	0.18 and 0.25	0.12 and 0.18	-	-
D71	0.37 and 0.55	0.25 and 0.37	-	-
D80	1.1	0.75	0.55	-
D90S	1.5	1.1	0.75	0.37
D90L	2.2	1.5	1.1	0.55

Output values are in kW.

The above is represented as follows:

```
tabu(['D56' , [300,  0.09, 0.12], [1500,0.06,0.09]]).
tabu(['D63' , [3000, 0.18, 0.25],[1500, 0.12, 0.18]]).
tabu(['D71' , [3000, 0.37, 0.55],[1500, 0.25, 0.37]]).
tabu(['D80' , [3000, 1.1],  [1500, 0.75],[1000, 0.55]]).
tabu(['D90S', [3000, 1.5],  [1500, 1.1], [1000, 0.75],[750, 0.37]]).
tabu(['D90L', [3000, 2.2],  [1500, 1.5], [1000, 1.1], [750, 0.55]]).
```

Suppose we have predicate "frm3" which should advise on the frame size to use given the output power and speed for a machine for which the design has been completed, and predicate "giv" which states the maximum power specified for a given frame size and motor speed:

```
frm3(X,Y) :-
  tabu([Z|_x]),
  member([X|_y],_x),
  member(Y,_y),
  write(X),spaces(4),write(Y),spaces(6),write('the frame size  is '),
  write(Z),nl.


giv(X,Y) :-
  tabu([X|Z]),
  member([Y|_x],Z),
  write(X),spaces(4),write(Y),spaces(6),write('output  power  is '),
  [_x1|[]] = _x,write(_x1),spaces(2),write('kW'),nl.
```

When these programs are run we would get something like this:

```
] ?frm3(1500,0.18).

1500   0.18     the frame size  is D63

Yes

] ?frm3(1000,1.1).

1000   1.1      the frame size  is D90L
```

Yes

] ?giv('D90L',3000).

D90L 3000 output power is 2.2 **kW**

Yes

] ?

So we can get any desired information contained in the specification.

The two examples illustrate that the process of checking whether or not a design conforms to given standards can be incorporated directly into the design system.

Chapter 7

DISCUSSION , FUTURE WORK AND CONCLUSION

7.1 Discussion

To recapitulate, it has been shown that there is a great deal more to be done to achieve satisfactory computer systems for design purposes in general, and in particular, electrical machine design. It was mentioned that one of the main obstacles has always been on how to code expert knowledge which it is argued is a very important matter of concern in the design industry. This was a motivation to this work ; to explore the use of logic programming in electric machine design.

Accordingly, the initial part of the project involved learning the use of Prolog and in the process figuring out or translating its working mechanism in relation to design. Eventually a simple model of the design process was conceived based on ideas by earlier researchers in artificial intelligence. The ensuing work laboured on the question of representation of the actual knowledge which is used in making a design. This proved to be the bottleneck.

The representation scheme adopted is thoroughly described. It is shown that using logic programing techniques, coding the design process can be easily carried out. These same techniques can be used for other design procedures not necessarily involving electrical machines.

The unique capabilities associated with logic programing are opening up a whole host of engineering applications in general and design in particular. The application of logic programing, using Prolog, shows several promising lines for the future. What may not have been achieved here has to do more with the limited experience of the writer rather than with the limitation of the language used. Logic programing languages are new compared with other common higher level languages including its rival - Lisp. Simplifications and improvements to the various dialects of Prolog are presently the preoccupations of implementers. Enhancements are coming about to make them general programing languages which meet a variety of applications requirements. We are thus helping to pave the way for the future.

This work addresses itself to important real life problems which when successfully tackled would revolutionize, not only the way designers do things, but also companies as a whole. We think we have given some useful pointers which will spur more research in this direction.

7.2 Future work

The more interesting concepts have not been implemented largely a result of a deliberate strategy to concentrate on finding out how far one could go in building a domain independent inference engine and applying it to a relatively modest engineering design task.

Much work still needs to be done. In particular the following issues need to be addressed:

7.2.1 Explanations

Unlike the more familiar traditional high level languages, Prolog allows for various modes of explanation. It is possible to include a feature for giving explanations to questions like, "how were the results arrived at?". Another kind of possible explanation relates to a "why" type of question. This one is immensely useful in design. Frequently, an engineer would want to know what would happen if some change is made to a design. Suppose as a result the system fails to arrive to a suitable design and reports accordingly. The designer would like to get some kind of explanation for the cause of the failure. If on the other hand a solution is obtained, a comment may be called for concerning, for example, special manufacturing or operation implications; such as the high cost of manufacturing a machine to that design.

A suitably built system with explanation facilities, would provide quick access to expert knowledge by trainees and, in turn, reduce training requirements. It should not be overlooked that in many countries, the number of machine design experts is very small and declining. In some countries there are none.

Locating sources of failures: In situations where the user poses the problem after which the system responds with a "No", the user may wish to know where failures originated from. This feature could be used as part of the rule base development tool.

More investigation is also needed into choice making strategies. As yet there is no formal description of the idea on which a choice is made.

7.2.2 Speed

Another matter of concern for a practical system is speed. Designers want to get results quickly so that they may be able to explore other options. With the implementation described, although the speed of execution measured in terms of response time is not prohibitively slow, it might not be fast enough for commercial usage involving large amounts of data. This is an important consideration if one is planning to use a personal computer. There are ways of improving the speed. One of them is what is known as program transformation; a process of transforming a given knowledge representation to one which can be used more efficiently and, by implication, run faster.

7.2.3 Checks for consistency and completeness

As it is now, it is assumed that the knowledge base is complete and does not contain conflicting facts or rules. By complete is meant all the necessary facts and rules are given. The worst that can happen for an incomplete knowledge base, is for the system to fail to arrive to an answer for which the solution exists. Solving goals with a conflicting or an inconsistent knowledge base, could give rise to unpredictable results. It is therefore necessary for future work to incorporate checks for consistency and completeness.

7.2.4 Interfaces to other packages

Increasingly, it is not possible for one to use packages in isolation. One has to use them with other packages as part of the design system. To cite one particular example, a great deal of effort has been put into developing numerical techniques for the computation of fields. Most noticeable of these is the finite element method. Packages are now available which can be used to predict electromagnetic field distributions in machine structures . One would like to pull out the the appropriate design data and input it into a finite element mesh generator.

In addition, problems arise with the use of packages. For they demand the user to have a good knowledge of their workings and the ability to draw satisfactory conclusions from a massive amount of output. In most cases the accuracy of such computations are only as good as the users themselves. The use of such packages could be eased by employing knowledge based systems. Some work in this direction has been recently reported [26]; using systems written in Lisp.

7.2.5 Simplification of data and rule input

Future work will also involve adoption of the use of techniques mentioned in the earlier chapters to make it easier for the industrial designer to use. The first thing the designer would like to do is enter another rule or a new table, and he would like to do that easily.

7.2.6 Applications in other areas

The area of design can be extended further to establishing rule bases for transformers, induction machines, etc. It can be used for other areas still further out, for example, in mechanical design.

7.2.7 Robustness:

The system is expected to be robust by being able to cope with aberrant input. It should reject queries, or answers as the case may be, which the system knows to be invalid. This has been included in the program to a very small degree because the main effort was directed towards the fundamental matters necessary for the construction of the inference engine.

7.2.8 Integration in a CAE system

There is an obvious need to make the design process an integral part of entire manufacturing process and not just an unimportant adjunct to it. Pushing this idea further, the engineering design of electrical machines should also be concerned with their context (environment, etc) and their operation and lifetime maintenance.

In order to address these problems, a great deal more work needs to be done in a number of associated areas, for example, relational data base design systems; optimisation; synthesis; monitoring systems and diagnostics. In all of these areas we expect knowledge based systems to be used extensively.

7.2.9 Redesigning

Redesigning of the machine, or more strictly re-planning, facility adds considerable flexibility to the use of the system. Further, the system becomes truly interactive. At some point during the design process a designer may wish to change the value of a parameter that has been established. Or he may wish to change the rules which were used to establish the parameter. The system has then to redesign, not from the start, but from the point where the change was introduced. At present this conflicts the logic of the program. Because as it is now once a goal has be achieved it can not be undone; it becomes protected.

7.3 Conclusion

This work has been concerned with a study to establish whether or not logic programming can be used to advantage in design, specifically rotating electric machine design. There were many who believed that the use of logic programming in design was not possible. On the contrary, in this thesis it has been shown that by careful system design, it is possible to use logic programming to great effect in building up a knowledge based system for design. It is in effect a "designer's assistant" or apprentice. It is not claimed that the expert designer can be replaced. There must always be human intelligence somewhere in the process. The simple examples given only hint at the great potential power of this branch of artificial intelligence. Using the system developed and described here as a base, it is possible to extend the ideas to accommodate most of the design domains because the main control program is in fact a design shell.

There is no doubt in the author's mind that the technique is an extremely powerful one with great potential. It is hoped that the work described here will be accepted as a significant addition to the corpus of engineering knowledge.

REFERENCES

[1] Chalmers B J & Bennington B J; "Digital-computer Program for Design Synthesis of Large Squirrel-cage Induction Motors", Proc. IEE, Vol 114, No.2, Feb. 1967.

[2] Sharpley W A & Oldfield J V ; "The Digital Computer Applied to the Design of Large Power Transformers"; Proc. IEE, pp 112 - 121, Aug. 1957.

[3] Veinott C G; "Induction Machinery Design Being Revolutionalised by Digital Computer"; Trans. Amer. Inst. Elec. Engnrs., Vol. 75, Pt. 3, pp 1059, 1956.

[4] Mouly V S R K & Rao G S; "Economic Design of Small Induction Motors", Electric Machines and Power Systems, Vol. 8, pp 123 - 130, 1983.

[5] Jozefowicz W; "Global Optimisation with Clustering Analysis for CAD of Electrical Machines"; 2nd Conf. on Electrical Machines - Design and Applications, London, 17 - 19 Sept. 1985.

[6] Lowther D A & Saldanha C M; "A Frame-based System for the Design of Electromagnetic Devices"; IEEE Trans. on Magnetics Vol MAG-22, No.5, pp 814-816, Sept 1986 / Intermag'86, Phoenix, Arizona, April 14-17, 1986.

[7] "Magnet11" Reference Manual

[8] Kowalski R A; "Logic for Problem Solving"; North-Holland, 1979.

[9] Kowalski R A; "Logic for Knowledge Representation" Imperial College of Science and Technology, Department of Computing.

[10] Melvin, S K; "Artificial Intelligence for Engineering Design"; IEE CAE- J pp 75 - 83, April 1984.

[11] McDermott J; "R1: An Expert in the Computer Systems Domain" Proceedings of the First Annual National Conference on Artificial Intelligence, pp 269-271; 1980.

[12] Duda R O, Gaschnig J G & Hart P E; "Model Design in Prospector Consultant System for Mineral Exploration"; Expert Systems in Microelectronic Age, Edinburgh University Press, pp 153 - 167.

[13] Feigenbaum E A, Buchanan B G & Lederberg; "On Generality and Problem Solving: A Case Study Using The DENDRAL Program"; Machine Intelligence 6, New York: American Elservier, pp 165 - 190, 1971.

[14] Shortliffe E H; "Computer Based Medical Consultations: MYCIN"; New York: Elsevier, 1976.

[15] Kowalski T J & Thomas D E; "The VLSI Design Automation Assistant: What's in knowledge Base" 22nd Design Automation Conference, Paper 18.1

[16] Kim J H, McDermott J & Siewiorek D P; "Exploiting Domain Knowledge in IC Cell Layout"; IEEE Design & Test, pp 52 - 64, 1884

[17] Sergot M J, Sadri F, Kowalski R A, Kriwaczek F, Hammond P, & Cory H T; "The British Nationality Act as a Logic Program"; Communications of the ACM, Vol. 29, No. 5, pp 370 - 386; May 1986.

[18] Clocksin W.F. & Mellish C.S.; "Programming in Prolog" Springer Verlang, 1981.

[19] Warren D.H.D., "A System for Generating Plans", Memo 76, Department of Computational Logic, University of Edinburgh, June 1974.

[20] Bailey D.; "The University of Salford Lisp/Prolog Reference Manual"; University of Salford; March 12, 1984.

[21] Gray, A. Electrical Machine Design, McGraw-Hill Book Co., Inc.; 1933.

[22] Winston P H & Horn B K P; "Lisp"; Addison-Wesley; 1981.

[23] Sergot M; "A Query-the-user Facility for Logic Programing"; Imperial College, Department of Computing - Research Report DOC 82/18; 1982.

[24] British Standards Institution; "General Requirements for Rotating Electrical Machines"; BS 4999; Part 41 (General Characteristics); 1977.

[25] British Standard BS 5000 : Specification for Rotating Machines of Particular Types or for Particular Applications. Part 10 : General Purpose Induction Motors : 1978.

[26] Lowther D A, Saldanha C M & Choy G; "The Application of Expert Systems to CAD in Electromagnetics"; COMPUMAG, Colorado, pp483-486, June 1985 / IEEE Trans. on Magnetics, Vol MAG-21, No.6, pp 2559-2562, Nov. 1985.

[27] O'Shea T & Eisenstadt M (eds); "Artificial Intelligence - Tools, Techniques, and Applications"; Harper & Row, Publishers, NY.; 1984.

[28] Luger G F; "Means End Analysis and the Solution of Mechanics Problems"; University of Edinburgh; D.A.I. Research Report, No.143, July 1980.

[28] Austin Tate, "INTERPLAN: A Plan Generation system which can deal with Interactions Between Goals", Memorandum MIP-R-109 MIR, Edinburgh University, December 1974.

[29] Winston P H; "Artificial Intelligence"; Addison-Wesley; 2e; 1984.

APPENDIX A

WARREN'S PLANNNER - WARPLAN

Definitions of operators "&" and "=>"

& : operator by which a conjunction of terms are linked together, and

=> : also an operator but with a different precedence.

WARPLAN uses the following predicates to represent facts and rules in the data base for the domain knowledge:

add(X,U)

Fact X is added by action U ; i.e. X is true in any state resulting from U (and U is a possible action in some state in which X is not true).

del(X,U)

Fact X is deleted by action U; ie. it is not the case that X is preserved by U. (X is preserved by U if and only if X is not added by U and X is true in a state resulting from U whenever X is true in the preceding state)

can(U,C)

The conjunction of facts C is the precondition of action U , i.e. U is possible in any state in which C is true.

always(X)

Fact X is true in any state.

impossX)

The conjunction of facts X is impossible in any state.

given(T,X) :

Fact X is true in the initial state T (but it is not the case that F is true in all states).

The Planner

```
plans(Goals,Initial_plan) :-
    not(consistent(Goals,true)),!,
    nl, write(impossible), nl, nl.

plans(Goals,Initial_plan) :-
    plan(Goals,true,Initial_plan,Final_plan),
    nl,
    output(Final_plan), nl, nl.


output(Remainder => Last_action) :- !,
    output1(Remainder),
    write(Last_action),
    write(.), nl.
```

```
output1(Other_actions => One_action) :- !,
    output1(Other_actions),
    write(One_action),
    write(;), nl.

output1(One_action) :-
    write(One_action),
    write(;), nl.
```

plan(C,P,T,T1) is a procedure entry point to the main recursive loop; where,

C is the conjunction of goals to be solved

T is an already generated partial plan

P is a conjunction of goals already solved by Arg_3 which must be protected.

T1 is a new plan, which contains Arg_3 as a subplan and preserves the already solved goals Arg_2, and also which solves the new goals Arg_1.

Essentially, the routine states that a plan can be produced by "solve"ing each goal in the order given.

```
plan(Agoal & Goals,Protected_goals,Old_plan,Final_plan) :- !,
    solve(Agoal,Protected_goals,Old_plan,Protected_goals_agoal, New_plan),
    plan(Goals,Protected_goals_agoal,New_plan,Final_plan).

plan(Agoal,Protected_goals,Old_plan,New_plan) :-
    solve(Agoal,Protected_goals,Old_plan,Protected_goals_agoal,New_plan).
```

solve(X,P,T,P1,T1) succeeds when X is solvable; where,

X is an atomic goal,

T is a partial plan,

P is a conjunction of goals achieved by T,

T1 is a plan, containing T as a subplan, which solves P1,

P1 is a conjunction comprising P and X, where X is not repeated.

Atomic goal X is solvable either by establishing that it is "always" true, or that it already "holds" in the state produced in the current plan, or by achieving an action U in the data base, if any, which as a result "add"s X to the list of solved goals.

```
solve(Agoal,Protected_goals,Old_plan,Protected_goals,Old_plan) :-
    (always(Agoal) ; Agoal).

solve(Agoal,Protected_goals,Old_plan,Protected_goals_agoal,Old_plan):-
    holds(Agoal,Old_plan),
    and(Agoal,Protected_goals,Protected_goals_agoal).

solve(Agoal,Protected_goals,Old_plan,Agoal & Protected_goals,New_plan):-
    add(Agoal,Action),
    achieve(Agoal,Action,Protected_goals,Old_plan,New_plan).
```

achieve(X,U,P,T,T1) will first attempt to achieve the goal X by finding the preconditns of the action U, generating a subplan to satify them, and then appending this subplan and U onto the original plan T to produce produce a new plan T1. If this fails, it will try to insert the action somewhere in the sequence of actions T, along with any preconditions required to achieve it, by recursively stripping off the last

action of T testing to see if this action deletes the goal X, and then trying to "achieve" U with the new plan.

```
achieve(Agoal,Action,Protected_goals,Old_plan,T1 => Action) :-
    preserves(Action,Protected_goals),
    can(Action,Conditions),
    consistent(Conditions,Protected_goals),
    plan(Conditions,Protected_goals,Old_plan,T1),
    preserves(Action,Protected_goals).

achieve(Agoal,Action,Protected_goals,Old_plan_1 => Last_action,T1 =>
      Last_action) :-
    preserved(Agoal,Last_action),
    retrace(Protected_goals,Last_action,P1),
    achieve(Agoal,Action,P1,Old_plan_1,T1),
    preserved(Agoal,Last_action).
```

Everything that "holds" in a state of the world can be determined from the plan which produced that state. The system chains back through the sequence of actions, so long as none of these actions deletes the sought for fact, untill the fact is found in the add set of an action or was "given" in the initial state.

```
holds(Agoal,Old_plan_1 => Last_action) :-
    add(Agoal,Last_action).
```

```
holds(Agoal,Old_plan_1 => Last_action) :- !,
    preserved(Agoal,Last_action),
    holds(Agoal,Old_plan_1),
    preserved(Agoal,Last_action).

holds(Agoal,Initial_plan) :-
    given(Initial_plan,Agoal).
```

preserves(U,P) checks to see if a conjunction of goals P is preserved by an action U.

```
preserves(Action,Agoal & Protected_goals) :-
    preserved(Agoal,Action),
    preserves(Action,Protected_goals).
preserves(Action,true).
```

preserved(X,U):To prove that a fact X is "preserved" by an action U show that it is not possible to prove that action U deletes X.

```
preserved(Agoal,Action) :-
    ground(Agoal & Action,0,N),
    del(Agoal,Action),!,
    fail.
preserved(Agoal,Action).
```

If V is the last action in the current plan, and we remove it, then we can "retrace" to the set of protected at the point before V. P is "retrace"d to P1 before V if, P1 = P minus(addset of V) plus(preconditions of V)

```
retrace(P,V,P2) :-
    can(V,C),
    retrace1(P,V,C,P1),
    &append(C,P1,P2).
```

"retrace1" removes addset of V from original set of facts

```
retrace1(X & P,V,C,P1) :-
    add(Y,V),
    equiv(X,Y),!,
    retrace1(P,V,C,P1).

retrace1(X & P,V,C,P1) :-
    elem(Y,C),
    equiv(X,Y),!,
    retrace1(P,V,C,P1).

retrace1(X & P,V,C,X & P1) :-
    retrace1(P,V,C,P1).
retrace1(true,V,C,true).
```

Given a set of proven facts P and a set of facts C yet to be proved; C is said to be an inconsistent set of goals by proving that a known inconsistent set of facts S is a subset of the union of sets C and P. Ofcourse in order the proof makes sense, it must be the case that some elements in C are contained in P. After all inconsistency checks would have been done in the course of proving facts P.

```
consistent(Conditions,Protected_goals) :-
    &append(Conditions,Protected_goals,Conditions_goals),
    ground(Conditions_goals,0,N),
    imposs(Imposs),
    not(not(intersect(Conditions,Imposs))),
    implied(Imposs,Conditions & Protected_goals),!,
    fail.
consistent(Conditions,Protected_goals).
```

Utility routines:

```
/* '&append'(X,Y,Z) : append argument Y to argument X to yield argument  Y, X and
Z
/* are of the form F1&F2&F3 .....
'&append'(X1&X, Y, X1&Y1) :- ! , '&append'(X, Y, Y1).
 '&append'(X1, Y, X1&Y).

'&extract'(_term, _term & _other_terms, _other_terms).
'&extract'(_term, _first_term & _other_terms, _first_term & _remaining_terms)
:-
'&extract'(_term, _other_terms, _remaining_terms).
'&extract'(_term, _term, true).

/* implied( S, C ): A conjunction of terms (facts) S is implied by a conjunction of
/* facts C if and only if each of the terms in S is either contained in C or exists in the
/* data base.
    implied( S1&S2, C ) :- ! , implied( S1, C ), implied( S2, C ).
    implied( X, C ) :- elem( X, C ).
    implied( X, C ) :- X .

/* elem( X, Y ): fact X is contained in the conjunction of facts Y
/*  e.g. : elem( voltage(200), speed(900) & voltage(200) & turns(150) )
    elem(X,Y&C) :- elem(X,Y).
    elem(X,Y&C) :- !, elem(X,C).
    elem(X,X).
```

```
intersect(Conjunction_1, Conjunction_2) :-
  elem(Aterm, Conjunction_1),
  elem(Aterm, Conjunction_2).

equal( X, X ).
notequal( X, Y ) :- not(equal( X, Y )), '&nonvars'(X), '&nonvars'(Y).
'&nonvars'(X) :- X =.. X1 , contains_vars(X1,X2), X2 = [].
equiv(X,Y) :-  not(nonequiv(X,Y)).

nonequiv(X,Y) :- ground(X&Y,0,N), X = Y ,!, fail.
nonequiv(_,_).

ground( X&Y, Z, _y) :- ! ,groundatom(X,Z,_x), ground(Y,_x,_y), ! .
ground(X,_x,_y) :- groundatom(X,_x,_y), ! .
groundatom(X,_x,_y) :- X =.. [X1|Y] , groundvar(Y,_x,_y),! .

groundvar([],_z,_z).
groundvar([X|Y],_x,_y) :- var(X),_z is _x+1,X = _z ,groundvar(Y, _z, _y).
groundvar([Y|Z], _x, _y) :- groundvar(Z, _x, _y).

/* and( X, Y, Z ) : A conjunction of facts Z composed of a term X and a conjunction of
/* terms Y can be derived if :  X is in Y,  so that Z equals Y;  or X and Y can be joined
 /* together to give Z.
    and(X,P,P) :- elem(Y,P), equiv(X,Y),! .
    and(X, P, X&P).
```

```
/*  append(X,Y,Z) : append list Y to X to give list Z
    append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
    append([],L,L).


reverse(L1,L2) :-  reverse1(L1,[],L2).
reverse1([X|L],L2,L3) :-  reverse1(L,[X|L2],L3).
reverse1([],L,L).

/* in_between(Value,Lower_limit,Upper_limit) : Holds if a given <Value> lies
between /* the lower and upper limits. Each of the arguments may be a number or a
numeric
/* expresion.
   in_between(Value,Lower_limit,Upper_limit) :-
       Value1 is Value,
       Value1 >= Lower_limit,
       Value1 =< Upper_limit.


last([],[]).
last(List,Last_element) :-  last1(List,Last_element),!.
last1([Last_element],Last_element).
last1([Head|Tail],Last_element) :-  last1(Tail,Last_element).

/* assign(A,N) : set each element in list A to a corresponding element in list N
    assign([X|X1],[Y|Y1]) :- %('SETQ'(X,Y)), assign(X1,Y1).
   assign([],[]).
```

```
/* extract(_element,_list,_remainder): procedure to extract an element from a list
   extract(_element,[_element|_tail],_tail).
   extract(_element,[_head|_tail],[_head|_remainder]) :-
         extract(_element,_tail,_remainder).

/* Delete all occurences of an element
   delete(Element,Given_list,Resulting_list) :-
      mode(nodebug,Mode),
      delete1(Element,Given_list,Resulting_list),
      mode(Mode),!.
   delete1(_,[],[]).
   delete1(X,[X|L],M) :- !,delete(X,L,M).
   delete1(X,[Y|L1],[Y|L2]) :- delete(X,L1,L2).

/* removes all elements of the first set from the second to generate the third
   remove([X|A],B,C) :- delete(X,B,B1),remove(A,B1,C).
   remove([],B,B).

/* try(Goal,Index,Given_list,True_list, False_list): From a given list of elements
/* <Given_list> , obtain a list of elements <True_list> each of which make <Goal> to
/* succeed. Each element in <False_list> fails when applied to the goal.

    try(Goal,Index,Given_list,True_list,False_list) :-
       try1(Goal,Index,Given_list,True_list,False_list),!.
   try1(Goal,Index,[],[],[]).
```

```
try1(Goal,Index,[Given_one|Given_rest],[Given_one|True],False) :-
        substitute(Goal,Index,Given_one,The_goal),
        call(The_goal), !,
        try1(Goal,Index,Given_rest,True,False).
    try1(Goal,Index,[Given_one|Given_rest],True,[Given_one|False]):-
        try1(Goal,Index,Given_rest,True,False).

/* Predicate 'any' defined below is used when its argument is of the form
/* <Predicate(A&B&C,X1,X2,X3,.....)> where A,B, etc, are any number of simple
/*  terms.
/* Uninstanciated variables for each term in the conjunction will be instanciates
/* when  the predicate for which it is the argument, succeeds; otherwise the variables
/* remain uninstanciated

    any(Term) :-
        Term =.. [Pred,Conjunct|Others],
        contains_vars(Others, Var_list),
        apply(substitute(Others,X,'_',New_term), X, Var_list),
        any1(Pred,Conjunct,Others,Var_list).

    any1(Pred,X1&X,Others,Var_list) :-
        U =.. [Pred,X1|Others],
        U,!,
        length(Var_list,L),
        length(Dummy,L),
        apply(substitute(Dummy,X,'_',New_dummy), X, Dummy),
        any1(Pred,X,Dummy,Var_list).
```

```
   any1(Pred,X1&X,Others,Var_list) :- any1(Pred,X,Others,Var_list).
   any1(Pred,X,Others,Var_list) :- U =.. [Pred,X|Others], U,!.
   any1(Pred,X,Others,Var_list).

contains_atoms(X,Y) :- cc(X,[],X1),!,setof(U,member(U,X1),Y).
cc([],_list,_list).
cc([X|Y],_inlist,_outlist) :- cc1(X,_inlist,_11), cc(Y,_11,_outlist).
cc1(X,_list,[X|_list]) :- atomic(X),!.
cc1(X,_list,_list) :- var(X),!.
cc(X,_inlist,_outlist) :- X =.. [H|Z], cc(Z,_inlist,_outlist).

/* ladd(_term): Logical addition of a term to working data base
   ladd(_term) :- asserta(_term).
   ladd(_term) :- retract(_term), fail.

/* ldel(_term): Logical deletion of a term from working data base
   ldel(_term) :- retract(_term).
   ldel(_term) :- asserta(_term), fail.

/* loop(Start_value,Final_value,Goal): For each count made from <Start_value> to
/* <Final_value>, solve <Goal>
   loop(M,N,Goal) :- Goal, X is M+1 , X =< N , loop(X,N,Goal).
   loop(_,_,Goal).
```

```
/* operator  =... usage :  List  =...  Conjunction (with connective '&').
   ([X|Y]  =... X&Z ) :- Y =... Z ,!.
   ([X] =... X).

/* is_arg(W,X,Y,Z):Suceeds if Y is argument number W of term Z whose user defined
/* predicate name is X. Mainly intended for a situation where it is required to find out
/* an existence of a term Z which is in the form of an assertion or a  head of a clause
/* whose predicate X has Y as its W-th argument.
/* Note here only the predicate name is known and not the form of the clause -- W, X
/* and Y are inputs and Z is the output. On the other hand if W and Y are uninstanciated
/* with X and Z instanciated, alternative results will be obtained on backtracking.
   is_arg(N_th,Pred_name,Arg_name,Term) :-
       user_defined(Pred_name,No_arguments),
       functor(Term,Pred_name,No_arguments),
       clause(Term,Tail),
       arg(N_th,Term,Arg_name).

/* is_clause(W,X,Y,Z): Suceeds if Y is argument number W of term Z whose user
/* defined predicate name is X. Mainly intended for a situation where it is required to
/* find out an existence of a term Z which is in the form of an assertion or a  head of a
/* clause  whose predicate X has Y as its W-th argument.
/* Note here only the predicate name is known and not the form of the clause -- W, X
/* and Y are inputs and Z is the output. On the other hand if W and Y are uninstanciated
/* with X and Z instanciated, alternative results will be obtained on backtracking.
```

```
is_clause(Term) :-
    Term =.. [Pred_name|Arguments],
    user_defined(Pred_name,No_arguments),
    functor(Term_a,Pred_name,No_arguments),
    Term = Term_a .
```

APPENDIX B

Built-in Lisp functions:

ABS	ACOS	ADD1	ADDPROPG	AND
APPEND	ARRAY	ASIN	ASSERTA	ASSERTZ
ASSOC	ASTRIP	ATAN	ATOM	ATRIM
BACKTRACE	BINAND	BINOR	CAAR	CADR
CALL	CAR	CASE	CDAR	CDDR
CDR	CGO	CHAR	CISSUE	CLEAR
CLEAR_SCREEN		CLOSEUNIT	CO	COMMAND_LOOP
COMOUT	COMPILE	COND	CONS	CONTINUE
COPY	COS	CURRENTOUTPUT		CURSOR
DATE_TIME	DEFBREAK	DEFUN	DELETE	DIFFERENCE
DIVIDE	DNEXT	DRAWTO	DRUN	DSKOUT
DUMP	ED	EDIT	EQ	EQUAL
ERROR	EVAL	EVERYATOM	EXP	EXPLODE
EXPT	EXTRACT	FAPPLY	FILEDELETE	FILEEXISTS
FILEOPEN FIX	FOR	FORMPRINT	FROM_BYTES	FUNCALL
GC	GCD	GE	GENSYM	GET
GETFILEPOSITION		GETPDEF	GETPLIST	GETPROP
GETTO	GETUNIT	GO	GREATERP	GT
HASH	HASHPOS	HASHTEXT	HELP	ICHAR
IF	IFG	IMPLODE	INDEX	INTEGERP
INTERNAL_FILEOPEN		INTO_BYTES	INWAIT	LAMBDA
LAST	LCASE	LDIFFERENCE	LE	LENGTH

LESSP	LF	LIBRARY	LINETO	LINTERSECTION
LISP_COMMAND		LISPINDENT	LISP_LIST	LIST
LISTP	LO	LOAD	LOADMAP	LOG
LOGOUT	LR	BER	MIN	MINUS
MINUSNOT	NULL	NUMBERP	ORT	
PRETTY_PRINT		PRINT	PROLOG_FAIL	PROLOG_LIST
PROLOG_VALUE		PROLOG_VAR	PROLOG_VMAKE	
PUT	PUTPDEF	PUTPLIST	PUTPROP	PUTPROPG
QUIT	QUOTE	QUOTIENT	RDLIN	READ
READBINARY	READCHAR	READLINE	READMACRO	READTEXT
REMAINDER	RETURN	REVERSE	REWIND	RND
RPLACA	RPLACD	RR	RS	RUN
SAVE	SAY	SAYINT	SCHEMA	SCHEMAINFO
SCHEMAPRINT	SELECT	SELECTOUTPUT		SET
SETBREAK	SETFILEPOSITION		SETQ	SETQQ
SIN	SLIST	SORT	SPACES	SQRT
STACK	STEP	SUBSTITUTE	T1IN	TAB
TAN	TEMP_FILE	TEXTLEN	TIME	TIMES
TRACE	TRAP_ROUTINE		TRUNCATE	UNTRACE
UPCASE	WATC	HDOG	WHILE	WRITEBINARY
WTLIN	ZEROP			

Built-in Prolog predicates:

,	~	!	%	/+
-	;	<	=	:=
::=	=..	=:=	=<	==
=\=	>	>=	\=	\==
@>	@>=	@=<	@<	-->
:-				
a	abolish	abort	abstrap	addrec
apply	arg	assert1	assert	asserta
asserted	assertz	atom	atomic	backtrace
bagof	built_in	call	cissue	clause
clause1	closedb	closeunit	compile	compiled
consult	contains_vars	cp	createdb	current_goal
debug	defbreak	deleterec	directive	edit
error	fail	fcal	functor	get
get0	rective	length	lisplo	not
number	op	repeat	retract	retract1
run	seekcost	seekrec	selectoutput	setbreak
setof	sorted	spaces	substitute	tabulate
time	true	user_defined	var	write