A Knowledge Based Approach to Process Engineering Design

by
Alistair Struthers

Thesis presented for the Degree of
Doctor of Philosophy
University of Edinburgh
June 1990



The work described in this thesis is the original work of the author, except where specific reference is made to other sources. It has not been submitted, in whole or in part, for anydegree at any other university.

Alistair Struthers

Acknowledgements

The author wishes to thank the following people for their assitance in the undertaking of this thesis and the preparation of the manuscript :

Jack Ponton for his supervisory role, the freedom I was given in chasing up some highly speculative ideas and for not giving up on me along the way.

Douglas Hutton and Anthony Waters for helping to proof read the manuscript, and special thanks to Dougy for all the other wee jobs he helped me out with.

"Like the wind that takes the leaf from the tree ... "
Remember the good days and forget the emptiness.

Preface

This thesis provides an extensive overview of the issues involved in adopting a knowledge based approach to the development of an integrated process design environment. The majority of the work described requires a good understanding of topics from several research areas outwith mainstream process engineering. The topics covered include logic, graph theory, knowledge representation, and theories of comprehension in cognitive science. Introductory reviews of these subjects are provided in order to make the thesis understandable to an engineering audience.

It is important to note that much of the research, covering the period 1984 to 1988, considerably predates the submission date of this thesis. As a consequence many of the ideas introduced in the first three chapters are now familiar. The content of the remaining chapters, however, concerns new ideas in the area of knowledge representation that have not been addressed elsewhere in the engineering literature.

The thesis consists of four main sections.

The first section consists of a critical review of previous attempts to develop integrated design systems using database technology. The section introduces the notions of data and knowledge integrity in an integrated process design environment and highlights the problems associated with the use of current database systems and traditional process engineering software. The problems are discussed both from a representational point of view and from one concerning the need to maintain some forms of data and knowledge integrity throughout a design project. Following this discussion a justification of the need to extend the database approach to a knowledge based approach is made. The latter approach is seen to require extra levels of representation not currently handled by relational database systems. In order to investigate this problem further a novel AIP (Appropriate and Incremental Parallelism) systems engineering framework is proposed along with a long term research framework to investigate the use of knowledge representation techniques to implement this design methodology. The remainder of the work presented was done with these long term research goals in mind.

The second section, chapter two, starts with a simple introduction to the types of knowledge representation requires in process design, e.g. objects, relationships, functional knowledge, self-awareness, etc. This is followed by a review of basic representation techniques in the areas of graph theory and first order predicate calculus from which most other A.I.(Artificial Intelligence) representation methods are derived. This is important as it provides a common means of comparing different

representation techniques and the reasoning processes involved. Chapter two finishes with a review of the techniques and problems associated with rule based production systems and discusses such problems within a logical framework to clarify the issues involved.

The third section of the thesis, chapter three, presents the initial research investigations into alternative representation techniques. Three pieces of work are described. The first involves the development of a generic rule based production system which is applied to the problem of heat exchanger type selection. This is followed by a more detailed discussion of search control problems and the use of meta-level information in rule based production systems.

The second system described, called Designer's Assistant, introduces the use of a blackboard type system architechture more suited to comples problems. A blackboard module interpreter is described in terms of its use for recommending physical property estimation techniques. Section 3.3.6 goes on to describe a fundamental problem associated with both types of system that is termed here "arbitrary tokenism". Realisation of the consequences of this problem motivated the rest of the later research described in chapter 4 onwards.

The third item of work described in chapter 3 is the development of a highly flexible frame based (or object oriented) system called CLAP (Combined Logic And Procedures). CLAP provides many features of current commercial systems as well as several unique features of its own. These include a generic relation handling mechanism, extended message passing, methods or procedures using Prolog like interpretations, and the use of "extended methods" to provide a means to implement high level process design methodologies.

Chapter 3 finishes with a summary of the relevance of the basic A.I. representation techniques to process engineering design and reviews other relevant research in this area.

Chapters 4 through 7 represent the fourth section of the thesis and draw heavily on the insight gained from the research presented in earlier chapters. The section begins with a reassessment of the requirements for an integrated process design environment and puts forward fundamental research goals that have yet to be achieved. With these goals in mind a more in depth assessment of the problems associated with both rule based production systems and frame based systems is presented. In particular the pitfalls associated with the interpretation of frames within an inheritance hierarchy are highlighted. The use of formal non-monotonic logics to tackle some of these problems is introduced.

Section 4.3 provides an extended review on the problem of representing uncertain information in knowledge based systems. Strong criticism is made of the inappropriate use of numerical based approaches, e.g. Baye's method, Dempster-Schafer theory, Fuzzy logic, etc. This is contrasted with research in the are of extended use of truth maintenance systems and the use of modal logics to represent such information.

Following these criticisms and the unlikelihood of current A.I. representation techniques being suited to the long term development of process design environments, the remainder of the thesis is concerned with a fundamental investigation of the nature and representation of conceptual information or knowledge. This work draws on advanced theories of comprehension in cognitive science and psychology. In particular the study of conceptual categorisation, family resemblances, and the use of mental models in presented in some detail. Chapter 6 goes on to describe the preliminary redevelopment of Designer's Assistant in a way that investigates the feasibility of practically implementing such ideas in a working knowledge based system.

The main conclusion of the work is that current knowledge representation techniques are far from adequate in terms of being able to provide the tools needed to develop an intelligent design environment. The work described in the latter half of the thesis indicates that considerable research remains to be done.

Table of Contents

1 The Role of Computers in Process Engineering Design and Analysis 1
1.1 The Integrated Process Design Problem 1
1.2 Sources and Flow of Information in Process Design 3
1.2.1 Example 1 12
1.2.2 Example 2 13
1.3 The Development and Characteristics of Process Design Software 15
1.3.1 Independent Programs 15
1.3.2 Flowsheet/Design Packages 16
1.3.3 Emergence of Database Management Systems (DBMS) 17
1.4 Review of the Development of DBMS in Process Engineering Design 17
1.4.1 DBMS Characteristics and Data Representation 19
1.4.2 The Network Data Model 20
1.4.3 Hierarchical Data Model 22
1.4.4 The Relational Data Model 23
1.4.4.1 The Binary Relational Model 27
1.4.5 Development of Database Management systems in Process Engineering 27
1.5 Impact of Database Management Systems 40
1.6 Fundamental Criticisms of the Database Approach to Integrated Process Design 43
1.7 Extension of the Database Approach to the Knowledge Based Approach 46
1.8 A Process Systems Engineering Formulation of Design Methodology 48
1.9 A Research Framework For Integrated Process Design 58

Table of Contents

2 Knowledge Representation in Artificial Intelligence 61
2.1 Types of Knowledge Used in Process Design 61
2.1.1 Objects, Attributes and Relations in Process Models 62
2.1.2 Functional, Event and Situational Knowledge 63
2.1.3 Self Awareness 64
2.1.4 A.I. Knowledge Representation Techniques 65
2.2 Heuristic Graph Theory 67
1 State Space Representation 68
2 Problem Reduction Representation and AND/OR Graphs 71
2.2.1 Uninformed and Heuristic Graph Searching Techniques 73
2.2.2 Informed Heuristic Search 77
2.2.3 Heuristic Search in OR graphs 79
2.2.4 Heuristic Search in AND/OR Graphs 83
2.2.5 Summary of Heuristic Graph Search Techniques 86
2.3 Classical Logics and Theorem Proving 87
2.3.1 First Order Predicate Calculus 94
2.3.2 Reasoning Techniques In First Order Logic 97
2.3.3 Unification And Pattern Matching 98
2.3.4 Theorem Proving Techniques In First Order Logic 98
2.3.4.1 Constructing Solutions In Resolution Refutation Systems 103
2.3.4.2 Simplifying The Search Strategy 104
2.3.4.3 Paramodulation 107
2.3.4.4 Alternative Theorem Proving Methods 109
2.3.5 The Control Problem in Logic Theorem Proving 113
2.3.6 Logic Programming and Prolog 114
2.4 Generalised Production Systems 116
2.4.1 Production Systems in a Logic Based Framework 120
2.4.1.1 Direct Representations, Kowalski Form and Prolog 120
2.4.1.2 Forward Chaining Deduction Systems 124
2.4.1.3 Backward Chaining Deduction Systems 125
2.5 Implementation Aspects of Production Systems 128
2.5.1 The Mycin System 128
2.5.1.1 Representation of Facts and Rules in Mycin 129
2.5.1.2 Mycin's Control Strategy 130
2.5.1.3 Explanation in Mycin 132
2.5.1.4 Handling Imprecise Information in MYCIN 133
2.5.2 Mycin from a Logic Point of View 134
2.5.3 The Prospector-AL/X Systems 136
2.5.4 Prospector from a Logic Point of View 139
2.5.5 The OPS5 system 141
2.5.5.1 Problems in Pattern Matching and the RETE Algorithm 142
2.5.5.2 Partitioning Rule Sets and Context Layering 144
2.5.6 OPS5 From a Logic Point of View 145

Table of Contents

3 Development of Representation Techniques For Process Design 147
3.1 Development of a Heat Exchanger Selection System 147
3.1.1 Background 147
3.1.2 Description of the Production System 148
3.1.2.1 Production Rule Format 149
3.1.3 Overall structure and Control Strategy 151
3.1.3.1 Simple Help and Explanation Facilities 153
3.1.3.2 Details Specific to the Heat Exchanger Selection Program 154
3.1.4 Discussion 156
3.2 Problems concerning the Use of Rule Based Production Systems 159
3.2.1 Implementing Control Strategies 159
3.2.2 Embedding Control Information in Rules 160
3.2.3 Using Meta-Level Information 163
3.2.4 Control and Meta Information in the Heat Exchanger Selection System 165
3.2.5 Meta-Level Inference in the MECHO System 166
3.2.6 Maintaining and Extending Rule Sets 169
3.3 Blackboard Systems 172
3.3.1 Observations on Blackboard Systems 176
3.3.2 Development of Designer's Assistant 177
3.3.3 Development of a Physical Properties Module 178
3.3.4 Module Language Definition and Knowledge Representation 178
3.3.4.1 The Rule Head 179
3.3.4.2 The Rule Entrance Guard 179
3.3.4.3 The Focus of Interest 179
3.3.4.4 The Situation/Action Body of the Rule 180
3.3.4.5 The Applicability Compromise 181
3.3.4.6 Making Data Compromises 182
3.3.4.7 Observations on Rule Format 182
3.3.4.8 Structuring of Physical Property Domain Data 183
3.3.4.9 Main Components of the Blackboard Data Structure 186
3.3.5 Overall Strategy of the Physical Properties Module Interpreter 187
3.3.5.1 Initialisation of a Module 188
3.3.5.2 Planning Requests 189
3.3.5.3 Evaluating Planned Requests 189
3.3.5.4 The Input/Output Form Filling System 192
3.3.5.5 Report Generation 193
3.3.6 The Problem of Arbitrary Tokenism and Module Language Definition 193
3.3.7 Intermediate Conclusions on Blackboard System Developments 198
3.4 Development of Object Orientated Representation Techniques in CLAP 200
3.4.1 Preface 200
3.4.2 Background 200
3.4.3 Generic Objects and Instances in CLAP 201
3.4.4 Setting and Retrieving Slot Values 204
3.4.5 Classes and Inheritance 204
3.4.6 Describing Slots at a Meta-Level, Default Values 206
3.4.7 Meta-View Descriptions of Slots 207
3.4.8 Embedding Control Information and Procedural Attachment 209
3.4.9 Other forms of User Specified Meta-Information, Arbitrary Tokenism Revisited 211
3.4.10 Multiple Contexts or Worlds in CLAP 212
3.4.11 Building Process Models from Relationships in CLAP 212
3.4.12 Symbolic Relationships, Demons and Meta-View Points 214
3.4.13 Establishing a Symbolic Relationship and its Inverse 215
3.4.14 Constraint Relationships 216
3.4.14.1 Specialise Relations 217
3.4.14.2 Design Specifications 218
3.4.14.3 Operator Relations, Evaluating Constraints 219

3.4.15 Summary of Development of Equation Based Process Models 223
3.4.16 Arbitrary Tokenism and Relations 225
3.4.17 Specifying Low Level Design Procedures 225
3.4.18 External Language Interface 227
3.4.19 Using Inference Techniques in CLAP Methods 230
3.4.20 Interpreting Methods in CLAP 232
3.4.20.1 Demonstration of CLAP Methods, A Simple Flowsheeting Tool 233
3.4.21 Message Passing in CLAP 240
3.4.22 Standard Message Passing and Method Inheritance 241
3.4.22.1 Other Forms of Message Passing in CLAP 242
3.4.22.2 When to Use Message Passing and When to Use Procedures 243
3.4.23 Display Options in CLAP 245
3.4.24 Specifying Design Methodologies in High Level Frameworks 246
3.4.25 Summary: Extended Methods 251
3.4.26 Discussion and Summary of the Overall Structure of CLAP 252
3.5 Relevance of Representation of Styles to Process Design 256
3.5.1 Rule Based Production Systems 256
3.5.2 Frame Based Systems and Object Orientated Programming 258
3.5.3 Logic Based Representations 260
3.5.4 Procedural Methods as Declarative Data Structures 260
3.5.5 Blackboard Systems 261
3.6 An Overview of AI applications in Chemical Engineering 262

Table of Contents

4 Towards an Intelligent Process Engineering Design Tool 267
1 Reassessment of the Requirements for an Integrated Process Design Environment 268
4.1 Fundamental Research Objectives 270
4.1.1 Intelligent/Expert Systems 271
4.2 Deficiencies of Rule and Frame Based Systems 274
4.2.1 Rule Based Systems 274
4.2.2 FRAME BASED SYSTEMS 276
4.2.3 The Basics of Frames 276
4.2.4 Frames - An Assertional Viewpoint 277
4.2.4.1 Semantics and Representation Languages 278
4.2.4.2 Interpretation of Frame Expressions 279
4.2.4.3 Instantiation, Criteriality and Matching 282
4.2.4.4 Making Comparisons and Caricatures 284
4.2.4.5 Default Reasoning 286
4.2.4.6 Reflexive Reasoning 287
4.2.4.7 Summary 288
4.2.5 Problems Concerning The Use of Frames 289
4.2.6 Improved Implementations of Frame Based Systems 293
4.2.6.1 CLAP - Conditional Definitions and Contingent Universals 294
4.2.6.2 KRYPTON 295
4.2.7 A Mathematical Analysis of Inheritance 297
4.2.7.1 The OPUS System 303
4.2.7.2 Summary 304
4.3 Problems of Dealing with Uncertainty 304
4.3.1 Bayesian-like Approaches and Other Numerical Measures of Uncertainty 307
4.3.1.1 Bayesian Inference 309
4.3.1.2 Dempster-Shafer Theory 312
4.3.1.3 Fuzzy Logic 313
4.3.1.4 Presenting Recommendations 314
4.3.1.5 Endorsement Based Reasoning and Truth Maintenance 316
4.3.1.6 Modal Forms of Logic to Express Uncertainty 324
4.3.1.7 Summary: Problems of Representing Uncertainty 329
4.4 SUMMARY 331

Table of Contents

5 A Study on the Nature of Concepts and Mental Modelling 333
5.1 Knowledge Representation - The Epistemological Level 334
5.1.1 The KLONE system 338
5.2 Conceptual Categorisation 342
5.2.1 Basic Category Types 344
5.2.1.1 Identity Categories 344
5.2.1.2 Equivalence Categories 345
5.2.1.3 Affective Equivalence Categories 345
5.2.1.4 Functional Equivalence Categories 346
5.2.1.5 Formal Equivalence Categories 346
5.2.2 What is Achieved by Categorising? 347
5.2.3 Attributes, Attribute Combinations and Concept Types 351
5.2.3.1 Defining, Criterial and Non-Criterial Attributes 352
5.2.3.2 Attribute Combinations and Basic Equivalence Category Types 354
5.2.4 The Process of Concept Attainment 356
5.2.4.1 Factors Affecting Concept Attainment 357
5.2.4.2 An Overall Framework for Concept Attainment 359
5.2.4.3 Validation and the Learning Process 359
5.2.5 Further Aspects of Atrributes : Typical and Generic Instances 361
5.3 Prototypes, Family Resemblances and Categorical Frameworks 363
5.3.1 Further Discussion and Criticisms 368
5.3.2 SUMMARY 374
5.4 MENTAL MODELS 376
5.4.1 Scope of Johnson-Laird's Study 376
5.4.1.1 The Doctrine of Functionalism 378
5.4.1.2 Mental Models and Effective Procedures 378
5.4.2 MENTAL LOGIC 380
5.4.2.1 Propositional Based Reasoning 381
5.4.2.2 Syllogistic Reasoning 383
5.4.2.3 Other Forms of Deductive Reasoning 386
5.4.2.4 Summary: The Status of Mental Logic and its Consequences for Knowledge based Systems 387
5.4.3 Types of Mental Representations 388
5.4.3.1 Intension, Extension and Formal Models of Language 390
5.4.3.2 Aspects of Word Meaning 392
5.4.3.3 Necessary and Sufficient Conditions 393
5.4.3.4 Concepts, Prototypes and Family Resemblances 394
5.4.3.5 The Nature of Intensions 395
5.4.3.6 Abstract Concepts, Formal Languages and Necessary Conditions 398
5.4.3.7 The Nature of Extensions and Vagueness 399
5.4.3.8 Summary: Word Meaning 400
5.4.3.9 Styles of Word Representation and the Effect of Context 401
5.4.4 A Procedural Semantics for Mental Models 404
5.4.4.1 Principle Assumptions 406
5.4.4.2 Basic Procedures 407
5.4.4.3 Other Aspects of Mental Models 410
5.4.4.4 Summary 416
5.5 Overall Summary 417

Table of Contents

6 Development of Knowledge Representation Ideas 421
6.1 Outline of Representation Requirements 421
6.2 System Structure 423
6.2.1 Use of Blackboards 426
6.2.2 Module Operation 427
6.3 The Development of Concepts 429
6.3.1 System Level Concepts 431
6.3.1.1 Concept Layout 434
6.3.1.2 Concept Header 435
6.3.1.3 Concept Referent Bindings 435
6.3.1.4 Context Dependencies 436
6.3.1.5 Concept Nature 437
6.3.1.6 Functional Aspects of Concepts 437
6.3.1.7 Criterial and Non-Criterial Attributes 438
6.3.1.8 Concept Structure 439
6.3.1.9 Subtypes, Special Cases and Use of Inherited Forms 440
6.3.2 Functional/Procedural Concepts 441
6.3.2.1 Examples in Set Theory 442
6.3.2.2 Using External Code to Solve Simple Equations 448
6.4 Initial Attempts to Implement a Maths Module 450
6.4.1 Simple Example Problems 450
6.4.2 Main Utility Areas 452
6.4.2.1 Expression Parser 452
6.4.2.2 Temporary Model Interpretations 453
6.4.2.3 Using Internal Concepts 453
6.4.2.4 Concept, Context, and Attribute Utilities 453
6.4.2.5 Topic Utilities 454
6.4.3 Object Entries in the Functional Model 455
6.4.4 Overview of Maths Module 455
6.4.5 Discussion of Designer's Assistant and Further Research 456

7 Final Conclusions 459

A Basic Graph Theory Terminology 463
A.1 AND/OR Graphs 464

B Basic Terminology of Formal Logic 465
B.1 Propositional Calculus Terminology 465
B.2 Predicate Calculus Terminology 467
B.3 Substitution and Unification 469
B.4 Prenex, Normal, Skolem and Clausal Forms 471
B.5 Formal Deduction Systems, Rules of Inference, Theorems and Proofs 472
B.6 Lambda Calculus 474
B.7 Validity checking using the Tableau Proof Method 475
B.8 Identifying Paths in Matrix Connection 476

C Uncertainty Factors and Probabilities in Production Systems 479
C.1 Confidence Factors in Mycin 479
C.2 Propagating Probabilities in Prospector 480

D Utility Files in Designer's Assistant 483
D.1 Example of a Word Concept 484

REFERENCES i

Chapter 1
The Role of Computers in Process Engineering Design and Analysis

The last two to three decades have seen an ever increasing and diverse use of computers in the process industries. A wide range of activities now depend crucially on the use of computing facilities such as process simulation and design, online optimisation and control, hazard analysis, and, more recently, the collection and management of plant data. One of the aspects of this thesis will be to assess how the long term development and integration of such software should proceed. Rather than discuss the development of computer software in all these diverse areas, however, an outline of the role that computer software now plays in specific areas of process engineering design will be given. This will provide the necessary background for a discussion on the future development of "integrated process software" which is generally seen as the means by which the various aspects of process engineering can be more closely integrated and coordinated in industrial design. In particular, this thesis will address itself to the basic issues involved in the specification and formulation of an integrated process design environment in terms of the scope of data representation and functionality required and its mode of interaction with the design engineer. Before attempting this however it must be remembered that the terms "process engineering" and "process engineering design and analysis" represent rather illdefined but extremely complex activities. To provide a more theoretical framework for this review simplified definition of the main activities involved in process engineering design and analysis will be introduced. This is intended to make clear what is meant here by the term "integrated process design". It is also necessary to identify and characterise the various types and sources of data involved in process engineering design, and to characterise the role which process engineering software currently plays in the design process. This analysis is required to establish a framework for formal identification of issues of software specification and is covered in sections 1.1 to 1.5.

The remainder of the chapter introduces the basic functional and representational requirements for a sophisticated integrated design environment, as well as a possible design methodology to exploit such facilities. The development of these ideas will form the content of the remainder of this thesis.

1.1 The Integrated Process Design Problem

Process design is a very complex activity. It is unlikely, therefore, that any one description of what process design actually involves will adequately cover the variety

of process development techniques encountered. However, one of the main problems in assessing computer software and the expected benefits to be gained from its use is precisely because there is very little formal discussion of the design process and the associated flow and management of "design information". The steps outlined in Figure 1.1a represent an idealised definition of the boundaries of the "classical" approach to process design. This outline follows and expands on Sargent(1983).

Figure 1.1a : Process Design Activities

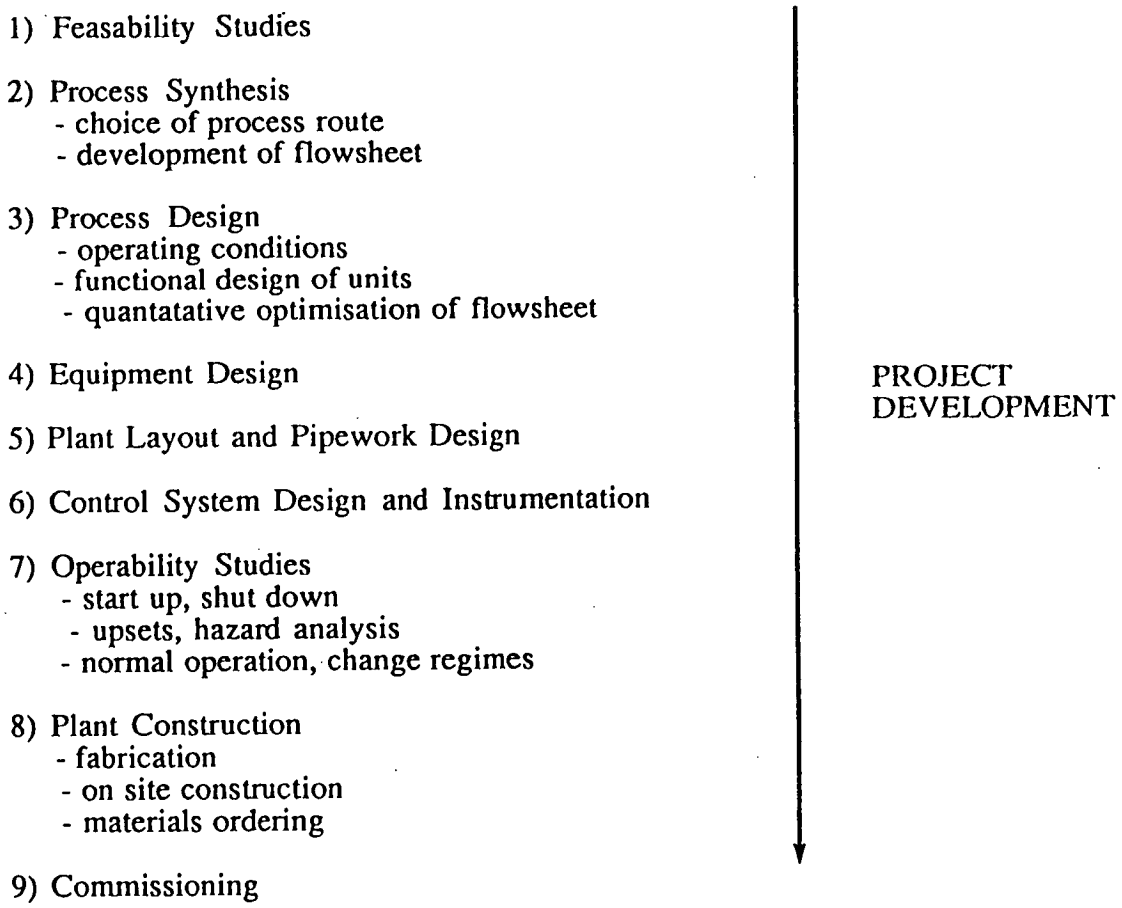
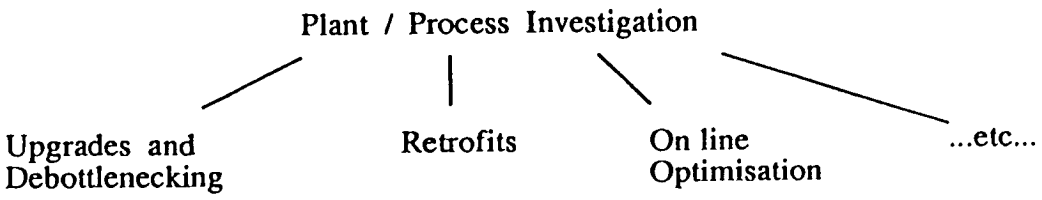


Figure 1.1b : Process Investigation Activities



(Activities conceptually similar to activities 2 and 3 in Figure 1a above)

It reflects the standard development and flow of information from initial feasibility studies to flowsheet development, followed by detailed equipment specification and design should the project be sanctioned to proceed. Although the flow of information seems to be sequential in nature, there are in fact many cross links and iterations between the various stages. The additional topics in Figure 1.1b have been added to extend the discussion to include aspects of process development and investigation considered relevant to this area of study. The information required for tackling such problems directly follows on from what has been established during the design phase. The activities in Figures 1.1a and 1.1b represent a minimum set of activities which constitute in part what will be referred to as the "integrated process design problem". The use of the word integration here, therefore, refers not only to the design phase itself but also to the integration of design project information into "downstream" activities such as plant investigation.

The actual formulation of the integrated design problem will be delayed until the notions of "integrity of data" and "integrity of knowledge" have been defined in the following section.

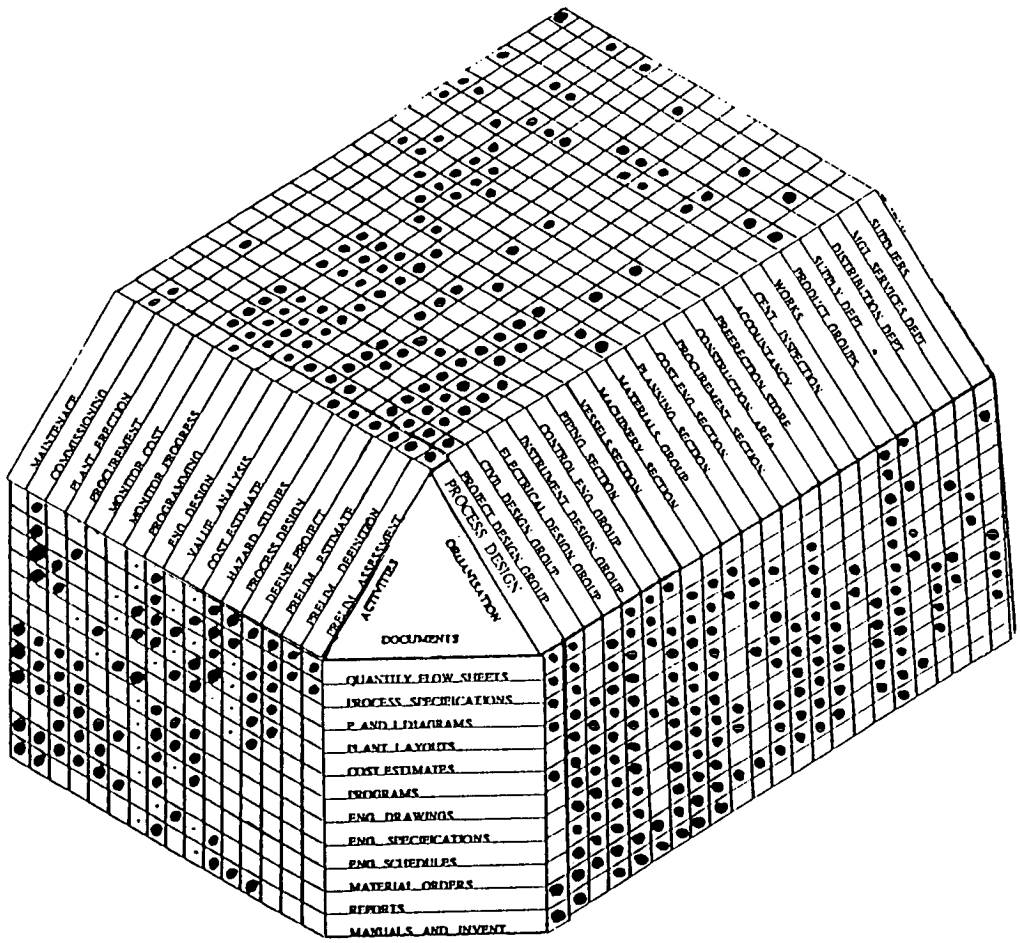
1.2 Sources and Flow of Information in Process Design

The variety of sources and large volume of information which must be handled in the design process has been recognised for some time as one of the key problem areas in process engineering software development, Zellnik(1967), Cherry et al(1982), Winter and Angus(1983),Sargent(1983).

In 1977 ICI undertook a one year study of engineering information management, the primary motivation being that it was hoped that improved communications and use of computers would lead to better management of increasingly complex and demanding design projects, while at the same time enabling more thorough analysis of safety

aspects of plant design. Although the work was mainly concerned with the flow of information and the interaction between various departments involved in an overall design process within ICI, the work is still useful in that it reveals the scope and range of interactions between qualified people from different backgrounds. The results of this work are shown in Figures 1.2 and 1.3 below.

Figure 1.2: Documents Relating to Design Activities in ICI



Comparing Figure 1.2 with Figures 1.1a and 1.1b it may be seen that the activities involved in the "integrated process design problem" extend beyond what is termed "process engineering" by ICI. Nevertheless, it is clear in both cases that :

1. There are strong and complex interactions between various design departments

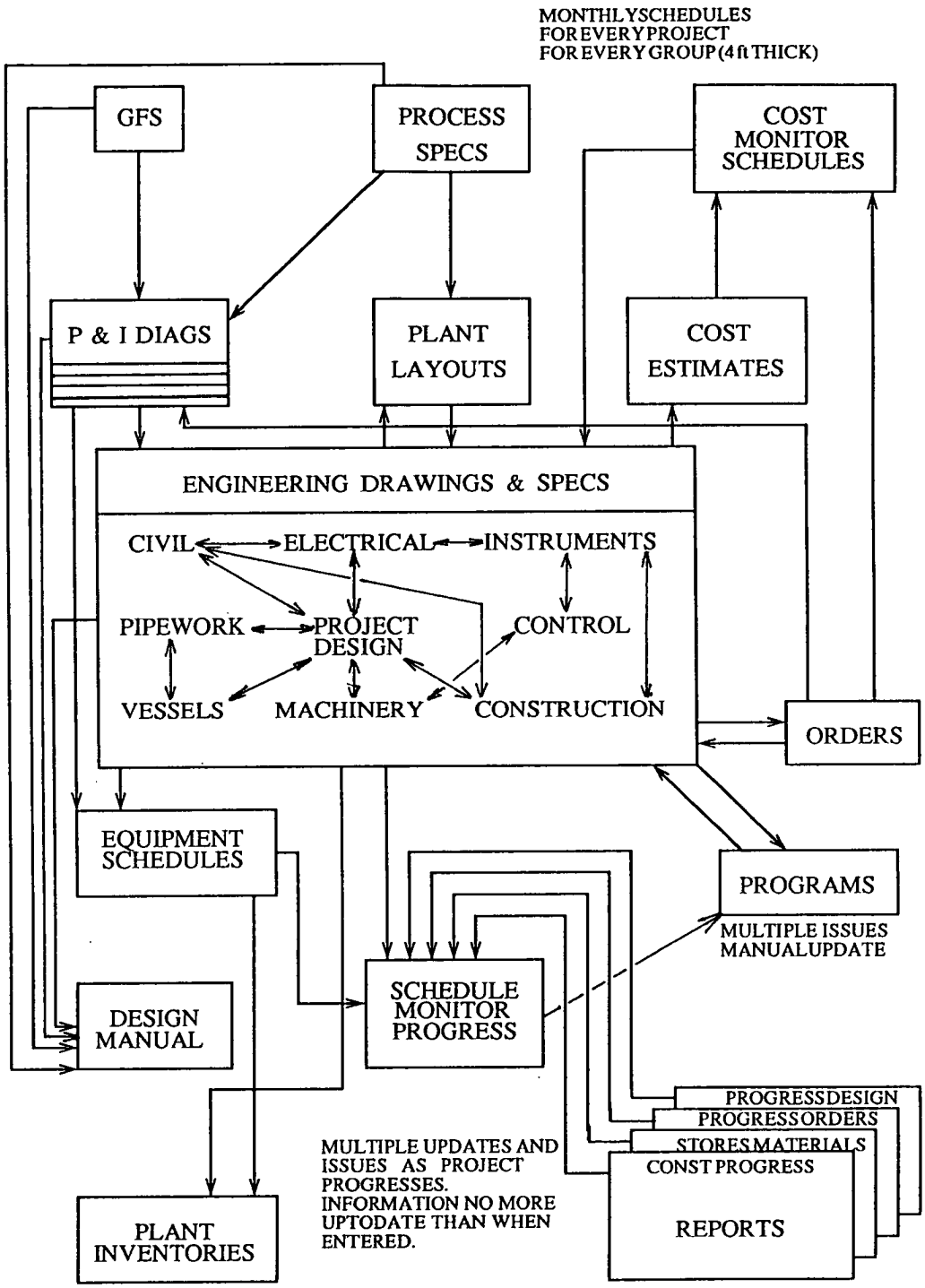


FIGURE 3 : Information Flow on Paper within ICI.

involved in the same design project, and
2. Such interactions can lead to a heavy reliance on the use of paperwork. This may cause considerable problems if any changes in design philosophy, generally referred to as "change management", are ever undertaken.

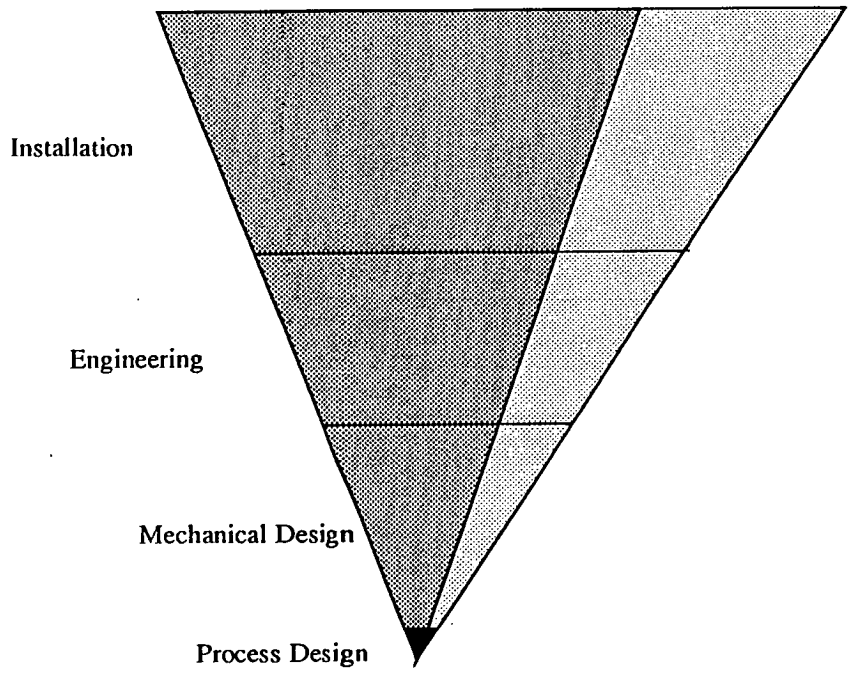
It is seen from the breakdown of ICI's design activities, Figure 1.2, that certain essential aspects of any thorough process design, such as control, cost and planning engineering, have been identified as distinct "sections" outwith "process engineering" . The result of structuring design responsibilities in this way and the subsequent handling of process documents between departments is shown in Figure 1.3. This reveals a somewhat "flat", i.e. non hierarchical, but complex shuffling of data. The question of how process design actually proceeds and the effect it has on the complexity and effort required is an important one that will be discussed later. However, it should be noted meanwhile that between ten and seventy percent of an engineer's time is spent on various forms of information handling, Cherry et al(1982). Similarly, James(1985), reports figures from a study by the Esso company which are shown in Table 1.1. The handling of information is again seen to take up some 55-65 percent of the engineer's available time.

Table 1 : Engineering Time Utilisation, by Esso Research and Engineering.

	%
Data Retrieval	20 - 25
Analysis / Calculation	20 - 30
Data Manupulation / Issue	35 - 40
Planning / Administration	15 - 20

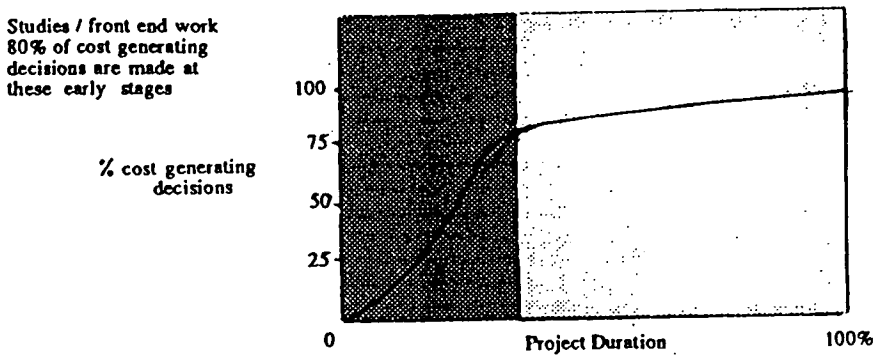
In terms of the influence design data can play in the overall costs of a project, Craft(1985) reports some interesting figures based on industrial studies. Figure 1.4 shows a typical project "cost pyramid" in which the process engineering cost represents less than five percent of the total project "spend", of which less than one percent will cover process simulation.

Figure 1.4: Typical Project "Cost Pyramid"



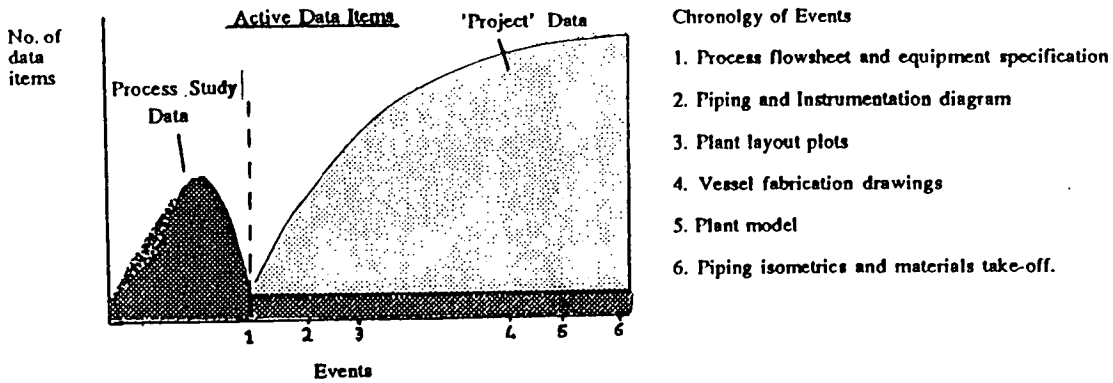
A simplistic analysis would assume therefore, but quite wrongly, that changes in this relatively small cost would have little effect on the overall project cost. This overlooks the fact that a more thorough and optimised design at an early stage could result in significant reductions in the size of the cost pyramid, as shown by the difference in the shaded areas in Figure 1.4. On the contrary, if alternative design options are not thoroughly investigated early on and too much "uncertainty" exists then the "cost pyramid" can be broadened considerably. Bechtel Ltd., see Craft(1985), suggest that some 80 percent or more of total project costs are fixed at the process flow diagram (PFD) stage of the project, see Figure 1.5.

Figure 1.5: Project Cost Generating Decisions



Process design activities will, thus, be placed firmly at the front end of most critical paths for such a project. As to how and where data is actually generated during a project, Figure 1.6 reproduces Craft's assessment of the number of data items generated by the events 1-6 listed on the figure.

Figure 1.6: Data Items Generated During a Project

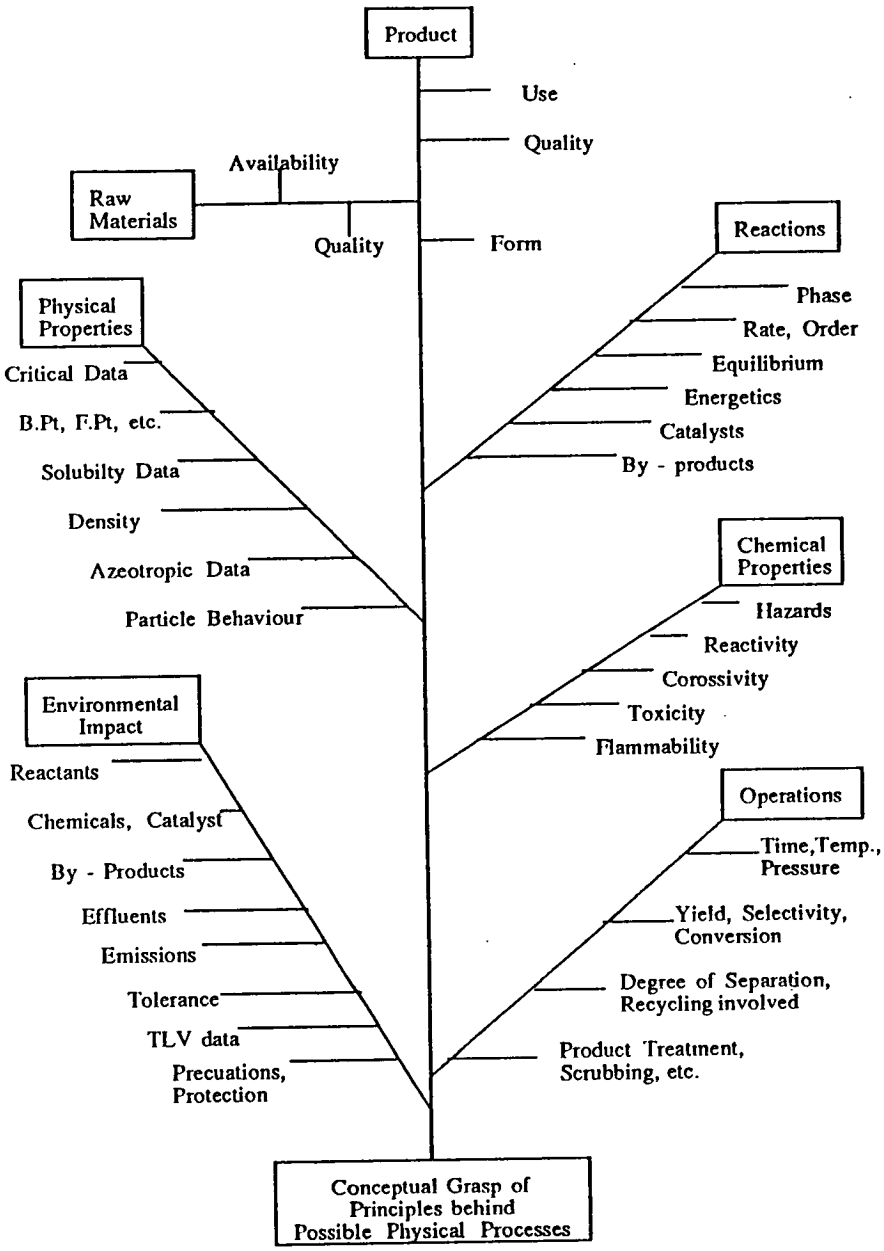


An important point to notice here is the intrinsic difference in functionality between what are called "process" and "project" design activities. In process activities many different alternatives, cases, eventualities, etc., will be considered before a fixed flowsheet is chosen for further development. During the project phase, however, a fixed flowsheet will be used as a basis from which new items of information will be generated. Hence, the temporary dip in the number of "active" items shown in Figure 1.6.

The importance of "data", and more generally, "information" at all levels in the

design process cannot be overemphasised. Consider the quantity and kinds of information required to carry out the initial design of a simple unit operation to a reasonable level of detail. Figure 1.7 shows the typical, but not complete, categories of information needed in any such design task.

Figure 1.7: Typical Design Information Categories



As this level of detail is typical of much of process engineering, integrity of data usage can be identified as one of the major problems to be faced by process software designers and one that will be frequently discussed. In order to help formalise subsequent criticisms of process software it is necessary to provide a more precise definition of what is actually meant by the term "integrity of data". The following

definition shall establish a useful basis for assessing the use and management of software in the design process.

The concept of data integrity can be considered at two levels. At a user's level the integrity of data shall be taken here to mean :

- the consistent use of data in design activities in terms of the availability, uniqueness of value, and explicit functional constraints imposed on the data. The notions of uniqueness and availability are not fixed but wholly dependent on the implementation of a data access control mechanism which then defines what these terms mean. For example, in a multi-user system which employs a data release strategy, in which each user operates on a private version of part of the project data base until a new global data update is released, then uniqueness and consistency of data can only be defined relative to a user's local viewpoint. In such a system, therefore, global consistency can only be periodically enforced when data updates are released which may or may not be a problem depending on how interdependent users' activities are.

  However it is implemented, the control of data should account explicitly for the way in which that data is used in related process design activities in order to provide useful definitions for the terms "uniqueness" and "availability". Functional constraints are user specified and provide both structural (equality of values) and semantic constraints on the use of data in the design process. Checking for violation of such constraints provides a simple way to ensure a minimum level of data integrity but has the drawback that all such constraints must be explicitly stated. Few systems operating at this level of data representation will be able to infer implicit constraints automatically.

The implications for a lower implementation level view of data integrity will be expanded throughout the rest of this chapter. Having specified at a user level what is meant by the integrity of data, it is possible to embody this within the higher level, and more powerful notion of "integrity of knowledge" in the design process. By this it is meant that

- there must be an explicit representation of the assumptions and principles upon which the data is based as well as an explicit knowledge of how that data can be used and manipulated in terms of data dependency, both of which must be consistently maintained throughout the design process. This can be thought of as being consistent about the actual reasons behind data constraints, which in turn requires that one is aware of the background knowledge, theory, and decisions responsible for the creation of the data in the first place.

This concept of "integerity of knowledge" is powerful and has far reaching consequences for software design. It is fair to say that this concept would be an integral part of any "ideal design process" but it is questionable whether or not it can be achieved in practise. Nevertheless, having provided such a definition it is now possible to formulate what will be meant by the term "integrated process design", as introduced in the preceding section. This activity shall be taken to mean one which proceeds according to

- the explicit statement of the principles of overall management of a design project in terms of the design activities and associated methods. The design methodology adopted is not fixed, but must incorporate the notion of knowledge integrity into its formulation.

The reasons for this specific formulation of process design shall be discussed later in this chapter. They have an important effect on how we should assess and develop process software which is to be used in an integrated design environment.

Finally, two typical examples taken from the literature will be described to illustrate two aspects of how the notion of integrity of knowledge might affect the use of process design software. The first example shows the importance of explicitly knowing how data is to be used, while the second shows the importance of access to explicit knowledge of the theory underlying user written software.

1.2.1 Example 1

Mah(1977) outlines some of the pitfalls of using "black box" thermodynamic routines in design packages, that is use of thermodynamic techniques without an appreciation of assumptions made in the software, likely accuracy of results etc. In particular Mah emphasises the "importance of relating thermophysical data estimation procedures to the design application of the information so generated". To illustrate this he considers two case studies in which vapour liquid equilibrium and solubility data are used in the design of an extractive distillation column and a stripping unit.

The first is the separation of ethanol and isopropanol by extractive distillation using water as the mass separating agent. In the absence of ternary data, the design could proceed by using high quality binary data for the binary systems ethanol-water and isopropanol-water and assuming the homologous pair ethanol-isopropanol to be closely approximated by an ideal solution. Note, however, the binary pairs ethanol-water and isopropanol-water are far from ideal and form minimum boiling azeotropes. Mah then extrapolated this binary data to predict ternary behaviour by fitting van Laar coefficients in a thermodynamic model, referred to as model I, which could then be used in subsequent calculations. This model was then compared to one in which van Laar coefficients were fitted to limited ternary data, model II. For prediction of equilibrium data model II proved to be far more accurate, model I deviating markedly from experimental data in both magnitude and shape of curve. To investigate the downstream effect on process design, stage to stage calculations were performed for an extractive distillation column and again model I departed significantly from a thorough design based on experimental data. The point to note , as regards the relevance of integrity of knowledge, is that both models provided equally valid data from the point of view of the integrity of data. The integrity of

knowledge, however, was violated in that no explicit account of how the data was generated, was to be used and its possible/likely effects was taken into consideration.

Mah's second case is the design of an ether stripper to separate ether and light ends from alcohols and water. Here liquid phase activity coefficients are estimated from solubility data which is possible when two of the components are sparingly miscible. Further comparison with a design based on fitted experimental data reveals that both models provide similar results despite the simplistic approach.

In summary, Mah recognises that one cannot answer the question "is a given thermophysical property correlation satisfactory" without knowing how the data will be used. The adequacy of any model depends on the application and the thermophysical data estimation procedures must thus be related to this "ultimate application". In the case of the ethanol-water/isopropanol-water system the quality of the binary data is high but the design of the extractive distillation column requires considerable but unjustified extrapolation of the data. On the other hand, the ether stripper design permits the use of binary solubility data, the system being one in which the components are sparingly soluble and little extrapolation is required. In addition, Mah points out that is unwise to place too much significance on the numerical values of individual coefficients fitted to empirical correlations. This is likely to cause problems in systems which might attempt to enforce data integrity without explicit knowledge of this heuristic. Different users might wish to use their own sets of fitted coefficients which although differing significantly in numerical value, produce quite similar results, as Mah's ether stripper example shows. A system attempting to enforce integrity of knowledge would feel easier about this state of affairs if it were able to check the effect of such data conflicts on downstream activities.

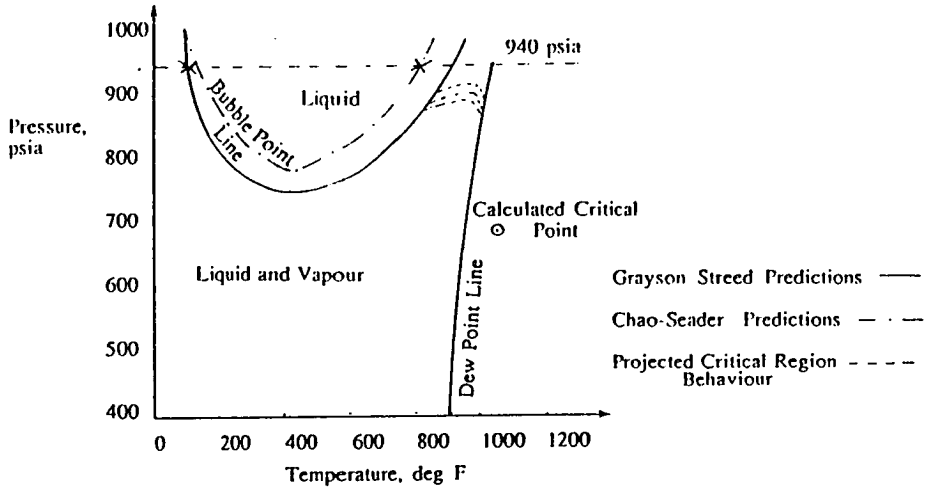
1.2.2 Example 2

This is taken from Adler et al(1977), on the industrial use of equations of state. Their case studies are interesting from the point of view of integrity of knowledge in that they show how an appreciation of the theory associated with bubble point, dew point and flash calculations is necessary to make sense of computer generated results.

The first case involves the quite common, but poorly documented phenomenon of multiple bubble points. The design in question was of a hydrodesulphurisation unit where hydrogen and hydrocarbon mercaptans catalytically reacted to form hydrogen sulphide. This is subsequently stripped out. A high pressure separator unit is positioned between the reactor and the stripper to provide a hydrogen recycle to the reactor. The design engineer had performed a computer calculation which modelled

heating a bubble point liquid from the base of the high pressure separator and feeding it to the stripping column to be flashed. The computer calculations showed that despite heating the bubble point liquid from 110 deg. F to 610 deg. F , the stream was still in the liquid phase, which seems implausible. The computer, however, was in fact correct for the following reasons. The liquid feed consisted of about eighty percent heavy oils. The behaviour of such mixtures can, however, still be influenced by light components and in particular hydrogen can have a large effect on the shape of the phase envelope. This particular feed contained 3.2 mole percent hydrogen. The phase envelope for this mixture is shown in Figure 1.8 and shows a pronounced minimum in the bubble point curve.

Figure 1.8: Example Phase Envelope



The engineer was performing the calculation in a single phase region and a temperature of around 810 deg. F would be needed to reach the second bubble point. This "retrograde" behaviour is interesting in that it is decidedly different from normal isobaric retrograde vapourisation, namely two bubble points at a fixed pressure above the critical pressure separated by a two phase region. In fact, Adler et al point out that this phenomenon is quite likely whenever hydrogen is present in heavier mixtures at about 7 mole percent or less, as in this case.

What is of interest here is that while several estimation techniques are capable of predicting this behaviour, a) commercial flash routines are unlikely to generate these multiple bubble points without user guidance and b) a data integrity system which performed simple minded checks on generated computer data would probably reject or handle such examples in an unsatisfactory way. Any system attempting to enforce integrity of knowledge should not only accept such results but try to establish a more

detailed description of the mixture behaviour. This would be necessary in order to satisfy itself of the validity of the results given an awareness of how such phenomena arise. In current design systems thermodynamic routines are usually embedded within a flowsheeting or other design package. It is then much more difficult to control or even be aware of such behaviour.

A similar example Adler studied was that of performing dew point calculations for natural gas mixtures which are very sensitive to heavy ends characterisation, even at concentrations of less than one mole percent. In such systems the "retrograde bulge" in the phase envelope can extend well beyond the true critical point of the mixture. This can result in converging to seemingly strange solutions when performing dew point calculations if the design pressure is chosen to be less than the critical pressure of the mixture. Once again, such results can be simply explained given an appreciation of the theory underlying dew point calculations.

A moments reflection on how the notion of integrity of knowledge would both constrain and guide how an engineer would proceed in solving these problems, and more importantly in the wider context of a complete process design, reveals the far reaching effects of this principle. In terms of an overall design project what the principle appears to require is a redefinition of what constitutes acceptable process design within an integrated environment : a strong but necessary statement to make if such a principle is adopted.

1.3 The Development and Characteristics of Process Design Software

Having defined the integrated process design problem and the associated types and sources of information to be managed, there follows a short review on how the development of process software has proceeded. There have been three distinct approaches to software development which still exist to lesser or greater degrees today, namely independent programs, interactive packages and database management systems. These approaches to program development will be discussed with regard to their eventual use in the design process.

1.3.1 Independent Programs

The earliest design software consisted of programs specially written to solve a specific problem such as distillation column design, reactor modelling, etc. The characteristic of this approach to software design is that it is "program orientated". That is, the data needed for the program is made to conform to the program requirements rather than vice versa. The end result is usually a collection or library of

independent "stand alone" programs that use and produce data in a non-standardised manner. Although such programs are undoubtedly useful there are several consequences of adopting such an approach :

- Much documentation is required if the programs are to be used by others
- Programs will tend to require data in specific, and often restrictive, sets of units. Units will vary widely between programs.
- There can be a high failure rate or misuse of the programs by "non experts" , Winter(1986).
- Tens of years of man effort can go into developing a single program. Whether the benefits gained and lessons learned can be usefully employed elsewhere depends on the contact between users and programmmers within departments. This style of programming on main frame computers within large companies was becoming less prevalent by the mid seventies but there has been a resurgence of such activity with the wide spread introduction of personal computers.

1.3.2 Flowsheet/Design Packages

From the mid sixties to mid seventies there was a rapid growth in the development of both computing capabilities and in the complexity and robustness of numerical techniques suitable for computer implementation. As a consequence, quite large general purpose flowsheeting and modelling programs began to emerge, e.g. Flowpack I (ICI, 1970), Concept (CADCentre), Design 2000 (Chemshare) and Process (Simulation Sciences). Many of these programs were written by specialist software houses and were intended to be suitable for a wide range of modelling needs, hence providing varying degrees of choice between model type, complexity, and solution technique. In most cases interactive user interfaces were provided to ease the problems for non-expert users. Important features of these are :

- As use by a variety of companies with varying needs was envisaged, extensive user and system documentation was necessary.
- Although programs were often interactive, varying levels of restriction were imposed on the user in terms of choice of units, input style etc.
- As the advantages of structured programming became apparent, programs became more modular in nature. In particular, input and output routines were separated from the main calculation routines enabling much easier maintenance and modification.
- Program output, however, was often in the form of lengthy printouts and considerable manual typing was necessary for transferring data to subsequent design programs. Despite the improvements in programming style these systems were typically costly to support, Winter and Rosen(1986), and the success rates were still only moderate for occasional users. From a process design point of view, however, these programs offered the possibility of much more productive use of a design engineer's time especially during the initial stages of project evaluation in a design project, see stages 2 and 3 in Figure 1.1a.

1.3.3 Emergence of Database Management Systems (DBMS)

The developments in sections 1.3.1 and 1.3.2 can be seen as part of a general trend that still continues :

- A desire to integrate more and more programs throughout the design process
- Engineers encouraged to use packages rather than write their own programs
- A desire for better efficiency and better utilisation of engineers' time in the "design phase"
- Continuing development of complex mathematical techniques and computer software and hardware to tackle more complex problems.

However, as already noted, the proliferation of a wide range of design software without satisfactory means of intercommunication did little to reduce significantly the time design engineers spent on handling information. In addition, project managers under ever increasing pressure to meet project deadlines were still faced with coordinating a seemingly unnecessary complex mixture of software design activities and resources. As a result many companies since the mid 1970's, as discussed below, have looked to the possibility of developing database management systems to integrate existing design software in a structured and coherent manner. Added incentives to develop such systems arose from the success of this type of software in a wide range of commercial applications. The development, implementation, and maintenance of a large database system requires, however, considerable financial investment as well as many changes in company management and design practices. Such a commitment requires that significant improvement can be made. The idea behind this approach to software development and the advantage that can be expected from its use in an engineering design environment will now be reviewed.

1.4 Review of the Development of DBMS in Process Engineering Design

The potential benefits of a data base approach to software design and intergration can be considered at two levels. First, from a programming and maintenance point of view and second, from the point of view of the non programming engineer using the system in a design.

The most important feature of data base management systems from the first viewpoint is that a data rather than program centred approach to software development must be adopted. In particular, the principle goal of decoupling programs from the data on which they operate must be achieved. This is in accordance with the currently preferred "structured and data independent" approach to program development. This involves the ability to formulate and develop

programs at a conceptually high level without regard for the detailed assignment and manipulation of data within individual programs. One of the main problems of current engineering software is the high cost of maintenance which, in many cases, may be due to strong coupling between program and data structure. Small changes in either the program or data structure can lead to a ripple of other necessary changes throughout the program. Any large changes of data structure in application software will generally constitute a major reprogramming effort. Hence the concept of data independence has much to offer both in terms of program development and maintenance.

The explicit management, access and dissemination of data independent of design software also becomes a major consideration in the use of such systems, given the important role the use of data plays in the design process, as previously discussed. As Motard(1983) points out, decision making in the design process depends heavily on the availability and reliability of information which is just "data" in one form or another.

The potential advantages to be gained from using a DBMS from an engineering point of view have been widely discussed, Winter and Rosen(1986), Winter and Angus(1983), Cherry et al(1982), Zellnik(1967), Benayoune and Preece(1987). The main points will be summarised briefly here but it should be remembered that most of these benefits have yet to be achieved in practice, as will be discussed in section 1.5.

One of the main aims of the DBMS approach is to provide ultimately improvements in the quality of design. Small improvements in reliability or efficiency, or reductions in design capital costs can often result in considerable extra profits or savings. The DBMS approach potentially enables such benefits to be gained by freeing the engineer from the time consuming and error prone problems of collecting, transferring and checking information and data, giving him more time to consider and improve the actual design. Proctor(1983) cites cost effective process design as the primary objective with benefits in engineering productivity and efficiency. In particular he points out that for a 100 million dollar capital project with engineering cost of about 20%, of which process design costs around 20% also, then a 25% increase in engineering productivity will only result in the same savings as a 1% reduction in capital. Winter and Angus list some 18 individual benefits to be gained, such as the elimination of artificial size constraints on problems, especially in regard to simulation, and the improved ability to evaluate the status of the design through case study failures, sensitivity analysis etc. They summarise the overall benefits as follows:

1. Better design through the capability afforded to the engineer to "close the loop" and achieve a more optimum solution
2. Reduced costs through the elimination of unnecessary clerical effort and redundant design activities.

1.4.1 DBMS Characteristics and Data Representation

Much of the present work has been concerned with the development of data, or more generally knowledge, representation techniques. It is thus important to consider what representation techniques are available in current database systems.

Large scale database systems have been available for well over a decade and a considerable amount of literature on them now exists in the computer science field. Unfortunately the term "database" has been overused and often misapplied. Until the development of relational databases along with strict functional requirements, there was much confusion as to the actual definition of a "database" and specifically, how this differs from some sophisticated file handling system. in a working integrated environment for process design. A description of a typical but hypothetical database system and a short summary of database terminology relevant to engineering applications will now be given.

A database management system can be considered to be the means of accessing, storing and manipulating data in a structured manner according to predefined "schemas" which are independent of any program manipulating the database. A schema will contain details of how the objects or entities which constitute the contents of the database are to be defined. This will usually include constraints, at various levels of detail, of what data types and values can be stored. The schema will be defined by a data definition language (DDL) and the database will be accessed by calls through a data manipulation language (DML). This may be a language in its own right or just the language, e.g. Pascal or Fortran, in which the database system has been written. There are, four basic features of any such system

1. The database contains not only data but also schema describing that data.
2. It has a "data structuring" capability in that it represents, or models the information required for specific applications.
3. It permits data to be shared between various programs and to continue to exist between multiple program runs.
4. It provides transparency of data storage and retrieval in that a programmer need not be concerned with the programming details of how the actual data is to be organised and stored.

Avoidance of redundancy, that is avoiding multiple storage of the same data, is also an important feature of most systems. This helps to try and ensure consistency within the database.

In some systems, the schema mostly contain information on the physical handling of

the data. In such cases, a "data dictionary" is often used to supplement details on the nature of the data and consequent constraints. This, however, is really only an artificial division of information in a schema. It some cases it can be regarded as a collection of metadata and its use will result in a layering of the database in such a way that the the abstract objects being modelled in the application become values or instances of the meta objects in the data dictionary. An example of the sorts of information contained in process engineering data dictionaries is given in Buchmann(1980) in which three dictionary categories are identified :

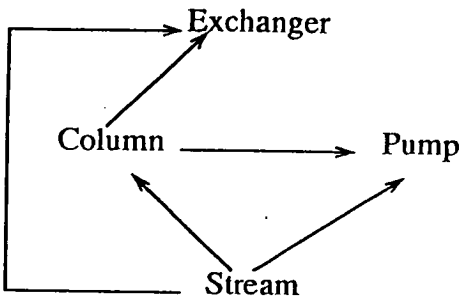
1. Design requirements such as data object identifiers, transaction descriptions, performance requirements, data volume and transaction rate information, updating and security constraints.
2. Description of local and global views of the data which includes all the object and relationship descriptions.
3. Definitions of equivalences and aliases which might be at an object-attribute level or at a relationship level (Buchmann's work is further described in section 1.4.5).

In order to utilise the information expressed in a schema, databases are based, at a conceptual level, on the idea of a "data model" which expresses the structure of the information inherent in a collection of data. The three most commonly used models adopted are the hierarchical models, network models and relational models. These models are briefly briefly discussed below. Further details can be found in any standard database text such as Ullman(1982).

1.4.2 The Network Data Model

One of the major influences in the development of database systems comes from the Data Base Task Group (DBTG) of the Conference on Data Systems Languages (CODASYL), see CODASYL(1971,1978). They proposed a formal notation for network data models in terms of a data definition language (DDL), a subschema DDL for defining different views of conceptual schema, and a data manipulation language (DML) for writing the application programs that manipulate the conceptual schema. Most network data models view data in terms of entities, or objects, which have attributes and are involved in relationships. An entity normally represents some object in the world being modelled and is often thought of as having an indivisible nature, but this is by no means a strict requirement. Attributes represent the properties of the object being represented, while relationships, or links, exist between entities in order to model associations between the real world objects they represent, see Figure 1.9.

Figure 1.9: Simple Network Model



Mathematically, relationships will either be one to one, one to many, or many to many. One to many is the most common type and many systems do not allow explicitly for many to many definitions although this can be circumvented by a further pseudo entity and two one to many relationships. Complex entity-relationship structures can be constructed using this model.

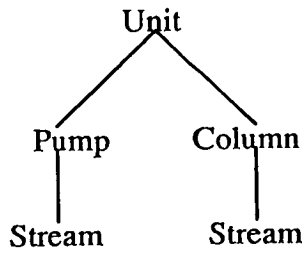
An important aspect of all data models is the need to avoid data redundancy. Data redundancy, i.e. multiple copies of the same data, can lead to inconsistencies when changes made to one copy of the data are not properly applied to all other copies of the same data. In the case of the network model the notion of "virtual fields", i.e. a virtual attribute which refers to the actual attribute on another entity, has been introduced to help overcome this problem. Appealing though this data model may seem at first sight, the approach can be fraught with difficulties if not very carefully applied. Due to the vagaries of the English language much confusion can arise in deciding whether something is an entity or a relationship, eg the design process itself could be treated as top level object or a complicated relation. A similar problem is that of deciding whether something is an attribute or a relationship, such as an entity "heat exchanger" with attribute tubes or alternatively, two entities heat exchanger and tube in relation to each other. It is difficult to predict in advance how such decisions will affect the subsequent operation of the system. Winter and Angus(1983) cite the common problem of deciding between a general entity type such as a Unit Operation or a set of individual class types. A distillation column, say, could have a very different set of attributes from a heat exchanger. Redundant attributes in object instances could cause data integrity problems in an application if one opted to describe plant items as instances of a single generic type Unit Operation. This problem also extends to classifying objects at lower levels such as whether to create entities such as plate heat exchanger, or to leave the exchanger type, "plate" in this case, as an attribute. In addition, the classification of data into entities and attributes

can result in conflicting descriptions when trying to model different applications which are to be "integrated". These problems are, in fact, part of much more fundamental representation problems which have now received some discussion in the AI literature and will be considered in chapter 4.

1.4.3 Hierarchical Data Model

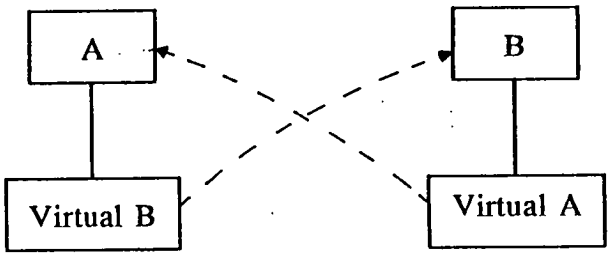
As its name suggests, the hierarchical data model adopts a layered tree/branch structure to organise data storage, see Figure 1.10.

Figure 1.10: Simple Hierarchical Model



It is a special case of the network model and is simply a network whose links form a "forest", i.e. a collection of trees, with each link representing a many to one relationship from the child to the parent node in the tree. At first sight it may appear that few networks are strict hierarchies and that, therefore, the hierarchical model is less powerful than a network model. In fact, any entity-relationship structure in a network model can always be represented in the hierarchical model if the notion of virtual entity types are introduced, see Ullman(1982) for a simple algorithm for this mapping. Virtual entities, or record types, are just pointers to other record types and enable apparent duplication of records without introducing redundancy. Such a technique is required to represent many to many relationships in the hierarchical model. Figure 1.11 shows the case of a many to many relationship between logical entities A and B in which virtual B is a child of A and virtual A is a child of B.

Figure 1.11: Many to Many Relationship



More sophisticated use of the hierarchical model has received little attention in the engineering literature, indeed in the review of Winter and Angus(1983) they are dismissed without comment. This is unfortunate since several engineering research applications using Artificial Intelligence Frame based Systems are based on techniques which are not too dissimilar to the more advanced use of hierarchical databases, as discussed later in chapter 3.

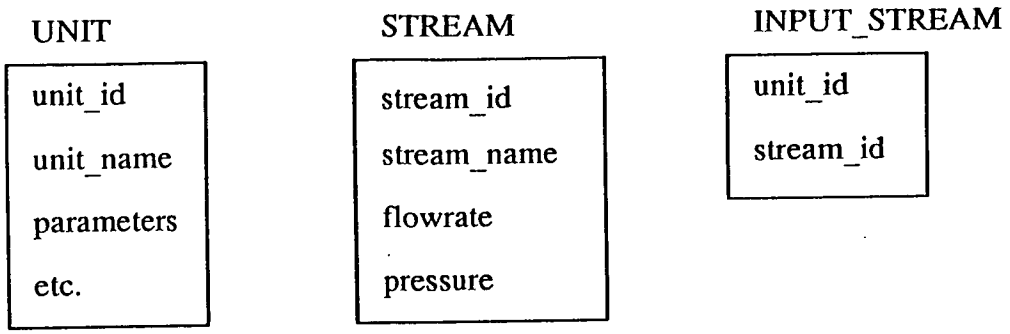
In the context of a process engineering design environment there is a problem common to both the hierarchical and network models. The drawback is that a careful analysis of the entities, attributes and relationships involved must be performed in advance of specifying the database. Given the size, scope and volatility of the types of data items involved in a project design this is almost impossible. As Buchmann(1980) points out, it is a common feature of both models that they are not adaptable to significant changes in the way an object system is modelled. As a result, if major changes are made, new relationships introduced etc., then semantic information, as represented by previously defined relationships, can be lost or misinterpreted when the global user view of the schema is modified.

1.4.4 The Relational Data Model

Codd(1970,1979), was a pioneer in taking a more formal and mathematical approach to the representation of data and the allowable operations on it. As a result the relational model gained wide early acceptance in academic circles but has only recently become the preferred approach in commercial applications. The mathematical background to this topic can be found in e.g. Gray(1984). The relational model consists essentially of relations, n-tuples and domains. A relation can be thought of as a flat, two dimensional m by n tuple. Each row is an n-tuple, that is a collection of n attribute values, and each column is a domain. Each of the n components of an n-tuple is mapped onto the domain of allowed values for that column. The number of attributes in a tuple is usually referred to as the arity. Figure

1.12 shows a simple example of a relation.

Figure 1.12: Simple Relational Tuples



Mathematically, a relation is a set theoretic relation which is a subset of the Cartesian product of a list of one or more domains. The Cartesian product of domains D1,D2...Dn is just the set of all n-tuples (a1,a2...an) such that a1 is in D1, a2 is in D2 and so on. A relation is just a subset of all such tuples, including the empty set. Individual columns can be identified by a role, or attribute, name as the same domain may then appear in more than one relation or more than once in the same relation. Notice that by attaching attribute names to columns of a relation then the order in which the columns are stored becomes unimportant. Tuples are then viewed mathematically as mappings from attributes' names to values in the domains of the attributes. This means that some relations will be "equal" which not be the case from a strict mathematical definition of a relation. In fact it is quite straightforward to change from one viewpoint to the other, and in particular, most literature on relational algebra adopts the more formal mathematical viewpoint.

The set of n-tuples in a relation is unordered but unique. Individual tuples in a relation are identified by a "primary key" which is unique to each tuple. In data processing a key refers to a unique value in a field of a record by which the record can be identified, such as someone's name, national insurance number, process stream number etc. In the relational model the key is formed from the concatenation of one or more of the attribute values, all of them if necessary, such that each tuple can be uniquely identified. Any remaining attributes are said to be functionally dependent. It is possible to have more than one potential primary key, the choices being called candidate keys.

A set of simple operations, which define the relational algebra, can then be used to derive new relations from existing relations. Typical operations include

- select or restrict : take one relation and give a new relation by selecting tuples of the first which satisfy some criteria.
- join : form a new relation by concatenating tuples from two existing relations where specified matching conditions hold on the domains of the relations.
- project : again form a new relation but only using a selected subset of the domains of each relation according to some criteria.

For modelling "real world" objects and relations, Chen(1976) proposed constraints on the choice of relations in an entity-relationship model which recognised two different classes of relation. Entity relations denote real world entities or objects, things which are usually referred to by nouns such as stream, pump etc. An entity relation contains data on all entities of the same type and, therefore, has one tuple per instance of the entity, e.g. all the particular pumps one is modelling. The other sort of relation Chen proposed is a relationship relation which links the keys of two or more entity relations, that is the attributes in such a relation are keys of other entities. There may also be further attributes which are functionally dependent on the key of this relation. Figure 1.12 above shows two entity relations, UNIT and STREAM, and a relationship relation INPUT_STREAM in which there are no functionally dependent attributes. Chen's model is an intuitive one and is one that has been quite widely drawn upon in the literature.

In relations where an attribute is the primary key of another entity it is called a foreign key. Two of the basic integrity constraints imposed on a relational model to ensure consistency are "key integrity", which requires that all primary keys have neither non-null nor duplicated values, and "referential integrity", which requires that all foreign keys have a matching primary key value stored in another tuple of a specified type.

On first sight much of this might seem very obvious and a simple, elegant way of storing data. Many problems do however arise in their implementation. As in the case of the network and hierarchical models, it is not always clear what constitutes an attribute, an entity relation or a relationship relation. Consider the apparently simple example of modelling the type of a heat exchanger, an example modified from Gray(1984). One possibility is to let it be an attribute of an entity relation such as Exchanger(Id_no, Plant, Rating, Type), where Id_no is a unique identification number. An alternative is to have an entity for each type, e.g. Type(Type_no, Fluid_type, Lower_viscosity, Upper_viscosity, Name) which describes special properties of the type in question. For instance, Type_no 23 might suit Fluid_type hydrocarbon in a given viscosity range and have name plate_to_plate. A relationship Type_of(Id_no, Type_no) could then describe the alternative types of exchanger. This representation is more suitable when an attribute is not really functionally dependent on the original entity relation, such as where alternative sizes, say, were

available for each model of vessel type available.

Another possibility would be to use a separate entity for each type of exchanger and listing the exchangers of that type, e.g. Plate_he(Id_no), Shell_tube(Id_no) etc. This might be useful if one is interested in requests of the type "print all plate heat exchangers" but makes it more difficult to answer "what is the type of exchanger A ?" since there is no attribute value to print out. Only yes/no questions can be easily Handled. This is similar to the problem that arises in Predicate Calculus between having Type(A,plate_he) and Plate_he(A), since one cannot quantify over predicate names such as Plate_he but only over attribute names. This is explained later in Chapter 2.

Another problem is knowing how many domains should be combined into one relation. Considerable work has been done on how to decompose relations into a normal or irreducible form. A relation is irreducible if it cannot be broken down by projection into two or more relations of lesser arity in such a way that they can be joined together to reproduce the original relation. Problems arise when the status of an irreducible relation depends on the values of the instances involved. An example might be Sells(Plant, Material, ToWho) which might have instances plant1 sells steam to plant2, ethylene to plant3 etc. If each plant could sell all its materials to each plant it sells to then there are really two relations, Sells(Plant, Material) and Exports(Plant, ToWho) since the materials exported do not depend on the recipient. However, if things change in the future and we only want to export certain materials to certain plants then the original relation is better. This problem is known as multivalued dependency, and such an example could arise, say, in a preliminary economic analysis. In the course of a design project it is difficult to predict all the value instances in advance and how they will affect interaction between relations. In such cases, normalisation of a database will be hard to achieve. Other common problems include partial dependency, in which an attribute depends only on part of a key and thus the relation should be split up, and transitive dependency in which an attribute depends on another attribute and hence only indirectly depends through the latter on the relation key. Date(1981) has reviewed these problems in some detail.

An important notion in the relational model is that of data independence. Many current implementations, following the guidelines set out in ANSI/X3/SPARC(1975), treat data at three levels of description in an attempt to achieve this. The low level, or internal, storage description is hidden from the programmer who works with a conceptual level description. A conceptual schema is a global view of the data structure and is independent of the various user views as well as the physical storage structure. This in turn is hidden from the end user who works with a "logical", or

external, level of description representing the user's view of the data. The importance of the conceptual level is that it isolates the user from the storage and access mechanisms involved in database transactions. This has important implications from the viewpoint of maintaining and updating widely used software.

Before leaving the relational model one should note, following Smith and Smith(1977), that a collection of relations forming a database are not really independent tables, but rather that an implicit hierarchy exists amongst them. Some tables will not make sense without information from others and a hierarchy can be formed showing the dependency between relations such as those between entity and relationship relations. One type of hierarchy is termed "hierarchy by aggregation" in which a higher level relation is constructed using foreign keys as components, the foreign keys referring to lower level relations. Data in this case is being passed on, or duplicated, upwards into the higher level relations. The other type of hierarchy is a sub-type hierarchy, each sub-type forming a subset or specialisation of a higher type. This hierarchy is closely related to the IS_A hierarchies used in Artificial Intelligence to represent information such as "A pump is a unit".

1.4.4.1 The Binary Relational Model

One solution to the problem of deciding how to arrange attributes in relations, is to decompose all relations into binary relations of two attributes. The relation STREAM(Id_no, Flow,Temp) can be decomposed into the relations ST_Flow(Id_no, Flow) and ST_Temp(Id_no, Temp). Notice that each second attribute is now functionally dependent on the first and these relations are just ways of representing the functions Flow(Id_no) and Temp(Id_no). An important constraint, however, is that for any given key we expect a matching tuple to appear in each binary relation. Decomposition becomes slightly more awkward when a relation involves more than one attribute in its key, such as CONNECTS_AT(Unit, Stream, Position) in which both Unit and Stream are required to form the key. To decompose this relation an extra field called the tuple identifier (or surrogate key) is created to identify the tuples and is usually some generated number, N say. The relation can now be decomposed as CON_UNIT(N,Unit), CON_STREAM(N,Stream) and CON_AT(T#,Position).

A factor in favour of the binary model for development of a project database is that it is fairly straightforward to extend and modify database schema. This is not usually the case in the other data models.

1.4.5 Development of Database Management systems in Process Engineering

Despite the elegant appeal of the database approach as a means of intergrating software, there are many practical difficulties in the development and application of

such systems in process engineering. Before discussing some of these in detail a short summary of database developments within industrial and academic areas will be given. Further references can also be found in the reviews by Benayoune and Preece(1987) and Winter and Angus(1983). Unfortunately a good deal of the literature of industrial origin is of a very general nature and lacking in any real technical information.

Several early attempts at software intergration, (Winter and Leesley(1975),Cherry(1975) and Niida et al(1977)), involved the use of what Leesley et al(1978) term the integrated sequential file handling system. This involves a pre- and post-processor for every two programs to be integrated so that the output file of one program is in a suitable form for the input of the other. This can be satisfactory for small numbers of packages but the number of pre- and post-processors rapidly grows as more packages are added, as does the number of data files. This will inevitably result in data items being duplicated in more than one file with a consequent increase in likelihood of data inconsistency. This approach is, therefore, no longer regarded as a satisfactory approach to process integration, see Lorie and Plouffe(1983).

The Chiyoda Chemical Engineering and Construction Company were one of the first industrial concerns to express interest in the development of a process design DBMS. An analysis of the requirements that the design process would place on a DBMS, as discussed later, suggested that at the time no commercially available DBMS was wholly adequate. As a result, an inhouse system known as CHEIS, Chiyoda Engineering Information System, was developed (1973 - 1975) to link a small number of existing applications programs. This work led on to the development of the DPLS (Database Program base and Language base Support) and SDSP (System Design/Development Standard Procedure) systems, see Tsubaki(1976). DPLS is based on the relational model using a three level data model and has been used to interface the CHESS process simulator, see Tsubaki and Motard(1979). One of the main changes in emphasis in this work was the importance given to shared project data and the adoption of a strong data oriented approach to program development. The SDSP system was viewed as an environment in which data oriented application programs could be written to run under DPLS.

However, there is no published work on the impact or use of these systems on the process design activities within Chiyoda. It is interesting to note, as discussed in Winter and Angus, that Chiyoda appear to have reverted to a more program oriented approach using a common filing system, WKFILE, to store text and data in large numbers of independent files. One may conclude that either the facilities in DPLS

were not sophisticated enough to cope with integrating design software throughout the design process or that the effort required to achieve this aim was considerably underestimated.

The next significant industrial contribution to consider is that of ICI, Imperial Chemical Industries. A one year study was initiated in 1977 on engineering information management with the view to managing more complex designs and ensuring tighter control of hazards and environmental factors with the aid of computing facilities. A good part of the work was concerned with information flow and design interrelationships within the design process as a whole, as has already been shown in Figure 1.3 and discussed earlier. The study accepted that computer based solutions to many of the problems would involve some form of advanced database management software. Cherry et al(1982) state that although a fully integrated database covering the complete design process was seen as an ideal solution, it was considered that database software at that time was not sufficiently advanced, big enough or reliable enough to cope with such a complex task. Even if such a system could be perceived to cope functionally it was thought that the system would operate very slowly. The more practical and shorter term solution was to design a system comprising a database for each major function in engineering design, termed "functional databases", with flexible communications between these functional databases. The aim of this work was to be able to store all the information which appears on the major output documents in process design, namely equipment specification sheets and process flowsheets.

An inhouse system called PEDB, Process Engineering Data Base, was developed which by 1982 had reached its second prototype. At that time little work had been done on communication between the functional databases but it was realised that there were many potential problems in controlling data transfer procedures and maintaining consistency between databases.

To accommodate both design data and performance variations, a structure based on a Network model was adopted with items such as "plant", "unit operation", "stream" etc., which are related by connectivity information. Each unit or stream has a set of design attributes describing properties such as size, material etc., as well as a series of "performances" which describe operating conditions under various circumstances. A set of performances for a plant under the same set of circumstances is known as a "case", such as 70% flowsheet rate, and is a special feature of PEDB. The main elements of the system were:

1. A program, HOUSE, for input of user commands an an English like language.
2. Interfaces to a variety of application programs such as the FLOWPACK

flowsheeting program, a physical properties system, FRI sieve tray performance program etc.
3. A specification sheet generator, SPEC, for user defined formatted reports although a library of standard specification sheets is provided.
4. A two dimensional graphics program for production of flowsheet schematics.

Encouraged by the results of this work PEDB has been developed into a commercial product called Design MASTER in conjunction with Chemshare and , as such, is one of the few integrated process design systems available, albeit somewhat limited in its capabilities. Jones and Liles(1983), Craft(1985), have reported impressive productivity figures achieved by DesignMASTER, see Tables 1.2a and 1.2b below, but these must be viewed in light of the effort required in setting up any particular in-house system as well as the value of the benefits to be gained. Craft(1985) estimates that some twenty one man years of effort has been spent in developing this system.

TABLE 1.2a: Productivity Gains Using DesignMaster

Activity	How	Productivity
Preparation for Calculation	Menu Driven Input	2x
Calculation Execution	Same as present	1x
Reviewing Calculations	Computerised Compare	2x
Equipment Spec. Sheets	Automated Production Error - free	10x
Process Flow Diagrams	Screen prompted	4x
Heat / Mass Drawings	Screen Prompted Error - free	4x
Data Transfer	Automated Error - free	Not quantified

TABLE 1.2b: Reduction in Task Time Using DesignMaster

Activity	Existing Methodology % Time	Designmaster Productivity Boost	%time with Designmaster
Preparing for Calculations	10	2x	5
Execution of Calculations	20	1x	20
Review/analyse Calculations	10	2x	5
Complet Equipment Spec. Sheets	15	10x	2
Produce Process Drawings	15	4x	4
Others	30	-	30
	100		66

The period 1973-1983 saw the development of three major engineering systems, based on database technology, namely PDMS (Plant Design Management System), PEGS (Project Engineering Graphics System) and CHES (Chemical Engineering System), all originating from CADCentre. PDMS and PEGS, however, were aimed at particular aspects of process design.

PDMS(1974-1977) is a system for three dimensional plant and piping layout and was developed in conjunctions with Isopipe Ltd. and Akzo Engineering. The system is based on its own database management system, DABACON. It is a large system and a description of its facilities can be found in Tricket and Chaney(1982). These include the ability to create designs of large size and complexity, the production of all forms of drawing from a single data source (e.g. plans, perspectives etc.), automatic selection of standard fittings from component libraries etc.

PEGS(1977-1981), in close cooperation with John Brown Engineers and Constructors Ltd., is mainly concerned with the development of piping and instrumentation diagrams along with associated lists and specification sheets. As such, it covers the area between basic process design and pipework engineering. PEGS also uses its own database management system, BUCCON, which was developed from DABACON. An interesting feature of this system was that it was one of the first to use multiple databases. There are three sets of databases :

1. drawing libraries containing the master copies of all drawings.
2. project databases containing non-graphical data.
3. reference libraries of standard symbols for use on drawings, report specifications etc.

The extent and scope of the information to be stored is user defined on a project by

project basis, a data dictionary being used to define the required attributes.

A major component of the system is the draughting module which handles the various froms of schematics. This module is also meant to capture what they call the "functional model" of the plant, by which they mean what the items are and how they are related and connected.

The system is based on a hierarchical model which supports forms of consistency checking and data security. Although it is claimed to provide total management information control across a whole project it can only communicate with other software using external files or specially written interfaces.

Finally, the CHES system (1981-1982), in cooperation with Davy McKee Ltd., tackles the problem of the integration of database technology with process design application programs. As such it is really more of a "front end" in that alternative database management systems can be interfaced provided they can perform the necessary tasks. It is based on the idea of an interlinked database architecture allowing each database to be optionally configured for its own storage and access requirements. One of the important aspects of this was the construction of a process flowsheeting system operating directly on the process design database. Many advanced facilities were provided using the notion of a SCHEME which is a collection of UNITS together with their associated input and output STREAMS. These include the ability to simulate the same unit at various levels of complexity and in different modes of operation, to construct more complex flowsheets from simpler ones, handle different thermodynamic models in different parts of the plant etc. User written routines can be interfaced at two levels. Stand alone programs can access the database via a set of interface routines but may not write data directly back into the database. If this is desired then the routines must be written in a form called a MODEL. This contains details of the input and output parameters, the latter of which can be made to have a PROPERTY. If so, any such parameters will be automatically be handled by a DATASET facility which will perform the necessary database storage operations. It is this facility which provides the means for automatic transfer of data between design software applications.

It is worth noting that other systems do exist which offer similar capabilities to those of PDMS and PEGS, namely the Intergraph System from Intergraph Corp., see Matthew and Dietz(1983), and the PLANTMAN system from Quest Genesys, see Whitney(1983), but little published information is available on these systems.

The final and most recent commercial system that will be considered is PRODOBAS from Prosys Technology Ltd., see Winter and Angus(1985) and Winter and

Rosen(1986). It is important in that it is one of the first systems to be specifically developed for Chemical Engineering purposes and is based on a relational data model. It is a large system with a great many facilities which cannot be described here. However, Figures 1.13 and 1.14 give an indication of the scope of software integration that can be achieved.

In terms of system facilities, see Figure 1.13, it is with noting the following :

- various system and utility functions exist for defining new projects, database security, defining new schema and forms etc.
- user written procedures can be written in the system's own procedural language for database access, performing calculations etc.
- utility subsystems exist for services such as form management and report writing independent of particular applications. These utilities can be used directly by the user or accessed by actual applications.
- application modules bring together user interfaces and utility subsystems into a single engineering "package"
- multiple schema definitions can be used by different applications

One further point of interest is the three different approaches PRODABAS offers for integrating application software. The first alternative, which is suitable for interfacing existing design software with minimum alteration, is to create an interface module which creates a, so called, input "deck" for the program. PRODABAS then runs the program with the input deck in either synchronous or asynchronous mode. In a networked workstation environment this may be done on different machines. The application then generates an output file which may require alteration to the design software to ensure that sensible and unambiguous output is obtained. A "write-back" interface module then reads the output file and updates the database. This approach has been used to interface ASPEN PLUS and the HTFS program TASC2. Although this offers a short term, pragmatic solution there are many drawbacks to adopting such an approach as shall be discussed later.

The second approach, which is particularly suited to new application developments, is to remove all conventional input/output operations from a program and replace them with direct read and write calls to the database using system provided utilities.

The third approach is to create the whole of the application within PRODABAS itself. In order to do this a procedural language, called Bertie, is provided to perform both calculations and database access. This may be appropriate where many checks are to be performed on items in the database.

Given the wide range of facilities in PRODABAS, it may be considered to be "state of the art" database technology.

Considering the lack of technical literature in this area it is unfortunate, and

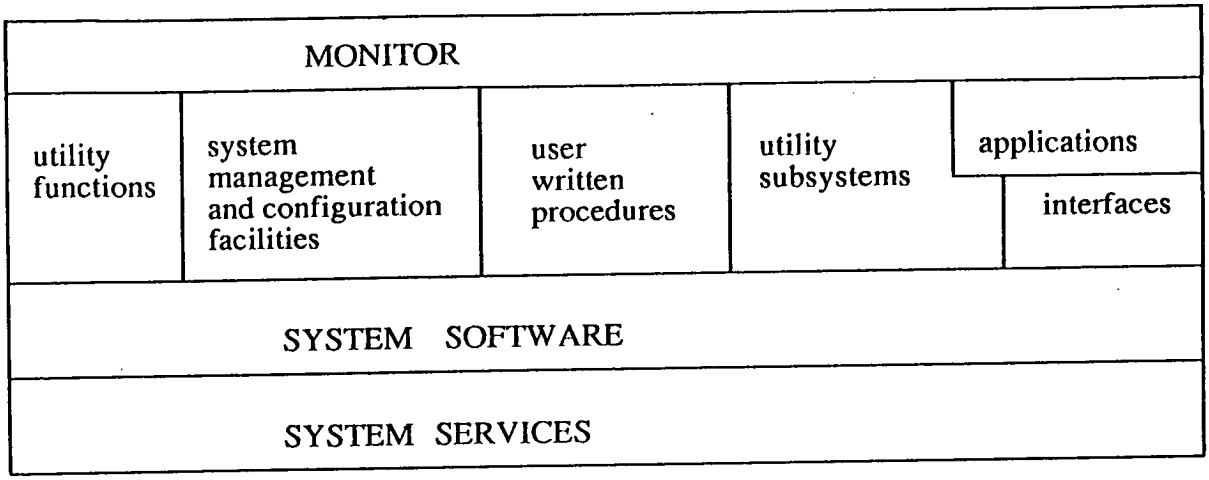


Figure113: Prodabas Software Structure

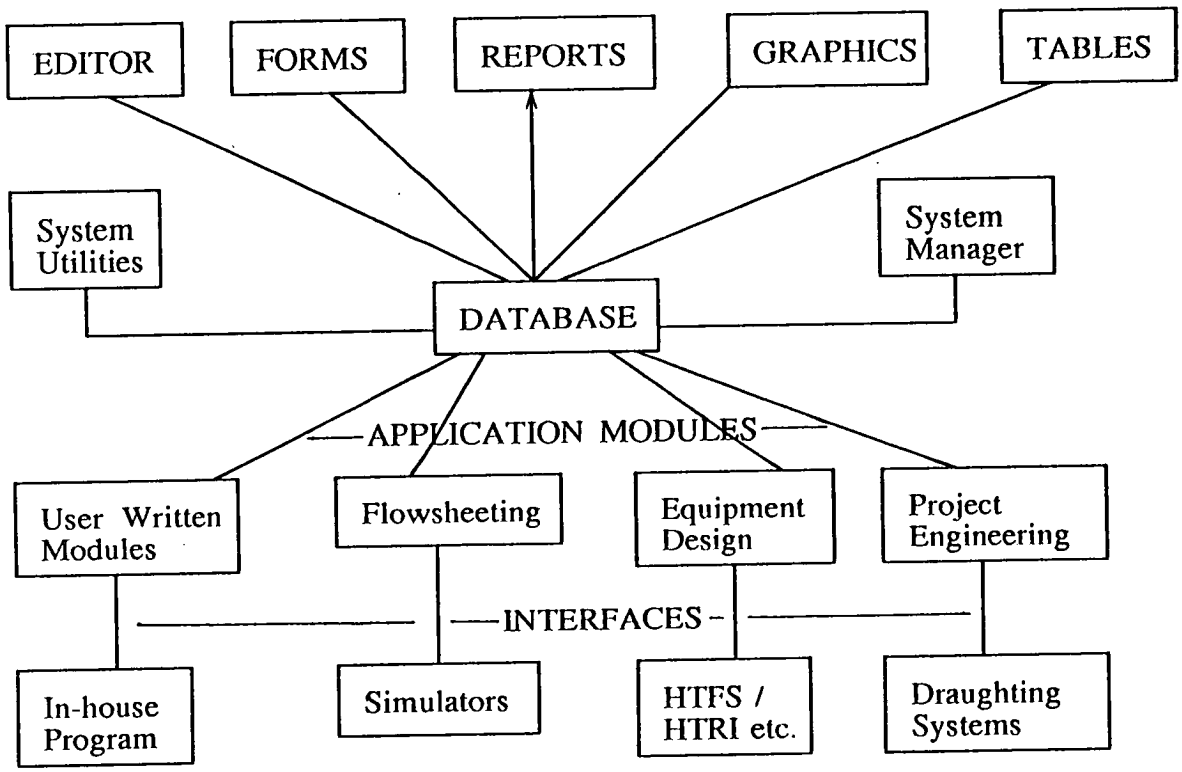


Figure114 : The Prodabas System

somewhat surprising, that very little academic work has been undertaken, especially compared to other related areas such as flowsheeting and simulation.

One of the earliest works was that of Cherry(1975) who argued for the need for new requirements in database management systems in order to handle engineering data. Many of his ideas can be seen in the development of PEDB with which he was connected. In particular he points out that there are two distinct classes of user, namely the end user and the application programs, and that these will place different requirements on a database system. For example, a program may access the database in relatively few, closely defined ways whereas a human user may wish to make ill-defined requests involving a wide range of data types. It is important, therefore, to take this into account in the design of a process engineering DBMS.

Buchmann(1980) has argued that given a good database design approach a general purpose database management system should be adequate for process design applications. His work compares different data models from which he argues that a binary data model is most suitable on the grounds that it is more stable, easier to construct and has explicit semantics which force the database designer to understand how data will be associated. It is proposed that a layered database along with a data dictionary based design approach will reduce the complexity of handling design data. As a result , he has proposed a structure for an integrated design environment, currently being developed at the University of Mexico, which includes a family of design databases in much the same vein as the functional databases in PEDB. In particular he suggests division into three types of databases : one project wide data base for approved data, a set of temporary databases for preliminary design data, and a set of static catalogues for standardised, non-specific data. Buchmann's assessment of the nature of the data handled in the three categories is shown below in Table 1.3.

Table 3: Volume and Nature of Data Handled in Project Design

data category	volume	volatility	update/ retrieval	simulataneous access
approved project data	monotonic growth, low -> high	low	low	high
temporary data	low to medium	high	high	low
catalogue data	variable	very low	zero	low

An interesting aspect of this work is the formulation of a homogeneous representation of both graphical and non-graphical data. In order to represent graphical data, such as that involved in plant layout, Buchmann has decomposed a relational type model into a binary model that remains consistent with the overall database model. The work makes use of recursive graph theory to describe the graphical information being modelled. The representation involves the use of octuples of classes, such as the sets of nodes and arcs involved, and various standard operations for manipulating the graphs. Typical operations include JOIN, combine two graphs, ZIN, insert a graph in another, etc. in much the same vein as a relational algebra. The standard representation of graphical data is a problem that has received little attention in the literature.

The final piece of academic work to be considered is that of Benayoune and Preece(1985). This work describes the specification of what is called an Engineering Data Model, EDM. This represents the world as a collection of hierarchies of OBJECTS linked together in a network type structure through ASSOCIATIONS. OBJECTS can be "primitive objects", the smallest indivisible objects such as "tube", or "sub-objects" or "composite objects". Sub-objects are components of a composite object and can be primitive or composite in nature. Composite objects are made up of two or more sub-objects. Objects can then be classified into OBJECT-TYPES according to their properties, which can be further classified into OBJECT-SUB-TYPES.

Properties can either be SIMPLE, such as temperature, or OBJECT-PROPERTIES,

such as upstream connection, and can be further described by characteristics such as minimum and maximum values etc.

Associations between objects can be of three types :

1. Object-Object : Those in which one object is a sub-object of another. This would appear to be semantically redundant, however, given the definition of a composite object.
2. Object-Property : One in which one object is an OBJECT-PROPERTY of another
3. Property-Property : Those in which at least one property of one object can be derived from the properties of another. It is not clear how useful this is since many properties will be in some way dependent on one another.

Objects can also be made "equivalent" to enable automatic update of one set of properties if the other set changes. The implementation is based on the RAPPORT relational database system, see Schmidt and Brodie(1983), but it is not clear what the EDM has to offer over other data models adopted in previous systems. The work, however, is still of interest due to the claim that a Knowledge Based Management system can also be supported.

Figure 1.15: System Architecture of EIMS

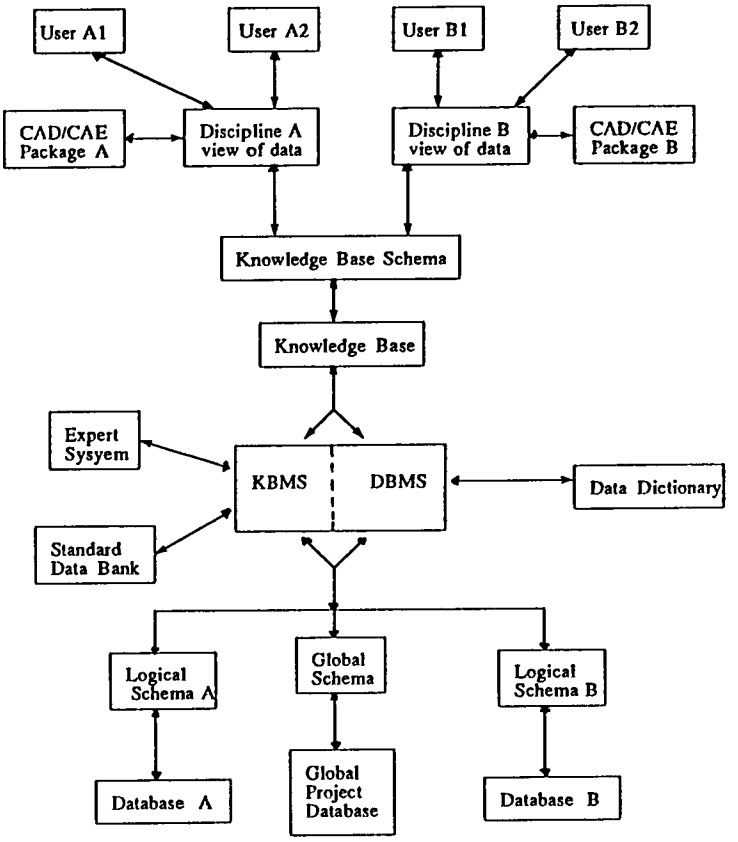


Figure 1.15 would imply that expert systems, yet to be discussed, can also operate within this environment. Unfortunately no details of any part of this work are given so it is impossible to comment further. Of obvious interest would be how they differentiate between "data" and "knowledge" and how the data and knowledge bases interact. It is stated that the "knowledge" component is represented by means of the database. If this is so, however, then there are serious drawbacks to this use of a relational database, see chapter 3. It is interesting to note that much of what is called the EDM is now provided, albeit in a more abstract form, by several current Artificial Intelligence Knowledge Based Systems.

An interesting feature of many of the database projects reviewed here, is the preferred use of a network of satellite process databases, which manipulate "live" data, linked to a central, more static, project database acting as a store for "approved data". The alternative solution would be to design a single, very large process database. Craft(1985) argues that the former approach provides a preferable solution in terms of

the difference in function of database systems on either side of the "Process Flow Diagram Event", in that each database can be tuned to the particular needs of any design activity, as in the case of DesignMASTER.

This seems somewhat of an artificial argument in that what is at issue is the need to allow alternative control and search mechanisms at different stages of a project and the ability to view data at different levels of detail as is appropriate. To achieve this through the use of separate databases, which may be acceptable for short term purposes, does introduce various problems with regard to transferral of data and maintaining up to date information throughout the network. Some of the more basic operational problems connected with the operation of database systems in engineering design shall now be discussed.

1.5 Impact of Database Management Systems

Despite the considerable interest generated in the use of database management systems and the development of commercial systems such as Prodabas, it is still true to say that such systems have yet to make a significant impact on the process industries in the same way that they did in business applications. It is important to discuss why this is so, since the potential benefits from the use of such systems have been widely accepted in terms of both, design improvements and financial gain. This shall be done in two levels : firstly some basic problems will be highlighted which have to be faced sooner or later in the development of a process engineering DBMS. Secondly, some fundamental criticisms will be made of what can ever be reasonably achieved by such systems. These criticisms, therefore, will form the main motivation for the rest of this work.

One of the most obvious problems, as seen from the review of industrial development projects, has been the non-trivial task of developing specific database management systems, often from scratch, since most commercial systems have been perceived to be inadequate for many aspects of the design process. As Sargent(1983) suggests, companies must be prepared to take the time and effort to develop a system which may mean many changes in working practises. The development of such systems will, however, require considerable guidance from experienced design engineers and many companies, especially those primarily involved in contracting work, will be unwilling to devote sufficient resources to what is often seen as a long term development. Thus, the resources and development time required to produce a workable DBMS, whether done in-house or, more recently, with the use of commercial systems, can be seen as one of the major stumbling blocks in their application.

It is worth considering just why commercial database systems were often seen to be inadequate for engineering purposes. One of the most basic differences between a data processing environment and an engineering environment is the nature of the data stored. As an example, business data will typically map easily into a relational data model : there being many recorded instances (tuples) of a few record types (relations). The schema for the relations are usually composed from simple data types such as scalars and short strings. Schema for relations in an engineering application are more complicated for two basic reasons:

1. Several additional data types have to be handled. This includes long unformatted textual data, numeric data of various types (integer, floating points, double precision, complex etc.), logical data, vectors, matrices and record data types. It is worth noting that many systems place considerable restrictions on the size of any individual data item.

2. There is often a need to be able to express the "structure" of engineering data, such as part-whole relationships, which will involve combining several tuples from more than one relation to form a more complex entity. Several methods have now been proposed for handling these structured data types, see Lorie(1983).

Another fundamental difference between engineering and business applications is the way in which the data is used. There are two aspects to be considered:

1. The engineering environment can be essentially characterised as a problem solving one rather than one for primarily disseminating information. Very little attention has been given to the problem of easily interfacing or developing "independent" design programs within database systems. To achieve this satisfactorily, it is likely that a variety of data models will need to be supported within a single DBMS to provide the required flexibility, see Balza, Bernhardt and Dube(1983).
2. The length of a transaction with the database in a data processing environment is usually short (a few seconds) and typically involves only a few "reads" or "writes". Most databases will be designed to optimise this style of operation. Engineering transactions will, however, be substantially longer, days, weeks or months, and may involve a great many "read" and "write" operations. Significantly different techniques for the management and control of such transactions must be provided if problems of data inconsistency are to be avoided.

The latter point raises one of the most vexed questions that the designer of a DBMS has to face, and yet to be properly answered, and that is the problem of how to deal with "versions" of data and "data sets", and how to provide the configuration control mechanisms necessary to cope with such data. This requires some explanation: An engineer will typically use data in order to solve a particular problem. The data will represent say, a particular fluid or object, A, as opposed to any other fluid or object, B, although the data has the same definition in both cases. Hence, the concept of a data set which associates a group of data with an abstract unit of "work". As yet, there is no problem. However, solving many engineering problems requires a significant period of time, and often proceeds in an iterative fashion. When part of a design is under revision it is usual to keep track of various alternatives or choices for a variety of reasons. It would be extremely expensive to duplicate all the project data every time an alternative is to be considered and so it is usual to create an "iteration baseline" and only store the modifications to changed items. This gives rise to the notion of "versions" of data. Major versions are complete copies of the database, while minor versions contain only the necessary modifications. The data sets in these versions now have to contain and represent a meta-level of data, i.e. data about data. This will include information on when the data was created, by whom, it s quality (however defined), it s source, it s validity (in relation to other versions etc.) and so on. Such information is necessary to enable judgements about the "stability or relative worth of the analysis performed on the data", Balza et al(1983), to be made.

We then have the problem of how to provide control mechanisms to manage such data if we are to avoid a chaotic situation where several people are working on the same problem at the same time. The problem is exacerbated by the length of transactions performed on the database so that standard lock, wait and deadlock resolution techniques cannot be used. One proposed solution, Lorie(1983), is to have controlled releases of data enabling individual engineers to work independently. However, there are then problems of reconciling various sets of data into one compatible, consistent updated database, some engineers having access to important data while others do not etc. Problems such as these are especially relevant in systems which adopt an approach using a network of smaller satellite databases. If a control mechanism can be devised it will have to allow for the particular practises of any one company, since no two companies will use the same set of classifications of Data Sets nor enforce the same set of restrictions on their use.

A further stumbling block in the application of many early systems was the inability to redefine schema easily, i.e. add new attributes etc. Due to the size and complexity of many engineering projects, it is virtually impossible to pre-define all the necessary data types, associated attributes and relationships prior to the start of a project. A continual but limited revision of the definition of schema and the level of detail required is therefore likely to be a feature of many large engineering projects. Such changes are difficult to achieve in many hierarchical and network data model implementations. However, many current relational database systems do provide the facility to dynamically redefine schema but will still require, in many cases, some re-organisation of the database. It is unlikely, and quite unacceptable, that any unaffected programs accessing the database would have to be modified.

One final problem, which has been overcome in several modern systems, is the important feature of allowing multiple logical views of data. This becomes important as soon as an attempt is made to integrate programs which share different forms of data. This facility, however, was not available in many early database systems.

Proctor(1983), has also discussed some of the problems that a process engineering DBMS should cope with, but in broader engineering terms. These include the problems of communication between groups in a design project (as indicated above), handling of project schedules and transfer of data, making expertise available and applying it appropriately, the treatment of "uncertainty" as a project develops and the management of the various trade-offs that have to be made in any project. These in turn will include a trade-off between

1. Design accuracy and reliability and the allowable time for design

2. Plant cost and process integration and the resulting flexibility in plant operation
3. Plant cost and ease of maintenance
4. Process flexibility and design complexity, tied to 2) above.
5. Design safety factors and desired reliability.

Many of these are classic problems in project management. Current database systems, on the other hand do not provide the means by which these issues can be conveniently represented in regard to restricting and controlling the design activities carried out on the database. Proctor also highlights the problems that automatic updating of the database can present. If, for instance, some new physical property data is obtained how far should one go in checking "downstream" decisions dependent on this data, such as sizing of separation columns, calculation of heating utilities etc. If such updates are to happen "automatically" then how are we to represent data dependence, and how much "history" of the use of data should be kept ? Indeed, who should be authorised to make such updates if they are to happen ? The vast majority of all these issues have yet to be satisfactorily resolved. It is interesting to note how many of these problems can be seen to be directly dependent upon the control mechanisms available within the database management system.

1.6 Fundamental Criticisms of the Database Approach to Integrated Process Design

Most of the "low level" problems that a DBMS has to cope with, as described in the previous section, could be overcome in a carefully designed environment. In spite of this, I shall argue here that there are several fundamental issues that any such system could not easily address and as a consequence, suggest that future development of DBMS in currently recognisable forms needs careful consideration given the notion of integrated process as described earlier.

Despite the wide usage and reference to "integrated process engineering", as if that were some well defined term, one of the striking features missing from the engineering literature on this subject is a precise statement of the problem that the people who adopt database software are actually trying to solve.

It is widely accepted that engineers spend a good part of their time in handling information of one sort or another, loosely referred to as the "data problem" (see Benayoune and Preece(1987)), and that there is a perceived need for some sort of information handling system. In my opinion there is a perhaps subtle, but crucial omission from this generally held view and that is one must ask "an information handling system - yes - but to do what ?". I shall now try to explain why the need for knowing and representing how an information system is to be used must be an

essential requirement for its development and operation. Most of the problems described in the previous section will be seen to be direct consequences of avoiding the solution of this problem.

Consider the following statements typical of those found on the development of DBMS within process engineering. Cherry(1982), on the development of PEDB, argues for the "provision of an effective information handling system", the long term goal being an "integrated suite of data bases for handling all project data, from the earlies stages of design, through detailed design to construction and operation". In a similar vein, Benayoune and Preece(1987) state that "the obvious solution to the data problem as outlined above would be an integrated system with automated data-handling facilities." They go on to state that the real solution to the problem lies "in a system which does more to the data than just transfer it from one package to another". It is interesting to note the basic criteria they specify for such a system:

- data should be independent of any CAD package that uses it
- data should be provided quickly and accurately when required.
- different design teams should be able to share data without any undesired redundancy
- data consistency and integrity should be maintained
- the system should be safe from any software or hardware failures
- data should be kept secure from unauthorised access
- data maintenance should be carried out by its owners without them requiring special training.
- the system should be easy to use even by casual users.

It would appear from this, and similar work (Winter and Angus(1983), Winter and Rosen(1986)), that the problem is basically seen as one of providing a convenient means of data storage and retrieval. The requirements provided by Benayoune and Preece, as quoted above, do little more than to add the words "effective and robust" to their described solution of an "integrated system with automated data-handling facilities". Indeed, other work on the requirements and characteristics of a DBMS (James(1985), Proctor(1983)) has all been done in a rather vague and general fashion in terms of ease and flexibility of use rather than with regard to what this would actually entail in the technical development of a database system. Given this state of affairs one can therefore assume that the main problem to be solved is seen as that of providing a means to ensure some sort of "integrity and consistency of data". (see also section 1.2 for a proposed definition of what this means.) If this is so then one has to seriously consider in what terms consistency and integrity of data can actually be achieved within a design context.

If an acceptable control mechanism can be devised for the access of data then one can obviously achieve consistency in the narrow terms of the availability of the same data at any one time. In addition, simple numerical or symbolic checks can be made to

achieve limited means of ensuring "sensible data values". The main point to be realised is that neither of these facilities provide the means to check consistency or integrity of data in terms of the principles upon which the data is based and the semantic implications of how that data is combined and manipulated. A trivial example would be a program supposed to be modelling a highly "non ideal" equilibrium situation using either "ideal" data or "ideal modelling techniques." Thus, we can have a "consistent" database in terms of data availability but a wholly "inconsistent" database in terms of what the data represent, how it has been generated and how it is to be manipulated. Further examples of this type of inconsistency were described earlier in section 1.2, taken from the work of Mah(1977) and Adler et al(1977). Notice that the consistent use of data manipulation techniques, which in the design context will be in the form of consistent design and modelling assumptions, are now just as important as the data itself. That is, we must now also know how a system, or the data within it, is to be used and how the actual data manipulation methods can be sensibly combined in much the same spirit as relational algebra constrains how data can be combined. A probable reason why so many problems are encountered when adapting database software for design, is that in business applications the user's view of the data and the operations he expects to perform on it are very much closer to the intentions of the designer of the database system and, hence, the appropriateness of the user interface presented by the database system. This is emphasised by the fact that in most commercial applications it is normal to carry out a formal data analysis phase in order to totally specify the conceptual schema with which will meet the requirements of all the programs in the application system. It is, however, much more difficult to analyse in advance exactly how a DBMS will be used throughout a design project and beyond.

The second basic criticism of the current database approach, following on immediately from points concerning the previous one, is that in an integrated design context it is not the "data integrity" problem that should be of primary concern. Rather, it would seem that the solution of the "integrated process design problem" and the ability to maintain "integrity of knowledge" see section 1.2 are of prime importance in the coherent development and operation of any engineering design system. It must be re-emphasised that design systems will primarily be used in a "problem solving" fashion, where knowing what you are doing in terms of assumptions and expected effects is all important. This is very different from data processing systems in which data is typically used in simple value matching operations. The next conclusion to be made, therefore, is that we must have a means of representing the issues involved in these higher level problems and adopt a "top down" approach, as it were, to system development rather than a "bottom up"

approach in which we try to add constraints to a basic data handling system. Whether or not one accepts the proposed definition of integrated process design, it is certainly true that most of the problems encountered in the development of DBMS in process design, especially in their controlled operation, stem from the inability to explicitly represent design methodologies both at the highest levels and at the low level of analysis of assumptions and functional data dependencies involved in a particular solution technique.

An example of this at a higher level would be the ability to provide flexible control mechanisms between different design groups involved with "long term data transactions" given an explicit representation of the relative importance of that work and the manner in which it interacts at any particular stage of the overall design process. Representation at a lower level, such as that of an individual design program (e.g. column sizing), is tantamount to the requirement that the system be explicitly aware of what that program does in functional terms. This, of course, radically differs from the current situation in which application programs are treated in a "black box" manner, both by the system and the non-frequent user. In retrospect, it seems rather obvious that trying to adopt software which was intended primarily for data processing rather than problem solving environments, and in which there was no need to consider these higher level issues in its development, would inevitably result in a great many practical problems. From a more general point of view, it is unfortunate that development in the integrated process design systems area, noticeably much of it industrially led, has apparently proceeded unchecked on

1. a relatively ill defined problem, whilst
2. adopting software representation techniques which seem far from adequate for the problem at hand.

In particular, current process based database systems fall far short of providing the functional capabilities required by a system which attempts to offer an integrated environment given the, albeit simplified, definitions of integrated process design and integrity of knowledge presented earlier.

1.7 Extension of the Database Approach to the Knowledge Based Approach

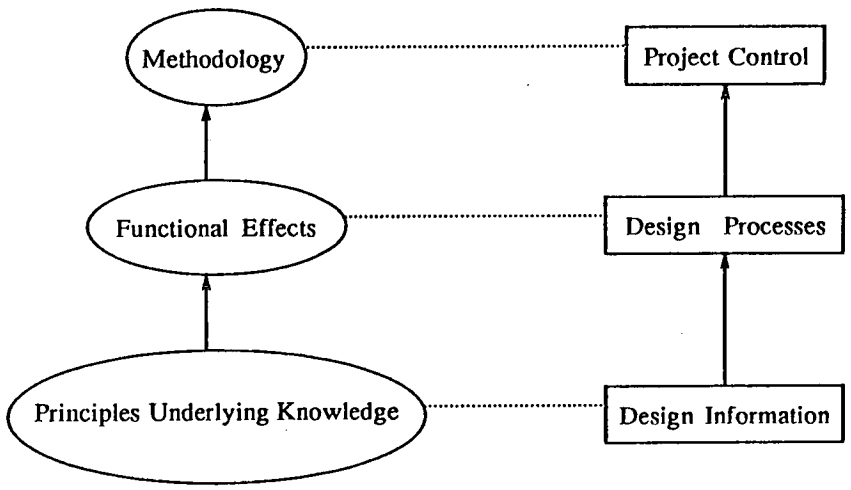
Given the previous discussion, the remainder of this work shall assume that the main problems to be considered are those of integrated process design and integrity of knowledge, as defined in section 1.2, in the expectation that many of the difficulties associated with the integrity of data shall be sufficiently constrained as to allow convenient solution. In order to attempt the solution of the problems two extra, but

essential, levels of explicit representation shall be proposed :

1. a means to both represent and interpret the basis and effect of operations which manipulate data in terms of known assumptions and functional dependencies. In terms of process design this will usually mean explicit knowledge of a process model, the assumptions upon which it is based, validity of techniques etc.
2. A means to represent, and hence reason about and control, an overall design methodology, whether at a project management level or a problem solving level. There is, of course, a third more basic level of representation required on which levels 1) and 2) will ultimately depend, and that is -
3. A means to represent the principles upon which "data" is based. One can think of this as an extension to functional dependency checking facilities in current database systems.

Levels 1) to 3), if they can be represented, can be seen as forming the basis of what shall be called a Knowledge Based System for process engineering. Figure 1.16 shows in a general way how these levels of representation interact with the design process.

Figure 1.16: Knowledge Representation Levels



It is important to notice how methodologies, that is problem solving strategies in a general sense, can be conceptually represented at various levels of detail in regard to the overall project design methodology.

The use of the word "operation" can be thought of as a specific formulation of a "model" if one is prepared to use the word "model" in a very general sense, e.g. a model of a unit operation, a model of a problem solving technique etc. The first two levels then become somewhat recursive in nature if successively reapplied at various

degrees of detail.

The requirement that one should be able to represent and reason about both the basis and effect of an "operation" raises yet another important feature of process design not currently catered for in database management systems. That is the facility to represent the reasoning behind decisions made in the design process. This type of information is of immense value in "downstream" activities such as hazard analysis, and of fundamental importance when considering plant retrofit or uprating projects. Obviously, if one has gone to the trouble of creating a project database it would be sensible to use this body of information to it s full extent in improving the day to day operation of a plant. It is frequently the case that many engineers involved in the design of a plant will not be involved in the long term operation of that plant and, hence, important information could be "lost" if design decisions and data are not fully explained in a plant's design manuals. One example, from personal experience, is that this often seems to be the case in terms of explanation of badly written detailed modelling programs used in the early stages of design.

1.8 A Process Systems Engineering Formulation of Design Methodology

Having identified the need for the explicit representation of various aspects of design methodology, i.e. how one can proceed at differing levels in a project design, it is worthwhile considering a basic problem solving framework, namely process systems engineering, in which these ideas can be developed, both in general engineering terms and the low level technical representation techniques that will be required.

An unfortunate feature of the chemical engineering literature is the lack of discussion on the nature and role of process systems engineering within the area of "integrated process design". It is unfortunate in the sense that a careful and well defined use of such an approach could provide a convenient and flexible framework in which many of the "control" issues in process design, as discussed earlier, could be resolved.

The term "process systems engineering" appears to have been first used in 1963 (title of CEP symposium series No. 46, vol. 59) . For a considerable time the term seems to have been synonymous with the notion of "computer applications in chemical engineering" but as Takamatsu(1983) points out this is a misconception in that process systems engineering, as a specialisation of systems engineering, is a topic in its own right with considerable methodological content.

Takamatsu defines "systems engineering" as a "methodology for making engineering decisions in a system which is composed of many sub-systems or parts and has to

satisfy and achieve any set of conditions and objectives". It is thus concerned with general methodologies for decision making in planning, design, management, control and operability issues etc. He goes on to specifically state that it should indicate

1. how to plan,
2. how to design,
3. how to operate, and
4. how to control

both the unit operations and the actual overall chemical process itself. We, therefore, have a potential framework in which it is possible to describe the wide set of activities, as described in section 1.1, which constitute the broad notion of integrated use of a knowledge based system in process engineering.

A brief review, mainly from the work of Blass(1986), of the concepts involved in the application of systems engineering to process development is provided here in order to introduce a novel approach to design methodology well suited to a sophisticated design environment, termed "appropriate and incremental parallelism".

From a systems engineering point of view, any "system", such as any chemical process or component thereof, can be analysed in term of its development, its realisation and its use. As a problem solving methodology, therefore, systems engineering embodies two basic ideas : a) the system concept and b) the systems engineering procedural model. The system concept simply refers to an arbitrary entity, called a system, which is separated from it's surroundings by a defined boundary. A system is composed of system elements which are "related to one another as well as to the surroundings via the system input and output". The definition of what constitutes a system is somewhat arbitrary in that the definition can be recursively applied to each sub-element until the required level of detail of analysis is obtained. This is a familiar notion to chemical engineers and provides a simple, generic way of describing a large range of widely differing "objects" such as particular items of equipment, an abstract unit operation, the concept of a flowsheet, process flow diagram, etc.

The systems engineering procedural model serves two functions. First, it labels the chronological development of a system, from the initial abstract process formulation right through to its concrete realisation, in terms of perceptible stages also known as life stages. In terms of process development these life stages will refer to the main process events such as "freezing of the process flowsheet diagram" etc. Secondly, the procedural model introduces a problem solving strategy which can, in principle, be successively reapplied to every life stage of the system. Figure 1.17 shows this in simplified form where the outer ring represents the generality of the procedural model

which can move over the various system life stages.

Figure 1.17: Simplified Procedural Model

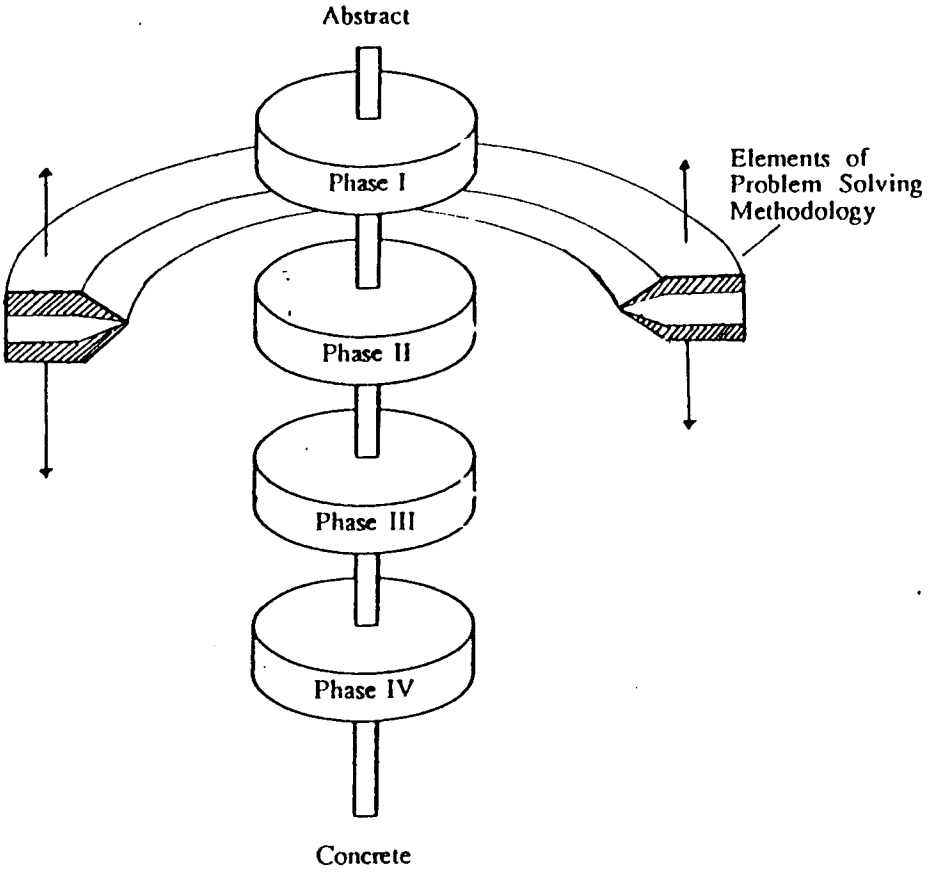
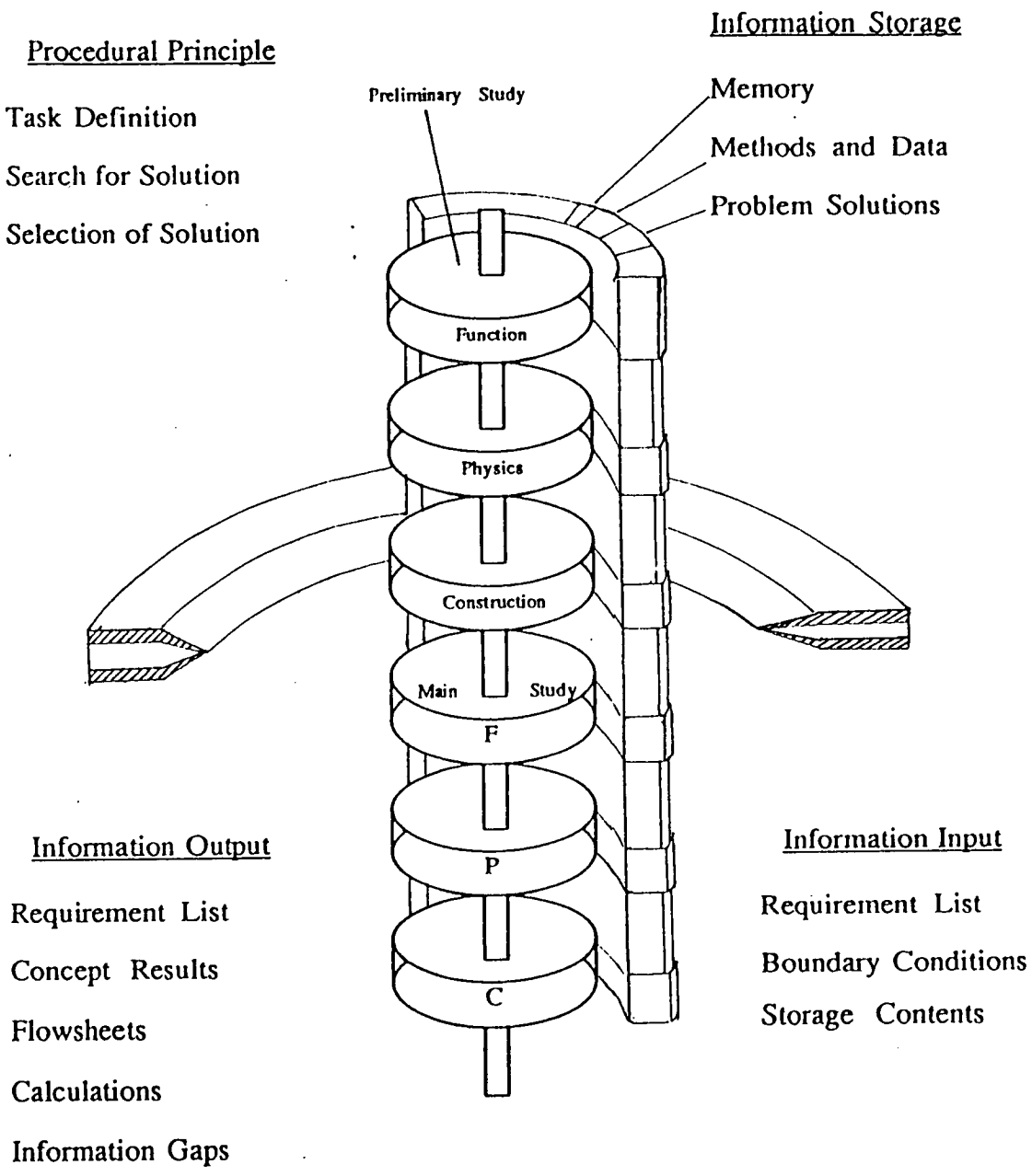


Figure 1.18 shows this in more detail, also showing an inner ring signifying the storage of knowledge (data, methods, etc.) required for each life stage.

Figure 1.18: Detailed Procedural Model



Note the presence of information "gaps " in the output, which will appear in the form of requests as there will nearly always be the need for experimental data in the development of a process. Blass(1983) considers in further detail the role of experimentation and its interaction with the procedural model. It should be also noted that the life stages are split into a preliminary study, a main study and detailed studies. The preliminary study provides an initial system from which further subsystems can be considered and defined. The main study must contain the concept

EX BIBL. UNIV. EDINBURGEN.

of investment decision and must also specify the necessary detailed studies that are to be subsequently carried and their relative importance. The concepts of project planning must therefore be also embedded in the systems engineering procedural model, which will be of considerable importance in later discussion. Blass then proposes three levels of data abstraction for use in these studies. These levels of abstraction prove useful in defining the level of detail that is required in defining a system, in analogy to the redefinition of schema as previously mentioned. The three levels are :

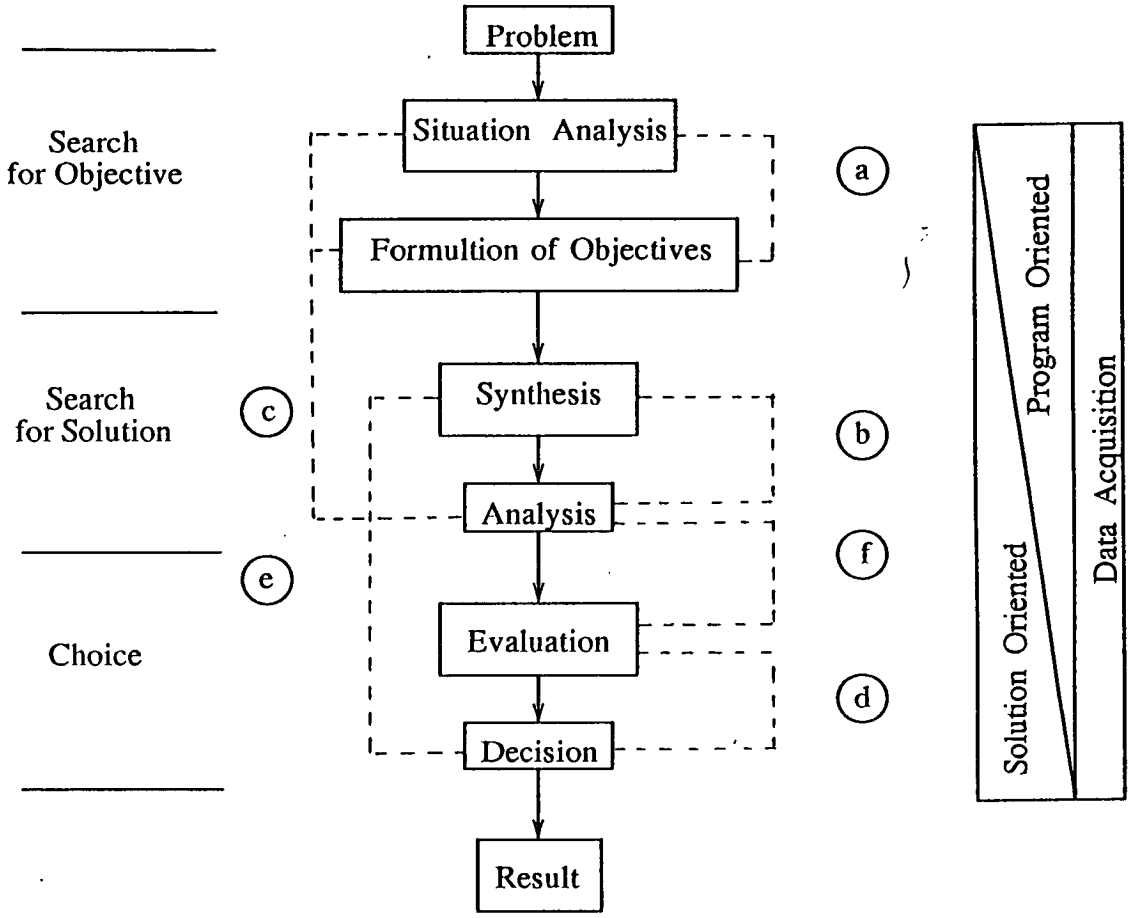
1. Functional Level - the initial task formulation must provide the global function of the process. This is further divided into particular functions, such as separation, reaction, mixing etc., which are combined to provide the functional structure of the system. The resulting level of representation corresponds closely to what one normally finds in a basic flowsheet or block flow diagram in chemical engineering.
2. Physical Level - in the preliminary study the functional structure is translated into physical and chemical effects which at the time of the main study can be further refined to the necessary level of detail. This level of detail will result in the formulation and specification of the process flowsheet, previously referred to the as the process flow diagram event.
3. Constructional Level - this level of detail corresponds to the detailed development of the piping and instrumentation diagram and beyond.

Obviously the specific classification of levels used in any one situation depends entirely on the application program developers. Nevertheless, they identify three abstract but convenient levels that can be developed in relation to the specification and operation of the procedural model. The need for relating data and knowledge to it s intelligent interpretation and use is a fundamental point which will be emphasised and expanded upon many times in this work.

The point of this short review, since most of the ideas are quite straightforward albeit in a new terminology, is to provide a theoretical framework within which it is possible to formulate formal control and operational mechanisms for a knowledge based system which adopts a structured approach to integrated process engineering. At the same time it is based on levels of representation which correspond closely to those described in the previous section. This is in stark contrast to the database approach where such management issues are not explicitly represented. From a process systems engineering viewpoint, this is a major argument against their continued development in present form.

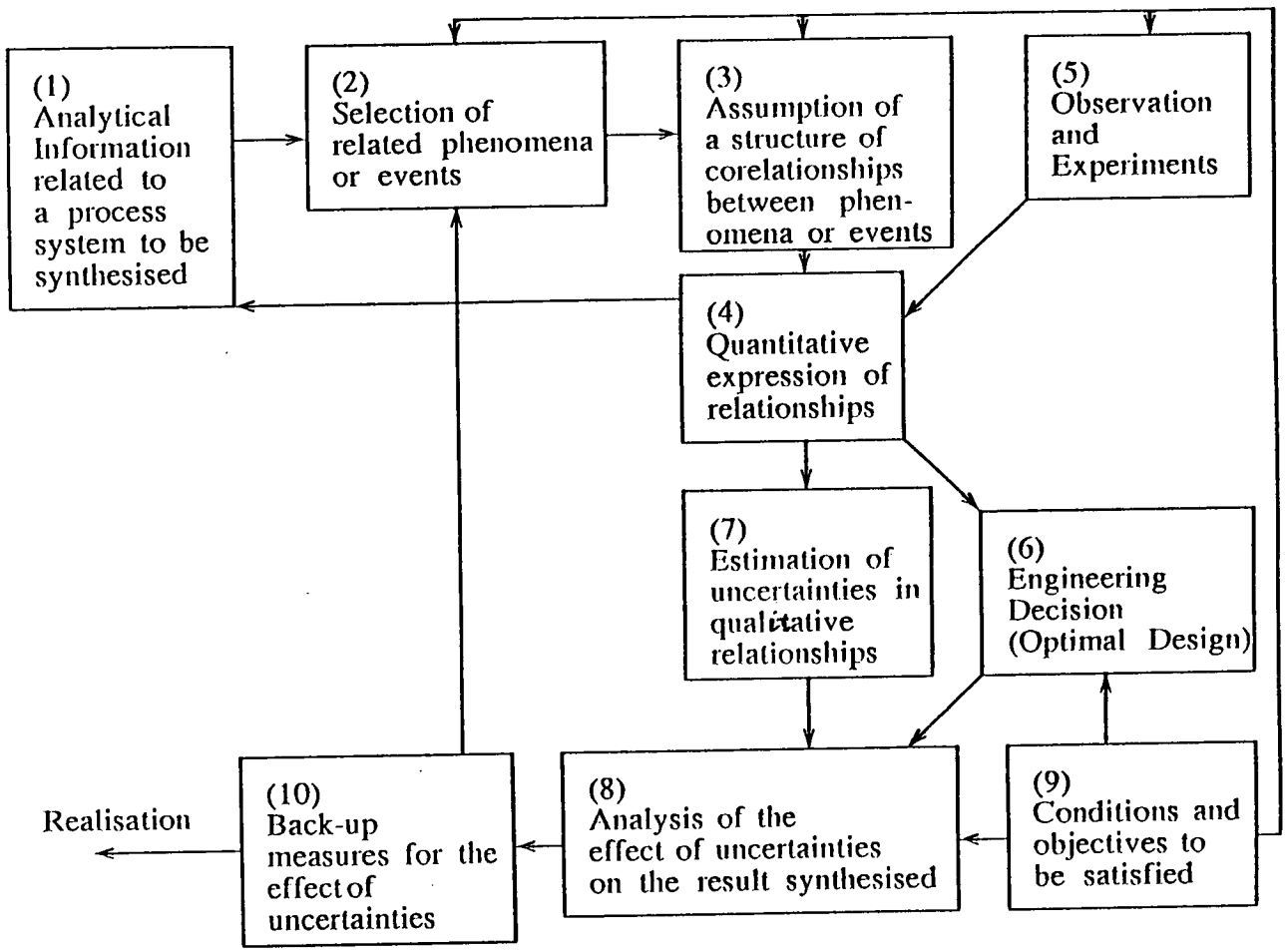
In terms of actually specifying the details of the procedural model Blass proposes a very general iterative method which characterises the typical "sequential/recycle" approach to design, as shown in Figure 1.19.

Figure 1.19: Procedural Approach of Blass



As indicated earlier the resulting effort and amount of data handling will d heavily on the order in which design tasks are formulated and solved. This that the definition of the procedural model must be very carefully considered if an efficient design methodology is to be developed. Takamatsu(1983) proposes a more detailed engineering formulation of this general method, see Figure 1.20, which is then applied to overall process design, detailed unit operation design, design for control and operability, and batch process systems.

Figure 1.20: System Analysis of Takamtsu



Both workers emphasise the importance of basing decisions within the wider context of the "design at that time". This is naturally catered for by taking account of any outer system which embodies the subsystem which is currently being considered if a convenient means of combining information between parent and child systems can be devised. Blass stresses the point that providing an explicit design methodology does not restrict the creativity of the design engineer. On the contrary, it provides a flexible framework which can be easily applied to novel or complex situations and must be "filled" in a creative way. In addition, the more novel or complex a process is, or the more it varies from known knowledge, then it is all the more important to provide a

methodology in order that a start can be made on the problem. This seems particularly relevant for the future development of novel processes, especially batch ones, and in face of the established trend of more highly heat integrated process plant.

The final feature of this introductory chapter is to introduce a novel design methodology called "appropriate and incremental parallelism", (AIP). This is based firmly on a process systems engineering approach to process development and represents a novel definition of a systems engineering procedural model for process design. The motivation for a fresh approach stems from the following observations :

1. As already discussed, there is a large but coherent flow of information throughout a design project, the processing of this taking a considerable amount of an engineer's time. This is complicated due to the interdisciplinary and collaborative nature of process design and the need to make design information easily accessible to all those concerned. It has been argued that the management of this information must be done in a way that not only ensures consistency of data but also ensures integrity of knowledge in terms of maintaining awareness and consistency of use of design intentions and assumptions.
2. The scale and complexity of most design problems means that they must be broken down into successively smaller sub-tasks. The overall size and management of the problem in conjunction with the inevitable trade-offs between design detail and available time for analysis means that important details can be overlooked. This can be compounded by the fact that good communication of skills and techniques can be difficult to achieve in a large company environment.
3. The traditional "sequential/recycle" approach to design and the less than perfect communication between design groups leads to a lack of awareness of what others are doing at a detailed level. This in turn often results in the unwelcome event of design modifications. Two typical activities which almost invariably give rise to such modifications are control system design and line by line hazard studies, Duxbury and Preston(1987).

It requires little thought to realise that the manner in which design activities are coordinated will directly dictate the speed at which the design can proceed and the effort involved. Traditionally, design methodology has been necessarily sequential in nature with some recycling of effort as the design proceeds in an iterative fashion. In addition, it should be well noted that this recycling of effort is often required precisely because certain activities such as control and operability analysis, detailed hazard analysis etc., are carried out in a sequential manner and are not fully considered in the early stages of design. Although it would obviously be inappropriate to consider these issues in too much detail at too early a stage in the design, considerable insight on the direction and feasibility of process design can often be made by a limited use of such activities. An alternative approach, therefore, is to use "incremental" design procedures allowing various expert engineers from different disciplines to contribute as much detail as is possible and "appropriate" to any stage in the design process. The essence of this approach is that issues such as

control and operability, process flexibility, hazard analysis etc., are considered in some detail throughout the whole of a process design. This is in contrast to the view that they are "downstream, revision" activities, as seems the case in traditional approaches to design. The difference between the two approaches is represented in Figures 1.21a and 21b below.

What is new is not the particular design activities involved but rather their interaction and the stage at which they become part of the overall design process. There are several advantages of this new approach worth discussing.

First, a much more thorough, well thought out design should be obtained by having expertise from various disciplines constantly monitoring the progress of the design. Thus, there should be far fewer "downstream" design changes encountered. Secondly, it provides an alternative means of tackling large scale combinational problems such as line by line hazard and operability studies (Hazop). Instead, the level of hazard identification can be continually refined as the design proceeds. At present, a detailed line-by-line hazop study can use up considerable resources, in terms of skilled engineers' time, in its completion. In addition, to the methods employed it is inevitable that a thorough plant wide investigation of both causes and consequences of hazards cannot be fully achieved. Plant scale problems such as these (including heat integration and many other optimisation problems) will no longer be viewed as large, difficult "one off" problems to be solved, but rather as an ongoing "learning" process.

In terms of "control" of the design activities, the expertise being constantly provided at the level appropriate to the stage of the design will prevent too much detail being introduced at too early a stage in the design. With regard to the systems engineering model a coherent parallel structure can be introduced into the design process which can be successively refined as the design proceeds. A parallel, broader based decision making process should also enable an easier means of ensuring both integrity of knowledge and data to be devised in view of the fact that there will be a much better awareness of what assumptions and decisions have been made in previous stages of the design. This is difficult to achieve in database systems where little attention is given to the overall structure and management of the design process.

The major difficulty with this approach is ensuring that the general design methodology and the sources of expertise can be combined in an effective manner whilst ensuring effective communication between the relevant design groups. The use of the word "incremental" is important here as it emphasises the important heuristic of adapting a stepwise progress through a design thereby enabling tight

Figure 1.21a: Sequential / Recycle Design Approach

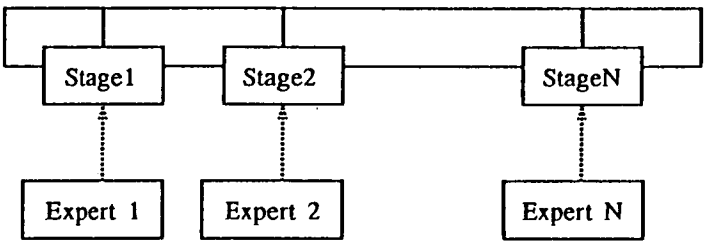
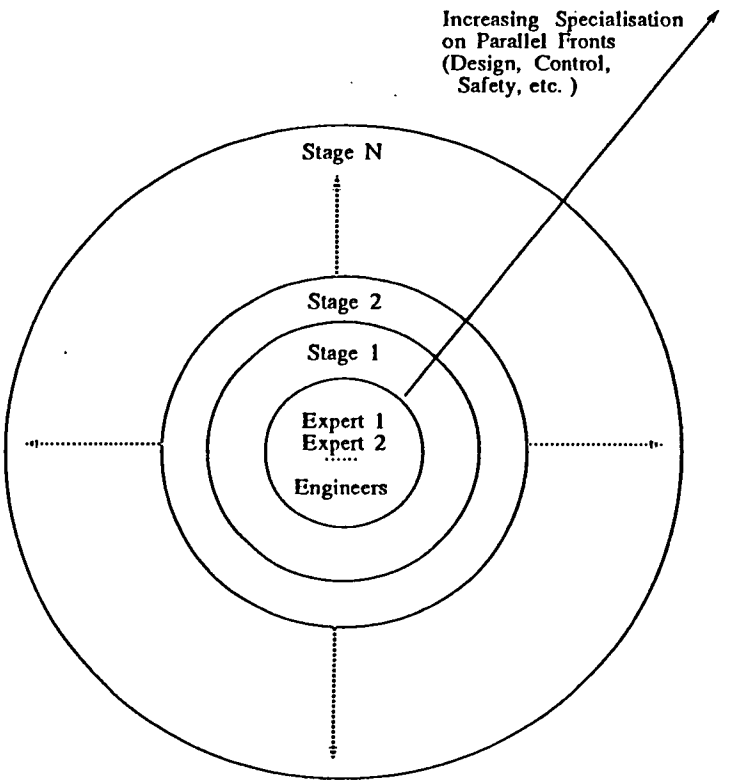


Figure 1.21b: A.I.P. Design Approach



control on the time spent on, relative importance of, and assimilation of knowledge from a variety of detailed design activities. Allowing engineers to work simultaneously on diverse problems and overwrite a common database in an essentially uncontrolled manner is a problem that must be avoided if general chaos is not to ensue.

The second problem with this approach is that it requires both the levels of semantic representation that were outlined in section 1.7 However, it is my opinion that this problem must be tackled anyway if we are to progress from the current situation and the use of DBMS software whether or not one adopts an "appropriate and incremental parallel design methodology" or not.

1.9 A Research Framework For Integrated Process Design

The main purpose of this introductory chapter was to make more formal the notions of integrated process design and the integrity of data and knowledge in that process. In particular, there is a need to get away from the simplistic idea that integrated process engineering merely consists of using software which provides the means to interface a collection or library of design programs. Rather, integrated process engineering, embodying the principle of knowledge integrity, must basically be viewed as a new approach to process design in terms of the handling, use and effect of information. Knowledge or data is no longer passive but must play an active or reactive role in the overall process. It has been suggested that a process systems engineering approach be adopted in which to formulate and develop these ideas in the prospect that a framework will emerge in which process engineering design activities themselves, their inter-relation, and their interaction with knowledge/data can be more formally defined. The use of database management systems as a means of achieving these goals was considered. Despite the limited success of such systems, many problems were highlighted which seem difficult to solve without substantially changing the nature of database management systems in terms of design and functional capabilities. In light of these criticisms several extensions to this approach have been preposed in terms of knowledge representation and the role that a knowledge based system should play in solving the problem of integrated process design. The following points, therefore, provide a summary of a long term research framework for integrated process design. The rest of this thesis has been based with these goals firmly in mind.

1. Develop representation techniques which are more closely suited to their use in process design problems. This must be investigated at three levels.

a. Description at at object, attribute and relationship level
b. Description of problem solving techniques in regard to consistency of knowledge at the object, attribute and relationship level.
c. Explicit representation of an overall design methodology and its application to particular decision making events.

2. Investigate what requirements an explicit design methodology places on the control and operation of a knowledge based system which attempts to maintain integrity of knowledge.
3. Investigate various formulations of design methodology which both reduce the design effort and most easily facilitate the integrity of knowledge within the design process to be maintained, e.g. an AIP approach.
4. Investigate the means to provide a simple, usable interface to such a system by a wide class of end user.

These goals are obviously long term indeed and as a consequence this thesis will only address itself to particular aspects of them.

It must be emphasised, and is fully appreciated by the author, that the breadth, scale and complexity of these issues precludes any short term solutions in terms of a usable demonstration system. In anticipation of following chapters, the complexity of the representation issues involved will mean bringing together many ideas from the fields of Artificial Intelligence and Cognitive Science. At times the theoretical issues involved may seem far removed indeed from process engineering. However, this is by no means an apology since it is my opinion that it is only when there is sufficient synergistic technical research in these diverse areas and the recognition of the specific needs of process engineering, will any real progress be made in the future development of knowledge based systems. It is hoped that the reader bear this point in mind given some of the more abstract or remote issues presented in subsequent chapters, albeit appearing "abstract" only to those unfamiliar with such ideas. The justification for this thesis, therefore, is that a useful start can be made on many of the basic problems, the solution of which will ultimately lie at the heart of future knowledge based systems for use in process engineering.

Chapter 2
Knowledge Representation in Artificial Intelligence

The previous chapter introduced a definition of the problem of integrated process design and the requirements it ultimately places on any software environment used to facilitate this problem solving activity. In particular, the notion of integrity of knowledge was defined and it was discussed how this concept should be adhered to if improvements are to be made in the use and operation of data and knowledge base systems during process design. A number of data base management systems have been produced to tackle various aspects of the integrated process design problem based on binary, hierarchical, network or relational data models as a means of data representation. It was concluded that such systems cannot easily be used to represent and manipulate the different types of knowledge that are required in any process design if integrity of knowledge is to be maintained. The purpose of this chapter is to introduce a variety of knowledge representation techniques taken from the area of Artificial Intelligence (AI) which seem, at least on first sight, directly relevant to the problem of extending the functionality of a database system to that of a knowledge base system. The review that follows is, for reasons of space, far from complete in that it covers a very wide research area and requires considerable background reading to appreciate the technical issues involved. As a result, complete subject areas such as logic or heuristic search techniques cannot be given the technical discussion they deserve. Instead, only the main aspects of each knowledge representation technique will be discussed with references provided for more technical reviews and papers on each topic. The later chapters will, however, assume some appreciation of the more technical issues involved.

2.1 Types of Knowledge Used in Process Design

Before presenting a review of knowledge representation techniques it is worthwhile discussing what types of knowledge will have to be represented in a process design environment. It was suggested in the first chapter that three levels of representation were required above and beyond simple data representation, namely

1. The ability to represent the principles or laws to which generated or experimental data must be ultimately related.
2. A means to represent the basis and intended effects of "operations" which manipulate data. This in turn requires an explicit representation of the process model and the assumptions upon which it is made.
3. Representation of the "operations" themselves. This applies to high level concepts such as the overall design methodology as well as specific low level solution techniques.

Although these levels do appear to capture whats involved in maintaining integrity of knowledge they are still too vague to be of use at a program specification level. In anticipation of arguments presented in chapters three to five, the following three sections discuss the minimum representation levels required in any advanced design environment. These ideas are based on a consideration of the activities involved and the documents and data produced in any process design (see chapter 1, section 1.2) as well as the basic mechanisms underlying items 1) to 3) above.

2.1.1 Objects, Attributes and Relations in Process Models

One of the most basic requirements of any system will be the ability to represent conveniently the process model throughout the lifetime of a process design. At it's most simple level, this will involve representation of the constituent parts of the process irrespective of whether they are undefined abstractions in the early stages of design or specified unit operations in a block flow diagram or, ultimately, detailed equipment and piping specifications. The abstract notion of an "object" shall be introduced to cover this spectrum of process description and can be thought of as a generalisation of the entity types normally encountered in typical process database systems. Since the notion of an object is merely an abstraction it can also be conveniently applied to the wide range of documents produced in a design project irrespective of graphical or textual form. Thus a flowsheet representation such as an ELD could be an object in its own right as opposed to a system in which it is merely an ad hoc graphical output resulting from arbitrary operation on a database.

A seemingly natural way of describing any such object is in terms of its attributes, that is ascertainable properties of the object which can be used to identify it. Thus, a chemical species can be treated as having an attribute molecular weight, a distillation column having a reflux rate, a process having a design capacity, etc. This idea is familiar enough from standard database specifications. However, it has already been noted (see section 1.4, chapter 1) that in specifying a database it is by no means clear what can be taken as an entity, an attribute or a relation. Similarly here, the problem of what can be considered to be an object or an attribute turns out to be a serious problem in knowledge representation and one which raises deep fundamental issues from a cognitive science point of view. The discussion of these issues is delayed, however, until the review of current AI techniques has been presented. The third fundamental requirement of any design system is the need to represent the relationships involved between the various constituent parts of the process model. At a simple level there will be object to object relationships to represent, for instance, the overall structure of the process. However, the notion of an abstract "relation" goes much further than is normally found in database applications. Abstract relations have

to account for the interrelation between attributes within an object, object to object relations as well as higher order relations which restrict or control the use of these simpler relations. For example, if the flowrates into and out of a column are represented as attributes then there must be an explicit relation describing their relational or "functional" dependence on one another, namely an overall mass balance. Further, this relation must link in some way to the more general relationship on which it is based, i.e. the principle of conservation of mass. It is the specification of this relation, or yet more general relationships, which determines how applicable the lower level relation is to the object in question. Thus, although simplifying relationships, such as the ideal gas law, can be used in the description of an object's attributes, e.g. a water-ethanol flash vessel, there will be an explicit representation of the applicability of the use of this relation. The specification of attributes and object-object dependencies is, therefore, no longer an ad hoc decision for the programmer. Rather, these decisions must be accounted for in an explicit relational manner.

The earlier requirement that one needs to be able to represent both the basis and intended effects of "operations", such as the calculation of process model data, as well as represent the overall design methodology merely results in an extension of the notion of an abstract relation, albeit to a very general level indeed. Low level operations, modelling reactor conversion for example, can be represented at an abstract level as a, perhaps complex, combination of relations to be satisfied. Suitable selection of such activities can be repeatedly combined into yet more general "relations" until the required description of any particular design methodology is achieved. At this level of abstraction it is perhaps more useful to describe a "relation" as a "relational concept". This is defined more fully later but for the moment it is only necessary to appeal to the familiar notions of the concept of individual column tray design, overall column design, downstream separation section design, etc. These activities, at different levels of design debuits, can be considered to be relational concepts in their own right, the interaction between more detailed activities being accounted for in the more general relations or relational concepts. It is worth noting that the notion of an abstract relation given here is wholly consistent with what is required for a systems engineering description of process design activity given in chapter 1.

2.1.2 Functional, Event and Situational Knowledge

The concept of an abstract relation is powerful indeed, but perhaps slightly too general to be considered at this early stage. Following Barr and Feigenbaum(1981), it is worthwhile identifying three, not necessarily distinct, types of relational

knowledge.

1. Functional knowledge -representation of any process design activity must involve knowledge of how to perform the activity and achieve it s intended function over and above the representation of the specific objects involved. This type of information will be referred to here as functional knowledge, alternatively termed performance or procedural knowledge elsewhere. (The term functional is chosen given the set theoretic definition of a function in terms of a relation and the emphasis this puts on the fact that there is an operation or transformation from one set of object descriptions to another).
2. Event Knowledge -given that one can represent what constitutes an activity, it is then desirable to represent the notion of an event. An event in this simplified context will be used to represent the fact that an activity has occurred, is occurring or will occur at some point in the future. Using this definition -a wide variety of process design information can be represented in a standard way. For example, the event may be a hypothesised one such as a valve closing in a dynamic simulation, or, at a higher level, the recognition of the completion of preliminary design of a unit operation within the overall design activity. High level descriptions of activities and events can also be used to form the basis of cause and effect descriptions of some relations. Cause and effect descriptions are an important part of understanding complex physical phenomena. A relevant example might be the recognition of the possibility of thermal runaway in reactor design and taking appropriate measures to minimise this problem based on a cause and effect description of this phenomenon. It should be noted that this type of information has introduced the need for a time ordering relation in the representation. In more general terms, an event occurs whenever a relation is involved which results in either a change in the representation of the objects of interest or a change in the state of relationships between objects.
3. Situational Knowledge It is useful to combine both functional and event knowledge into higher level relations which shall be labelled "situations". One aspect of intelligent behaviour is the ability to recognise certain situations or scenaries and to be able to predict which actions and events should be taken or avoided in order that an individual benefit from that situation. Similarly, it is highly desirable to view design activities in the overall context of the stage a design has reached. Design decisions can then be taken given a global view of the situation rather than based solely on a narrow, local viewpoint. The representation of well known situations would also seem to be a necessary precursor to be able to respond in an intelligent way to sudden or unexpected changes to design procedure. This sort of knowledge is termed here situational knowledge and will involve an extension or generalisation of functional knowledge described in 1).

2.1.3 Self Awareness

Meta-knowledge is a term frequently used in a simplistic way in the literature on knowledge based systems, loosely referring to knowledge about what's already known, ie knowledge about knowledge. This circular definition, however, is of questionable use in characterising knowledge since it is then hard to avoid the introduction of what would be layer upon layer of meta-meta-knowledge. What is of importance is to recognise that engineers are aware, for example, of the scope and origin of their expertise in a particular area, the reliability of technical data, the relative importance of types of design information pertaining to the solution of a

design problem, etc. An explicit awareness of such "higher level" knowledge is an important part of the process of responding in an intelligent manner to design problems and can often be instrumental in determining how an individual will proceed when faced with difficult tasks. This type of knowledge is termed self-awareness here, as it more properly emphasises the role of the individual within a problem solving activity and how this affects the manner in which lower level knowledge shall be used in arriving at this solution.

Another important part of any design methodology will be the description of what type of person is competent to take particular kinds of design decision and how design engineers from various backgrounds will be expected to participate in complex activities, eg. the role of process engineers in control system design. The concept of self awareness can also be used to describe the relationships between the design activity of any particular engineer and fellow engineers involved in the same project. It is precisely this type of information that is required to tackle the problem of integrity of knowledge in a multi-user environment as discussed in chapter 1.

2.1.4 A.I. Knowledge Representation Techniques

The previous section described in an abstract form the types of information that should be represented in a process design environment. The question of how to represent these ideas in a form that can be manipulated in a useful way by computers is a research problem central to many areas of Artificial Intelligence, namely that of knowledge representation. The question of what actually constitutes knowledge or a representation of it raises deep philosophical issues. Some of these issues will be explored later but for the moment the following simple points should be considered. Any representation of knowledge in a computer must involve the use of low level data structures to store information. Complex data structures can be described in yet simpler structures, and the process repeated until a level of description is reached involving only the most basic symbols that the computer manipulates. Viewed independently of the mechanisms that manipulate the data, it is clear that the data structures in themselves do not constitute "knowledge". In the same way, Barr and Feigenbaum make the point that a book is a potential source of knowledge but without a reader to understand it s contents, the book is no more than ink on paper. Of fundamental importance, therefore, in knowledge representation research is the construction and manipulation of relations between symbolic data structures which result in program behaviour that appears analogous to certain aspects of human behaviour. In this context, Barr and Feigenbaum suggest that "a representation of knowledge is a combination of data structures and interpretive procedures that, if used in the right way in a program will lead to knowledge behaviour".

Winston(1984) defines a representation more specifically as "a set of syntactic and semantic conventions" which can be used when interpreting symbolic structures. The rules of synbax might describe how structures can be constructed from simpler structures or primitive symbols, while the semantic conventions would indicate how to assign meaning to a symbolic structure as some function of its constituent parts, the birth of a proposition for example. The importance of the co-existance of interpretive mechanisms and data structures cannot be over emphasised as it is the resulting interaction which determines the reasoning capabilities and, hence, ultimate use of any knowledge based system.

The review that follows in the reset of this chapter briefly discusses the types of representation and reasoning techniques found in the A.I. literature deemed relevant to the design of a knowledge based process engineering environment. More detailed introductions can be found in Barr and Feigenbaum(1981) Jackson(1986,1987), Charniak(1985), Winston(1984), Frost(1986), and Nilsson(1982). Much of the early A.I. research was based on two areas :

1. graph theory, state-space representation, and heuristic search techniques.
2. predicate calculus, i.e. first order logic, and the development of automated theorem proving techniques.

Most of the current, state of the art representation techniques developed from a thorough understanding of these more mathematically formal topics. In order to compare different representation techniques, the main aspects of these two areas shall be discussed first. Subsequent discussion will employ state-space and logical descriptions where possible in order to highlight the similarities and differences between the representation schemes.

2.2 Heuristic Graph Theory

A formal definition of graph theory nomenclature can be found in standard mathematical texts, e.g. Gibbons(1985) or Boffey(1982), and is not discussed in detail here. A brief summary of the relevant terminology and basic definitions in Appendix A for those unfamiliar with the subject.

An informal definition of a graph is that it is a set of points in a space interconneted by a set of lines which indicate some relation between relevant points. Mathematical descriptions refer to the set of points in the graph as vertices and the lines as edges. In the A.I. literature it is more usual to describe a graph being constructed from abstract entities called "nodes" and "arcs", the arcs representing the connections between the nodes. In addition, it is frequently the case that "labels" are attached to both nodes and arcs. There is no inherent restriction on the form or definition of a label and it may well be a complex structured entity in its own right. Given the definition of a graph in Appendix A, it is worth noting that if L is a set of labels associated with a NxN graph, then any subset of NxNxL is often referred to as a labelled graph or a network.

In many applications the nodes and arcs are used to represent different kinds of information. This is useful in that a variety of representation and reasoning, or inference, techniques can then be described and compared in a more formal, mathematical notation, as is shown in the remainder of this chapter.

Much of the early work in A.I. and Operations Research during the 1960's and 1970's was concerned with applications of heuristic search techniques in game playing and optimisation. A typical example is that of chess playing programs. One of the problems in writing such a program is how to represent the game as it progresses and how to represent the players' moves. The general technique adopted was to express this information in the form of a graph or a tree. In the case of a chess program each node could represent the state of the pieces on the board and the links between nodes would indicate each players move. The resulting graphs or trees to be searched, however, are usually very large. The term "combinatorial explosion" is used to describe the rapid growth in the size of the search space as more moves are considered. Choice of a suitable heuristic search technique is extremely important, therefore, if reasonable performance is to be achieved by a game playing or optimisation program.

Many other problem solving processes can be represented in tree or graph form in one of two general forms, namely state space representation and problem reduction representation.

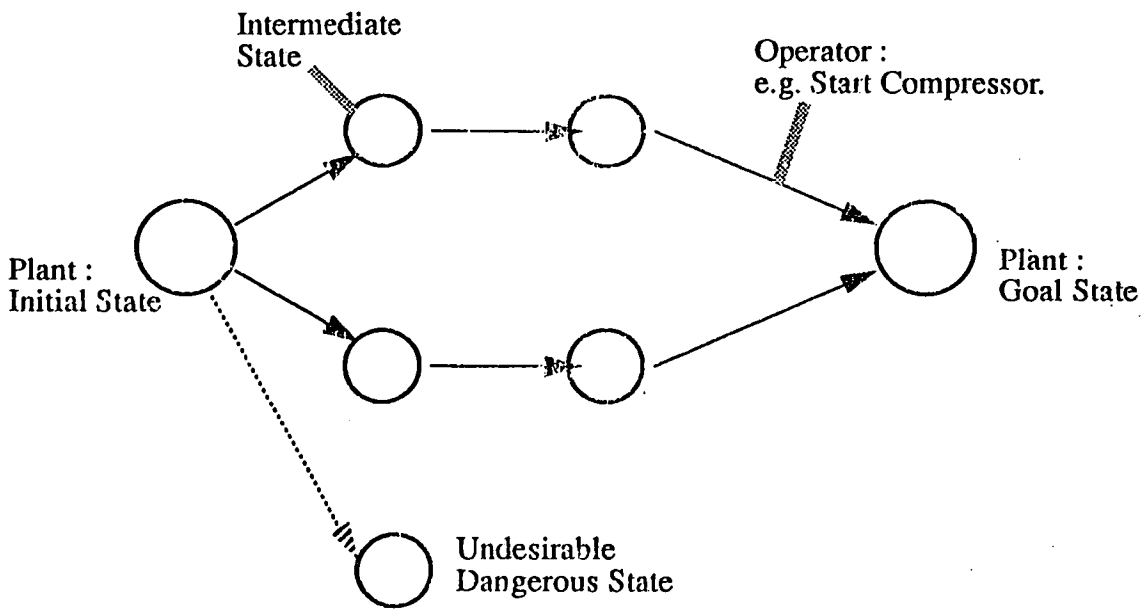
1 State Space Representation

A state space description of a problem involves a set of initial states of a "world", a set of goal states of a world, and a number of operators which can be used to transform one state to another. The initial states represent the starting point or initial conditions of the problem and the goal states represent the desired solution, usually described as the satisfaction of a set of constraints. The problem solving process is thus seen as one of choosing appropriate operators to transform the initial state to the goal state via a sequence of intermediate states. Each state in this process can be thought of as a node in a graph or network, the arcs representing the applicable operators used to transform on state to another.

In general, there will be a number of pre-conditions attached to the use of each operator which will determine its applicability depending upon the current state of the world. There must also be a set of post-conditions for each operator describing how it modifies the state of a world should it be selected. State space representations can be used in a wide range of constraint satisfaction problems and sequence seeking problems. Problems are viewed as finding paths between selected nodes in a state space graph.

A simple process engineering example conveniently described in these terms is the problem of representing and reasoning about the start-up procedure of a chemical plant. The initial state represents the units comprising the plant, the goal state representing the plant running under design conditions, see Figure 2.1.

Figure 2.1: State Space Representation of Plant Start Up



The arcs in the graph represent operations, i.e. start-up decisions, like open a valve, start a pump etc, which gradually move the plant towards design operating conditions. The goal state is reached when these operating conditions, or constraints, are satisfied. The pre-conditions attached to operators will normally include both safety and economic considerations, eg "is it safe to open a valve?", "delay costly side effects as long as possible", etc. More detailed evaluation of these constraints would require consideration or simulation of the dynamic response of the plant.

As a result, a realistic implementation of this problem solving process would require extremely sophisticated search techniques and extensive background process engineering expertise in order to minimise the resulting search space. Nevertheless, it is easy to see how this type of process design activity can be be described using a state-space representation. Fusillo and Powers(1988) have attempted to formulate this problem using a state-space representation as described here.

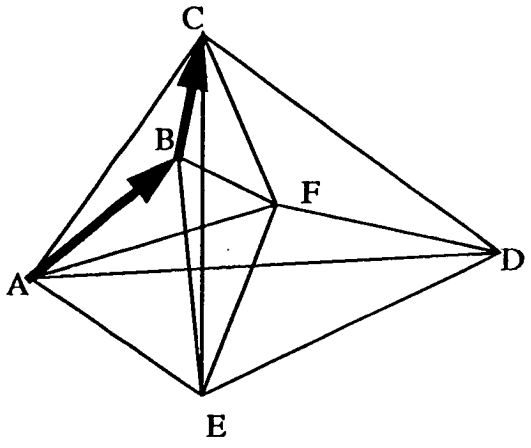
It is important to notice that there will usually be many ways of transforming the initial state to the goal state, i.e. paths between the initial node(s) and final node(s). This is certainly true in the case of most synthesis problems in process design whether it be synthesis of start-up procedures or selection of distillation sequences to separate mixtures (e.g. Hendry and Hughes(1972)), etc. If one solution path is

somehow better than another and one wishes to find the "optimal" solution path then the degree of complexity of solution becomes very high indeed. Many problems posed in this way belong to a class of problems termed "NP-hard", see Gibbons(1985).

All known algorithms for such problems require exponential time in their solution so unless very good bounding functions or heuristics can be found, non- optimal solutions have to be accepted.

The description of the state space representation above is somewhat informal. Following Pearl(1984), a more precise description is possible. Consider the graph in Figure 2.2 in which the partial solution evaluated so far involves one path through nodes A,B,C.

Figure 2.2: Partial Solutions in State Space Representations



Programs which use a state-space representation require a data structure to represent the partial solutions, or alternatively, the subsets of potential solutions called "candidate subsets", e.g. {A,B,C} in this example. Included in this data structure is additional information which "explicitly defines the remaining subproblem" to be solved resulting from this candidate subset. It is the code specifying this additional information that is called a "state".
Further "the set of all subproblems obtainable by executing some sequence of refinement operators from a given position is called a state space". The state- space graph refers to the elements of this space which are connected by arcs labelled by the appropriate refinement operators.

In Figure 2.2, the path A->B->C is a candidate subset, and the connections to the

right of C represent the state space graph. The nodes represent the set of all subproblems that may result from the path leading to that node and are merely labelled A,B,C etc for convenient reference.

As Pearl points out, it is worth noting that the candidate subset is both necessary and sufficient to completely specify the remaining problem whereas a state is both redundant and incomplete. A solution of the subproblem does not identify the solution of the overall problem. One of the main reasons for maintaining an explicit representation of a state is that it can dramatically reduce the computational cost of making heuristic estimates of the amount of work still to be done in achieving a solution. As such, a good state representation can provide a reliable and effective means of controlling and directing search over large state spaces. In addition, if different candidate subsets result in identical state subproblems, i.e. different paths leading to the same intermediate node, then the candidate subsets can be compared and the inferior one can be ignored in any future searches. This reduction in search effort is called "pruning by dominance", see Ibaraki(1977) for more detailed discussion.

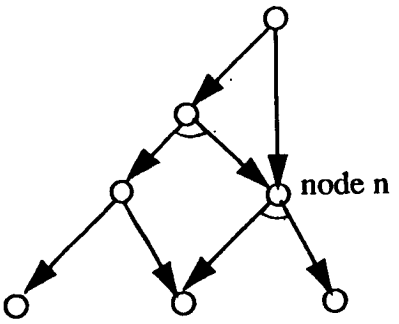
2 Problem Reduction Representation and AND/OR Graphs

The principal feature of the problem reduction representation is that the overall problem can be expressed as a goal or a logical combination of goals to be reached. The more general goals are expressed as a combination of more specific goals, these in turn being composed of yet simpler, more specific goals. The basic approach, therefore, is to decompose goals in a recursive fashion until a set of primitive goals are obtained, the solution of which can be stated in known facts, actions or operations. It is worth noting that this corresponds closely to basic problem solving notions adopted in systems engineering, as described earlier in section 1.8. The main point to note is that during the decomposition process each residual subproblem or goal created by some subset of candidate solutions can be viewed as "a conjunction of several subproblems that may be solved independently of each other, may require specialised treatments, and should, therefore, be represented as separate entities", Pearl(1984). As a consequence, individual nodes in the search-space graph of a problem reduction representation indicate subproblems or subgoals to be solved as part of an overall solution. There are two types of links connecting the nodes in such a graph:

1. Links that represent alternative approaches to the solution of the problem node from which they emerge. These are termed OR links and are identical to those found in state-space graphs.
2. Links which connect a parent problem node to the conjunction of individual subproblem nodes of which it is composed. These are called AND links and have no equivalent in state-space graphs.

Figure 2.3 shows an example of an AND/OR graph.

Figure 2.3: A Simple AND/OR Graph



The curved lines indicate which sets of subproblems must be solved in conjunction. When a node issues either OR links or AND links, but not both, then it is usual to refer to the nodes as OR nodes and AND nodes respectively. Mixed nodes, e.g. node n in Figure 2.3, can always be reformulated as pure AND and OR nodes by introducing dummy nodes if necessary (replace AND links by a dummy AND node which is then OR linked to the original mixed parent node. The parent node becomes an OR node and the dummy AND nodes maintain the original AND links). Unlike a state-space formulation, the solution to the problem is no longer a leaf or tip node in the graph. The problem solution is represented by an AND/OR subgraph, referred to as the solution graph. A more formal definition of solution graphs, AND/OR graphs, which are actually hypergraphs, and common terminology such as k- connectors, terminal nodes etc, is given in Appendix A.

Many problem solving tasks can be represented using the problem-reduction representation. One important example in A.I. is that of logical reasoning and automated theorem proving, as discussed later in section 2.3. In a process engineering context, many standard design techniques can be superficially represented in this form. The design of a distillation column, for example, or even an entire plant for that matter, can notionally be described as the solution of its constituent parts in conjunction with the satisfaction of overall constraints such as heat and mass balances. It is somewhat unlikely however that large problems can be solved directly in this manner, see section 2.2.5. The point to note is that problems

that can be identified and decomposed into independent subproblems are easily handled in this general framework. As in the case of state-space search, extremely well guided search strategies are needed to minimise the effort needed in producing a solution. In addition, it is usually the case that the search space is too large to be explicitly represented in computer memory. Procedures are needed which allow a program to generate a graph incrementally in any required direction. In a process design problem, this may simply be in the form of database lookup of standard design procedures or alternatives, e.g. "when designing a separation column consider both packed bed and hydraulic tray designs".

2.2.1 Uninformed and Heuristic Graph Searching Techniques

It was emphasised earlier, section 2.1, that a representation technique must be considered in light of the inference or reasoning mechanisms which operate upon it. Any inference mechanism must have access to appropriate background general knowledge and specific heuristic guidelines if it is to be successful in finding solutions to problems. In the worst case, inference techniques, for representations based on state space or problem reduction formulations, having no or little access to such knowledge will degenerate into standard graph searching algorithms.

It is worthwhile briefly discussing the characteristics of the more well known techniques for several reasons:

1. Their complexity and computational efficiency can be formally defined, hence, providing base cases for comparison with new or more complicated search techniques.
2. The very general description of state space and problem reduction representations means that standard searching algorithms arise in a variety of guises. It is important to identify and discriminate between basic graph searching algorithms and more advanced techniques.
3. The limitations of all known algorithms when dealing with problems that are NP- hard are well documented, see Garey and Johnson(1979). It is therefore, important to identify such problems when they arise.

A search strategy is a procedure for determining the order in which nodes in a graph are to be generated. If an arbitrary scheme is used to specify the order in which nodes are expanded then the search procedure is called uninformed or blind search. This happens when there is no information available at nodes in the graph other than the goal node, to affect the ordering decision. The progress of the search, therefore, depends neither on the nature of the solution nor on the nature of the unexplored region of the graph.

The converse to blind search is informed or heuristic search. This type of search will attempt to use partial information about the problem domain and the nature of the

goal to help guide the search toward the solution. Following Nilsson(1982), it is possible to describe a general procedure which covers both these situations and is shown in Figure 2.4.

Figure 2.4: Nilsson's General GRAPHSEARCH Procedure

```
1 Create a search graph G consisting solely of the start node s.
  Put s on a list called OPEN.

2 Create a list called CLOSED that is initially empty.

3 LOOP : if OPEN is empty then exit with failure.

4 Select the first node on OPEN, remove it from OPEN
  and put it on CLOSED. Label this node n.

5 If n is a goal node then exit successfully with the solution
  obtained by tracing a path along the pointers from n to s
  in G. (The pointers are set up in step 7).

6 Expand node n generating its set of successors M that
  are not already in n's ancestor set. Install these members
  of M as successors of n in G.

7 Establish a pointer to n from those members of M that were not
  already in G (i.e. neither in OPEN or CLOSED). Add these members
  to OPEN. For each member of M that was already in G decide whether
  to redirect its pointer to n.
  For each member of M already on CLOSED decide for each of its
  descendants in G whether to redirect its pointer.

8 Reorder OPEN either according to some arbitrarry or heuristic
  evaluation scheme.

9 Go To LOOP.
```

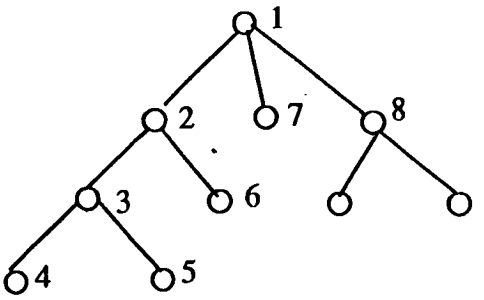
The terminology used is covered in Appendix A. In particular, nodes that have been expanded, i.e. those whose successor nodes are known, are called CLOSED. Nodes which have been generated but yet to be expanded are called OPEN.

An important point about all such strategies is that they maintain an explicit record of the generated search graph G. A subset T of G, called the search tree, is defined by the pointers or links set up in step 7 of the procedure. A node in G, other than the start node, has a pointer directed to just one of its parents in G, and this is its unique parent in T. All possible paths to a node that are generated are preserved in G whereas T only preserves the final chosen path to any node as defined by the pointers.

The most important step to consider here is step 8 in which the nodes on OPEN are reordered according to some arbitrary scheme or using a measure of heuristic merit. There are two classical approaches adopted in uninformed search, namely a depth first approach and a breadth first approach. In a depth first approach the strategy adopted in Step 8 is to expand the deepest nodes in the search tree first, i.e. the most recently generated nodes are put at the front of the OPEN list, the first of which is

then itself selected for expansion. This recursive procedure is blindly adhered to until the goal node or a dead end, which usually means failure, occurs. Upon failure the next most recent node on the OPEN list is selected for expansion and the whole process repeated. The path of the closed nodes maintained by the search program is called the traversal path, an example of which is shown in Figure 2.5. The node numbering indicates the usual depth first, left to right search amongst nodes.

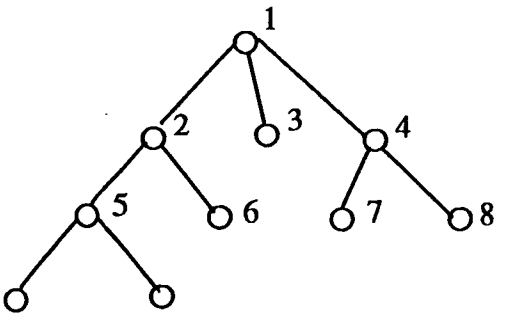
Figure 2.5: Depth First Traversal of a Tree



The depth first approach is simply a last-in-first-out policy which guarantees that no node at depth D will be expanded whenever nodes of depth greater than D still remain on OPEN. Nodes at the same depth can, in general, be ordered arbitrarily. In order to prevent the strategy from expanding any particular path too deeply, it is usual to provide a depth bound which limits the depth to which one parent node can be expanded. Since no information is available, however, it is impossible to estimate how far the last node visited in the stopped path lay from the goal node. The node may be just one step from the goal or many.

The second basic uninformed strategy is breadth first search. In this case nodes are ordered in OPEN in increasing order of depth in the search tree. Nodes of lowest and equal depth are moved to the front of the OPEN list. On expansion their successor nodes will get placed at the end of the list due to their greater depth in the search tree. A typical traversal path is shown in Figure 2.6.

Figure 2.6: Breadth First Traversal of a Tree



The breadth first search is, thus, a first-in-first-out policy, priority being given to nodes at the shallower levels in the search graph and recursively expanding sections of the graph in layers of equal depth. Unlike depth-first search, a breadth-first approach will guarantee to find the shortest length path to a goal in a locally finite graph, if such a path exists. If no path exists, both methods will fail on finite graphs or never terminate for infinite graphs (unless depth bounding is used). Ross(1985) reports that if the depth of the goal node is less than the local depth of the tree, then as far as the number of nodes searched is concerned, the depth first approach is likely to be better on average. It is impossible to guarantee this, however, in any particular case.

A widely used variation of the breadth first search is the "uniform cost" strategy. In this case the search graph is expanded in layers of equal cost rather than equal depth, where the cost is the cost of the path from any given node back to the start node. The node with the highest cost in OPEN determines how far other nodes will be expanded. A path expansion of any node is halted when we reach or exceed this value.

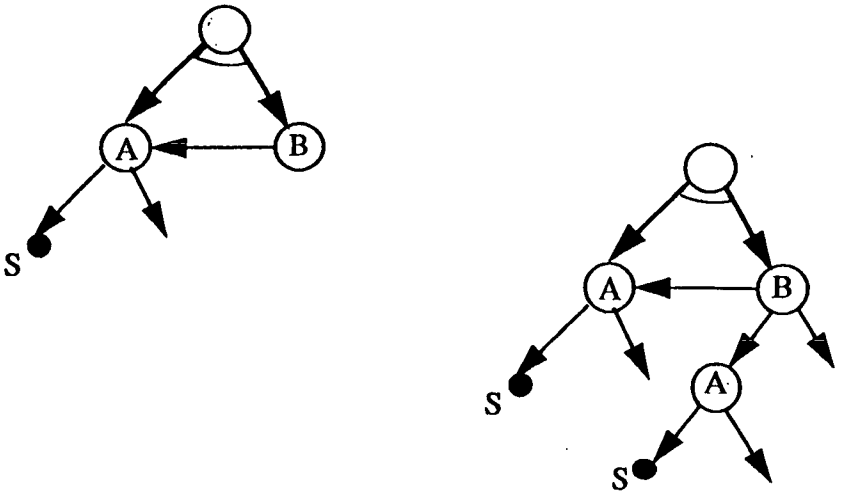
An important and widely used variation of the depth first search is "backtracking". In this version, when a node is selected for expansion only one of its successors is generated. This newly generated node is in turn expanded to one of its successors, and the process repeated until termination or failure. Upon failure the closest ancestor node, which is not yet fully expanded, is again selected for expansion and a new successor generated. The last-in-first-out policy now applies to node generation rather than node expansion.

The backtracking method is not strictly a graph search strategy since there is not an entire record of the search space. Only one (partial) path to a goal node is ever

maintained. The correspondence between the two schemes would be exact if the depth first process only ever generated one successor at a time. An important application of the depth-first/backbreaking approach is in the searching mechanisms embodied in the programming language PROLOG, introduced shortly in section 2.3.6.

Both the breadth-first and depth-first strategies can be adopted to search AND/OR trees in problem reduction formulations. The main difference arises in determining the termination conditions, as it is now necessary to know whether a given set of solved nodes represents a solution or not. Backtracking strategies cannot take full advantage of the information expressed in an AND/OR graph, as they only maintain a single traversal path at any one time. As a result identical subproblems encountered on different paths will be repeatedly solved unless modifications are made to the technique. Figure 2.7 shows a simple example of this phenomenon.

Figure 2.7: Repeated Problem Solving during Backtracking



The problem of recognising identical sub-problems also arises if cycles in the graph are to be detected and broken.

2.2.2 Informed Heuristic Search

The uninformed search techniques do, in principle, provide a solution technique to any path finding problem provided the problem can be represented in a convenient form. In practice, the techniques are likely to be infeasible for any non-trivial problems as the potential search spaces that can be generated are often very large. In order to control the way in which the search method expands nodes in the search space, and reduce the search effort, is necessary to make use of "heuristic" or

task-independent information. Formal statements of domain specific information are usually referred to as "heuristics" or rules of thumb. Pearl(1984) gives the following definition: "Heuristics are criteria, methods, or principle, for deciding which among several alternative courses of action promises to be the most effective in order to achieve some goal. They represent compromises between two requirements: the need to make such criteria simple and, at the same time, the desire to see them discriminate correctly between good and bad choices".

This definition raises two interesting points:

1. Heuristics cannot, in general, guarantee decisions made but can only hope to be effective most of the time.
2. The evaluation criteria should be relatively simple to compute in comparison to the total amount of effort expended in the search process.

The question of whether a useful set of heuristics can actually be found for many aspects of process design is an interesting one. The nature of the heuristics will vary from problem to problem. In design problems, such as distillation column design, one possibility may be to estimate the difficulty of solving sub-problems represented by a node. Flow sheet development techniques, such as the top-down hierarchical approach of Douglas(1988) could estimate the quality of the set of candidate solutions represented by a node according to some criteria such as avoid large gas recycles if possible, avoid large inventory hold-ups of material, especially if toxic, etc. In preliminary design or hazard studies it may be useful to estimate the amount of information that will be gained by expanding a node and the relative worth of this information in terms of guiding the overall search strategy. In many cases it will be difficult to formulate general, consistent sets of heuristics.

Irrespective of the type of heuristic, the potential worth of a node, n, can be estimated by a "heuristic evaluation function", f(n). The most important point about any such function is that it should depend not only on the description of an arbitrary node n, the goal node description, and the information gathered by the search so far, but also on background knowledge relevant to the problem domain. It is this background knowledge which will be required to discriminate intelligently between conflicting advice arising from the simplified heuristics, or to know when to suspend or discard a particular line of reasoning.

Heuristic or informed search techniques are those which use such a function to re-order the nodes in OPEN in step 8 of the GRAPHSEARCH technique. The study of the formal properties of these techniques has received much attention in the areas of Operations Research and AI. The following two sections very briefly outline the classical algorithms in this area, which is conveniently split into the search of OR

graphs (the A* algorithm) and AND/OR graphs (the AO* algorithm). These are worth discussing as many well know techniques, such as branch-and- bound search, dynamic programming etc. used in process synthesis studies, and aspects of "expert systems" (to be introduced in section 2.4) are just specialisations of these algorithms.

2.2.3 Heuristic Search in OR graphs

One of the most general heuristic techniques for searching state-space graphs is the Best First (BF) technique. In its most general form, the evaluation function f(n) is an arbitrary real valued function which, by convention, orders the nodes in OPEN in increasing order of their f values Thus, the next node to be expanded will be the one with the lowest f value and if two paths lead to the same node then the one with the higher f value is discarded. The important point to note is that the selection of node to be expanded takes account of all the nodes encountered so far, and is not dependent on node location in any partial developed tree.

As it stands, the description of BF is merely an outline as no details have been given on how to compute f, what information is required, and how values are propagated through the graph. These are important points as they represent a significant proportion of the search effort and have been the focus of considerable research activity. By restricting the nature of the evaluation function four important variations of the BF algorithm can be identified, namely BF*, Z, Z* and A*. The restrictions involve -

- defining a recursive weight function W_G for a given solution graph G to represent either merit(Q) or cost(C).
- The concept of an optimal solution graph G^* of merit $Q^*(s)$ maximised over all solution graphs rooted at s.
- The use of a "backed-up evaluation function", e(n), which provides an estimate of $Q^*(n)$ for all nodes of the explored graph.

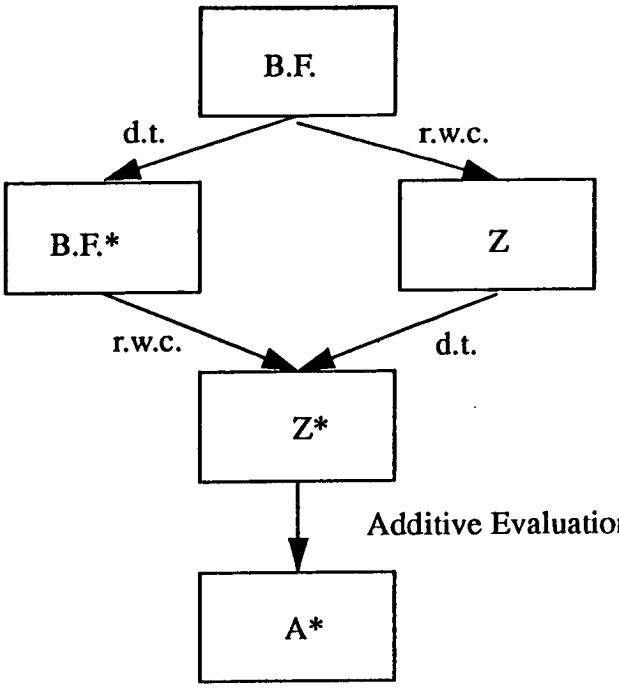
(These items are expanded in Appendix A, details can be found in standard texts such as Pearl(1984), Nilsson(1982)).

In the case of BF search on an OR graph, each node in OPEN is associated with only one solution base given by the path P(n) from the root node s to n. To decide which node to expand the function value $e_{P(n)}(s)$ is computed bottom-up along the path s to n and assigned to each node on OPEN. The evaluation function f(n) is simply

$$f(n) = -e_{P(n)}(s)$$

since f represents costs and is to be minimised whereas e represents quality and is to be maximised. The use of a recursive weight function enables e to be calculated locally using information stored at the parent node. Figure 2.8 shows the relationships between the algorithms.

Figure 2.8: Relationships between OR Graph Algorithms



(d.t. : delayed termination,
r.w.c. : recursive weight computation)

Additive Evaluation : $f(n') = g(n')+h(n')$

$g(n') = g(n) + c(n,n')$

The most general BF search corresponds to GRAPHSEARCH with an arbitrary heuristic evaluation routine f. If f is computed by the "rollback" procedure just described, the search is called the Z algorithm. The BF* algorithm which always finds an optimal solution if one exists, introduces the notion of "delayed-termination" of evaluation of the goal node. This requires a slight modification of the GRAPHSEARCH strategy. Rather than terminate immediately on reaching a goal node, step 5, the algorithm has to loop again to see if any other nodes on OPEN indicate a more promising solution base. Combining both techniques results in the Z* algorithm. In order to guarantee optimality all the cost estimates must be optimistic, i.e. under-estimating costs and over-estimating merit. (For proofs of the formal properties of these methods see Dechter and Pearl(1983), Nilsson(1980)).

The most well known algorithm, however, is the A* algorithm, Hart, Nilsson and Raphael(1968). The evaluation function f(n) at any node n is simply an estimate of the sum of the cost of the minimal cost path from the start node s to node n plus the cost of a minimal cost path from node n to a goal node, i.e. it is an estimate of the cost of the minimal cost path passing through node n. If $k(n,t_i)$ is the actual cost of a minimal cost path between node n and a goal node t_i, then let $h^*(n)$ be the minimum of all the $k(n,t_I)$ over the set of goal nodes. $h^*(n)$ therefore is the cost of the minimal cost path from n to a goal node. The complementary function, $g^*(n)$, gives the cost of the minimal cost path from start node s to node n, i.e.

$$g^*(n) = k(s,n), \text{ for all n accessible from s.}$$

The actual cost of an optimal path constrained to pass through node n is given by $f^*(n)$ such that

$$f^*(n) = g^*(n) + h^*(n)$$

and

$$f^*(s) = h^*(s)$$

is the actual cost of an unconstrained optimal path from s to a goal node. The heuristic evaluation function, f(n), is just an estimate of $f^*(n)$ given by

$$f(n) = -e_{P(n)}(s) = g(n) + h(n),$$

where g and h are estimates of g^* and h^* respectively. In particular, it can be shown that if h provides a lower bound on h^* (i.e. $h(n) \leq h^*(n)$ for all nodes n) the A^* algorithm terminates in an optimal path if such a path exists, i.e. A^* is said to be "admissible" (eg see Nilsson(1982)). Further restrictions on the nature of h can be used to improve the efficiency of A^*. One such restriction is the monotone restriction which requires that $h(ni) \leq h(n_j) + c(n_i,n_j)$ where node n_j is a successor of n_i and $c(n_i,n_j)$ is the arc cost between the nodes. Both Pearl(1984) and Nilsson(1982) prove a variety of useful theorems stemming from this restriction.

It is worth noting that the simpler uninformed searches are directly related to these more general search strategies. Breadth first search is a special case of A^* with h=O and g=d, the depth of a node in the search tree. Setting h to zero is certainly a lower bound on h^* but provides no information as regards controlling the search. Depth first searches can be obtained from Z^* by using f(s) = O and $f(n_j) = f(n)-1$ since f denotes cost which is to be minimised. The concept of dynamic programming (Bellman and Dreyfus(1962)) is just a reformulation of breadth-first search utilising a recursive graph weighting function which can be computed locally from parent node information. The study of branch-and-bound search techniques in operations research, Lawler and Wood(1966), exploits heuristic information in the form of real valued bounding functions to control the search.

Most of the algorithms which utilise restricted heuristic evaluation functions and delayed search termination conditions are concerned with guaranteeing the optimality of a solution. In a design environment encompassing complete project design and development, as discussed in chapter 1, there will be a wide range of problem content, varying from very high level "searches" such as flowsheet development down to very specific searches such as optimal tray design in a distillation column. Higher level design decisions will normally take account of a variety of factors which cannot be readily expressed in a quantitative measure, and indeed would be unnatural

or meaningless to do so. Typical examples of more qualitative aspects of design might include consideration of ease of plant maintenance and overall safety in plant layout problems, company policy on waste minimisation issues regarding the generation, treatment and disposal of effluent, investment decisions regarding the scale and flexibility of plant design etc.

In many design situations, therefore, decisions will be viewed as satisfying problems rather than strict optimisation problems. The use of sub-optimal but otherwise acceptable solutions cannot be avoided in many early stages of design when faced with imprecise or unavailable information. Pearl(1984) reviews the more well known empirical techniques used in situations in which the optimality requirement can be relaxed. A general form suggested by Pohl(1970) is to use a weighted evaluation function of the form

$$f_w(n) = (1-w)\, g(n) + w.h(n)$$

or

$$f_w(n) = g(n) + w.h(n)$$

In the latter form, it is obvious that a large positive value for w will over-emphasise the heuristic component of the evaluation function while very small values of w tend towards breadth-first search strategies.
In the first form, values for w of 0, 0.5 and 1 correspond to uniform cost, A^* and BF^* search strategies respectively. Pohl(1973) showed that search efficiency is often enhanced by dynamically changing the value of w so as to lessen the contribution of h the deeper the search. The evaluation function used was

$$f(n) = g(n) + hn + \varepsilon\left[1 - \frac{d(n)}{N}\right]h(n)$$

where d(n) is the depth of node n and N is the anticipated depth of the desired goal node. For small values of d, $d \ll N$ in the initial search steps, the search will be guided by the heuristic information. At deeper levels, however, the evaluation function contributions take on a more equal weighting. The search then resembles more closely A^* and helps protect against termination with local optimum solutions. The solution found will not be more than the optimal one by a factor of $1 + \varepsilon$
Another bounded version of A^* which is useful in problems if flat optimal solution regions is the FOCAL or A^* algorithm of Pearl(1982). FOCAL is a sublist of OPEN given by

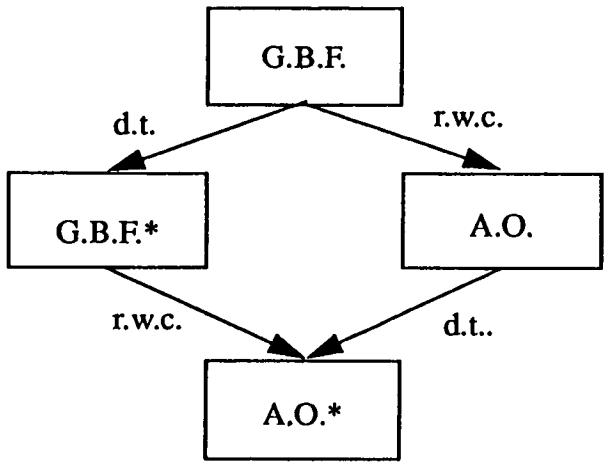
$$\text{FOCAL} = \{n : f(n) \leq (1+\varepsilon) \min f(n')$$

, n' ϵ OPEN that is those nodes not greater than the minimum f node by a factor of $1+\varepsilon$. The reason for this formulation is that experience shows that A^* often spends a disproportionate amount of time choosing between fairly equal solution candidates which prevents early termination of the search with a sub-optimal but acceptable solution. In the FOCAL algorithm, however, effort is spent in estimating the computational effort required to complete a search between nodes on FOCAL which have roughly equal solution quality. Thus, the node selection heuristic, which is concerned with remaining computational effort, is of an entirely different nature than the heuristic used to form FOCAL which deals with evaluation of solution "quality". This type of two-tier algorithm seems appropriate to many high level satisfying design problems that can be gradually decomposed into evaluation studies of well defined problems for which good estimates of the computational effort required in their solution can be provided.

2.2.4 Heuristic Search in AND/OR Graphs

The previous section described heuristic search techniques for state-space, OR graph formulations. A basic best first strategy was outlined and it was seen that many searching techniques were simply specialisations of this more general algorithm. This simple approach also applies to the development of search strategies for problem-reduction formulations, or AND/OR graphs. Figure 2.9 shows the relationship between four variations of the best first search method, the most well known of which is AO^*.

Figure 2.9: Relationships between AND/OR Graph Algorithms



(d.t. : delayed termination, r.w.c. : recursive weight computation)

The description of these algorithms is more involved and requires careful definition of what is meant by "best" and "candidate solutions". Full details can be found in standard texts, e.g. Pearl(1984). The important point to note is that simpler aspects of many process design techniques can be loosely described in terms of an AND/OR framework. This applies both in detailed situations, e.g. column internals design, and in higher level strategies such as the top down, hierarchical design approach of Douglas (1988). In those cases, heuristic variants of AO* will prove to be a useful starting point for the comparison and development of process design search techniques.

A class of algorithms that received much attention in early AI literature are the game playing procedures, the most well known being the MINIMAX and ALPHA-BETA methods. The basic idea is to represent alternative moves for two "players" in a "game" at alternating levels in a game graph or tree. The players are called MAX and MIN by convention, with nodes at an even depth representing MAX's positions, i.e. MAX to move next, those at an odd depth representing MIN'S positions. The arcs in the graph represent alternative moves in the same way as arcs represent actions or transformations in an AND/OR graph. The search strategy attempts to achieve a win,

or at least a draw, for player MAX. From MAX's point of view the successors of a MIN node are like AND nodes in that a win must be obtainable from all these positions. Successors of a MAX node, however, are like OR nodes in that a win is to be obtained from any of these positions.

In complex games such as chess the growth of the game tree suffers from combinatorial explosion which precludes use of exhaustive search techniques (Nilsson 1980) estimates that simpler game of checkers has approximately 10^{120} nodes in the game tree which would take about 10^{21} centuries to search even if each node's successor in the tree could be generated in one third of a nanosecond). This is exactly analogous to the situation that would arise if a naive representation and search technique were used to tackle an overall process design or synthesis problem. The search goal reduces, therefore, to simply find good next moves from a given position. Any of the previously described search techniques can be used where suitable to provide limited local search. The termination conditions will vary from problem to problem and may include constraints on time elapsed, storage space consumed etc. Once the local search terminates an estimate of the "best" move is made by use of a static evaluation function,v , to measure the quality of a leaf node n, Q(n).
The node values are then used in a rollback or backed-up evaluation function which propagates this information back to the start node, just as in AND/OR searches. A decision can then be made regarding MAX's next most likely move.

In the case of the MINIMAX method the quality of a node is simply

$$Q(n) = \begin{cases} v(n) & \text{if n is terminal} \\ \max [Q(n_i)] & \text{if } n_i \text{ represent nodes with MIN to move next} \\ \min [Q(n_i)] & \text{if } n_i \text{ represent nodes with MAX to move next} \end{cases}$$

(Convention scores MAX positively, MIN negatively, so MAX wants to maximise the backed up values of the successor MIN nodes, while MIN wants to minimise the backed up values of successor MAX nodes).
This technique relies on the assumption that the backed up value of a node is more reliable than that simply obtained by applying the static evaluation function to the node. The idea of basing decisions on information gained by limited "look-ahead" is one common to most game playing algorithms.

The ALPHA-BETA algorithm is a simple variation of MINIMAX which can significantly reduce the amount of search required by keeping lower and upper bounds, termed alpha and beta, for the maximising and minimising stages respectively. Full details can be found in Knuth and Moore(1975), Pearl(1984).

Further improvements in the performance of ALPHA-BETA can be obtained by carefully selecting the order in which nodes are expanded in partially developed trees. Well known examples are SSS* (Stockman(1979)) and SCOUT (Pearl(1980)). Roizen and Pearl(1983) present a detailed comparison of the ALPHA-BETA and SSS* methods.

2.2.5 Summary of Heuristic Graph Search Techniques

It is apparent that there are many similarities in the heuristic graph search techniques summarised above. Kumar and Kanal(1983) have formulated a generalised branch and bound algorithm for heuristic search of AND/OR graphs which clearly sets out the differences between algorithms such as AO*, SSS*, ALPHA-BETA etc. Martelli and Montanari(1975, 1978) also discuss more fully how dynamic programming problems can be recast in terms of an AND/OR search. The same strategies can also be applied in theorem proving techniques in the area of first order logic, as introduced in section 2.3. The generality of the state-space and problem reduction representations also means that most of the problem solving techniques developed recently in the AI field, as discussed below and in chapter three, can be described in a common framework. Much of the following discussion, therefore, assumes a good understanding of the basic search techniques introduced here.

The key to the successful use of these methods, lies in the power of the heuristic evaluation functions and their ability to effectively utilise background subject knowledge when making decisions. Struthers(1984) is an extensive review of heuristic graph search techniques for various process synthesis design problems. It is quite clear that most of this research, including more recent work by Grossmann and co-workers on the use of mixed integer non-linear programming techniques (e.g. Kocis and Grossmann(1988)), could be more easily formalised and described in graph theoretic terms as introduced here. In addition, such formulations would then be more easily compared to the use of heuristic evaluation methods from the field of A.I. As now described, many well known A.I. techniques can be described in the same general terms.

2.3 Classical Logics and Theorem Proving

The previous section outlined the basic algorithms used in heuristic graph search theory which have since been used as the basis for development of many AI search techniques. Similarly this section introduces the fundamental concepts of logic. A thorough understanding of these concepts is required to appreciate much of the subsequent discussion for the following reasons:

1. Logic based representations were the first to be used in AI and the majority of subsequent developments draw strongly on the theoretical aspects this early work.
2. A variety of representation schemes can be described in logical terms. This is useful for comparison purposes.
3. A precise definition of the meaning and use of terms such as "semantics" is possible. Chapter 3 discusses more complex theoretical issues central to any representation system to be used in a process design environment.

The mathematical study of logic has developed rapidly since the work of philosophers such as Frege, Wittgenstein and Russell in the late 1800's, and Quine, Tarski and Hilbert in the first half of this century. Of particular importance is the study of propositional and predicate calculus as basic representation mechanisms. Several introductory texts are available describing these formal languages in detail, eg Thayse(1988), Pospesel(1976), etc. Appendix B provides a brief summary of some of the terminology and concepts used in propositional and predicate calculus which are necessary for this introductory discussion. Later chapters assume some familiarity with the basic literature in this area.

Propositional, or sentential, calculus one of the simplest formal languages. It deals with propositional sentences that are either true or false. For example, "item h101 is an heat exchanger", "the composition of the reactor is within the explosive limits", "three plus four equals seven", and "if the mixture is not electrolytic then UNIFAC is a good estimation technique" are all propositional sentences. Truth values can be assigned to them unlike the terms "the reactor's temperature", "five times four" etc. which are not propositions. Appendix B outlines the syntax of well-formed formulas ("wffs") of propositional logic, and the use of the logical connectives "and", "or", "not". The notions of material implication and equivalence are also defined. Material implication is of importance as atomic propositions can be combined in an "if .... then ...." manner. The general formula

$$(H_1 \wedge H_2 \wedge ... \wedge H_n) \supset (C_1 \wedge C_2 ... \wedge C_n)$$

represents an arbitrary conjunction of hypotheses, or antecedents H, and conclusions or consequents C. This "rule based" format can obviously be used to represent design information of a heuristic nature and is discussed in more detail below. Also defined

in Appendix B is the notion of semantics with regard to a formal language, namely truth assignment. The use of the terms valid, contingent, satisfiable and consistent formulas is outlined along with the concept of logical consequence and use of tautologies.

It is not the purpose to discuss propositional logic to any depth here. The primary objective is to highlight the basic features of the language as a representation mechanism, and especially those features which have been used to develop more sophisticated representation schemes. Two considerations for any knowledge representation are:

1. "what can be said or expressed", ie the types of information that can be represented, and
2. the deductive, or inference, mechanisms that allow inferences to be made.

Well-formed formulas in propositional logic can represent arbitrary propositions and their combination using the five basic logical connectives $\wedge$, $\vee$, $\neg$, $\supset$, and $\equiv$. Universally valid formulas in a formal deduction system, see Appendix B, are usually referred to as axioms. A typical example of an axiom might be

$$P \supset (P \vee Q)$$

which can be used to guide inference mechanisms.
This means that any "english-like" sentence which can be represented as a proposition, such as the examples earlier, can be treated in propositional logic A wide variety of both general and process design specific information could be expressed in this simple formalism if desired. Some of the basic reasons why propositional logic is not directly used in AI systems are now discussed. The issues involved are important as many of them have resulted in the development of more adequate representations, including more complex logics, as discussed later in this chapter.

Consider the sentence "r101 is a reactor" and the design temperature of r101 is 100 degrees celsius". This could be expressed as the wff $p \wedge q$, where p and q are the propositions "r101 is a reactor" and "the design temperature of r101 is 100 degrees celsius" respectively. The essential restriction imposed by propositional logic is that the propositions p and q are "atomic" in nature, i.e. they must be treated as a unit whole and no reference can be made in isolation to the constituent parts of the propositions, namely the reactor, r101, or its design temperature. It is clear from the requirements and types of knowledge representation required for process design, as discussed earlier in section 2.1, that it will often be necessary to refer to specific objects, events or situations, hypothesis and deduce relationships between them, and generalise these relationships and basic physical principles over classes of abstract

objects. Simple concepts such as the "stream's flowrate" cannot be expressed as a true or false proposition and cannot, therefore, be directly represented in propositional logic.

Despite the inadequacy of propositional logic as a direct representation language, the theorem proving mechanisms developed to show the consistency of a set of propositions, or that one proposition logically follows from a known set of propositions, have considerably influenced most inference techniques used in A.I. In addition, the concepts of maintaining consistency of data and knowledge, defined earlier, are likely to involve similar or related mechanisms in a process design environment. The principal techniques are now described briefly to provide comparison later with reasoning mechanisms in alternative knowledge representation schemes.

Theorem proving involving a formal language, such as propositional logic, is performed within the framework of a formal deductive, or axiomatic, system (see Appendix B). This is defined to consist of a formal language, a set of logical axioms and a set of inference rules by which new formulas can be derived from existing wffs. Important inference rules include:

conjunction - from P and Q, infer $P \wedge Q$

simplification - from $P \wedge Q$, infer P

modus ponens - from $P \rightarrow Q$ and P, infer Q

hypothetical syllogism - from $P \rightarrow Q$ and $Q \rightarrow R$, infer $P \rightarrow R$

Modus ponens is one of the most important inference rules as it allows conclusions to be drawn directly from a given set of wffs representing a particular situation. As such it is central to theorem proving in logic and in most other A.I. representation schemes, either implicitly or explicitly. It should be well noted, however, that the choice of axioms and inference rules in any particular deduction system is completely arbitrary. Appendix B briefly outlines two well known but alternative axiomatisations of propositional logic due to Lukasiewicz(1929) and Hilbert and Ackermann(1928), both detailed in Frost(1986). The wffs derived from inference rules are called theorems, a proof of a theorem simply being the sequence of inference rules used in the derivation. Theorem proving, viewed as the selection of inference rules to transform a given set of wffs into a final desired wff, can be compared to the selection

of operators to transform intermediate states into a desired goal state in state-space heuristic graph search as discussed earlier (section 2.2).

Three basic approaches can be identified in theorem proving: purely syntactic theorem proving, purely semantic, and refutation procedures.

A purely syntactic approach to deriving a proof for a theorem is possible in proposional logic since it is concerned solely with the form, or syntax, of formulas and the assignment of truth by syntactic manipulation of formulas. Given a theory T, a syntactic proof of whether a formula F1 is a theorem of T can be found by repeatedly applying the inference rules of the system until the desired wff is obtained. Similarly, the consistency of a theory T, such as facts in a database, could possibly be determined by generalising all the theorems of T to check whether any wff F1 and its negation ¬F1 are derived. If they are both derived then T is inconsistent. This, however, is only possible in "complete axiomatic" systems (see Appendix B). The problem of deciding which inference rules to apply, in what order, along with example techniques is discussed later, (section 2.3.5), following the introduction of first order logic.

A semantic approach to theorem proving can sometimes be used in sound and complete deduction systems. This involves building a truth table involving all satisfying truth assignments of a theory T, i.e. those assignments in which all the proper axioms of T are satisfied, and checking whether the formula F1 is true in each case. If so, then F1 is a theorem of T. Consistency of a theory T can also be shown by generating all truth assignments of T until an interpretation which satisfies T is found. If no such interpretation can be found then T is inconsistent. The word "semantic" is used as proof checking relies on establishing the truth values of wffs under given interpretations. This is in contrast to syntactic approaches which can proceed without evaluation of the truth values of intermediate derived wffs. It will be seen shortly that a "semantic" approach is not always possible in other logics.

The third common style of theorem proving involves the use of refutation procedures. Given a theory T and a potential theorem F1, the basic idea is to negate F1 to ¬F1, add ¬F1 to the proper axioms of T, to give T2 say, and test whether T2 is inconsistent. The basic algorithm is given in steps 1-4 below. The refutation method is seen to combine the use of syntactic procedures followed by semantic checks in determining a proof.

```
1       Add ¬F to the set, S, of proper axioms of
        theory T to obtain the set
        S1 of wffs : S1 = S ∪ ¬F
```

```
2       Apply an inference rule to derive a new wff
        which is added to S1.

3       Test S1 for unsatisfiability.

4       If S1 is unsatisfiable,
        i.e. it contains a wff and its negation, then
        stop, else repeat from step 2.
```

It the set S1 is unsatisfiable then so is the set $S \cup \neg F$ and thus, F must be a theorem of T. The formulation of the problem was given in more general terms in Appendix B where it is described as the "deduction principle". This is expressed as

$$\{H_1,\ldots H_n\} \models C \equiv \{H_1,\ldots,H_n,\neg C\} \models F$$

where H_i are hypotheses, C is a conclusion and F represents the truth value false. This expression states that to prove a conclusion C logically follows from a set of hypotheses is equivalent to proving that adding the negation of C to the hypotheses results in an inconsistent set of wffs.

The most widely used style of the refutation based approach is resolution refutation, developed by Robinson(1965). Theorem proving methods are usually classified as "uniform" or "non-uniform" in nature. Uniform proof methods require that the proper axioms of a theory be recast in a standard, uniform syntax. Non-uniform methods work with wffs in arbitrary form. Resolution refutation is a uniform proof method which requires wffs to be expressed in a regular form called "clausal form" (see Appendix B).

Any wff in prepositional logic can be expressed in conjunctive normal form and, hence, in clausal form (Details of the process can be found in standard texts, eg Frost(1986), Nilsson(1982) Appendix B outlines the process in the case of first order logic). Resolution is an inference rule that can be applied to wffs that are clauses and can be written as

$$\{\neg X \supset A, \neg X \supset B\} \models A \vee B \text{ or,}$$

$$\{A \vee X, B \vee \neg X\} \models A \vee B\}$$

where X is a single proposition and A and B are clauses. The rule simply states that if a literal and its negation occur in a clause set then the literal can be eliminated without affecting the validity of the derived wff. More formally, if s_1, and s_2 are member clauses of the normal form S, and l is a literal such that $l \in s_1$ and $\neg l \in s_2$, then the clause

$$r = (s_1 \backslash \{l\}) \cup (s_2 \backslash \{\neg l\}), \backslash \text{denoting set difference,}$$

is a logical consequence of S. The derived clause r is called the resolvent of the two

parent clauses s_1, and s_2.

It can be shown, eg Chang and Lee(1973), that if resolution is repeatedly applied to a finite set of inconsistent clauses, S, then the empty clause, NIL, is necessarily contained in S. This is expressed in the resolution algorithm in Figure 2.10 which can be used to check the inconsistency of a wff. This can be used, therefore, to prove a conclusion given a set of hypotheses as stated above in the deduction principle.

Figure 2.10: Resolution Refutation on a set S of clauses

```
1)      Assign CLAUSE_SET = S

2)      While NIL is not a member of CLAUSE_SET
        begin loop

        3)   select two distinct clauses  s_i, s_j in CLAUSE_SET
             such that literal l ∈ s_i and ¬ l ∈ s_j
             ELSE exit loop

        4)   compute the resolvent r_ij of s_i and s_j

        5)   Assign CLAUSE_SET = CLAUSE_SET ∪ r_ij
             end loop

6)      If NIL is a member of CLAUSE_SET then S is inconsistent.
```

The choice of parent clauses s_i and s_j, step 3), dominates the efficiency of the resolution procedure. Consider the following axioms:

$$1.\ p \vee q \quad 2.\ p \vee r$$

$$3.\ \neg q \vee \neg r \quad 4.\ \neg p$$

Figure 2.11 traces two possible paths that could be chosen by a resolution approach. The numbers in brackets refer to the parent clauses s_i and s_j used in each resolution step.

Figure 2.11: Possible Resolution Paths

5. q	(1,4)
6. r	(2,4)
7. ¬q	(3,6)
8. NIL	(5,7)

5. p∨¬r	(1,3)
6. q	(1,4)
7. p∨¬q	(2,3)
8. r	(2,4)
9. p	(2,5)
10. ¬r	(3,6)
11. ¬q	(3,8)
12. ¬r	(4,5)
13. ¬q	(4,7)

The left hand column uses as few resolvents as possible. The right-hand proof uses a blind, recursive "left to right" strategy to select parent clauses. The latter strategy is obviously non-optimal, some resolvents being computed needlessly or more than once. Some basis techniques for improving the control strategy of refutation methods are briefly discussed in section 2.3.5.

Two further important styles of theorem proving are worth mentioning, namely natural deduction proofs and top-down proofs. These styles are merely outlined in the context of propositional logic in order to simplify the discussion.

Appendix B introduces the notions of "Hilbert-like" proofs and "Gentzen-like" proofs. Hilbert-like proofs involve the use of a set of axioms and a single inference rule such as modus ponens or resolution. Gentzen-like proofs are used in natural deduction systems. In such a system the set of axioms is empty and the inference rules used are classified as either "introduction" or "elmination" rules. The terms introduction and elimination refer to the use of the logical connectives, depending on whether they are introduced or eliminated in an inference rule. Natural deduction proofs are interesting as they are intended to mimic the way in which mathematicians often construct proofs. This involves constructing sub-proofs in a recursive manner and requires the use of suppositions which act as temporary local axioms assumed within a particular subproof. The control strategy and choice of inference rule relies heavily on the expertise of the person, or heuristics used, involved in constructing the proof. The technique can, however, be automated to a certain extent and does not require proper axioms to be expressed in a specific form. The latter point is important in that people often express axioms or rules in a way that indicates a certain "directionality" or structure which could otherwise be lost,

e.g. the pair of axioms $\neg P \rightarrow [Q \vee R]$, $M \rightarrow P$
seems intuitively different from the equivalent forms $P \vee Q \vee R$, $\neg M \vee P$.
Spoken implications often indicate a left to right flow of information which can be used to guide the construction of a proof. This hidden information can be lost when axioms are re-expressed in an alternative form.

The syntactic theorem proving techniques described so far all proceed in a "bottom-up" fashion, ie given a set of logical axioms or proper axioms or suppositions, inference rules are repeatedly applied until a proof is obtained. This method of proof is alternatively described as data-driven or forward- chaining or

deductive in nature.

An alternative approach is to use a "top-down", "backward-chaining", goal-driven or reductive proof method. As the name implies, the method involves starting with the required formula to be proved, along with the proper axioms of theory, and repeatedly apply inference rules in a "recursive sub-proof" style in order to determine whether or not the formula is a theorem. One such technique is the tableau-proof method which is briefly described in Appendix B. This method can be easily and efficiently implemented and is widely used as a top-down proof approach. It has also been extended for use in more complicated logics, see Section 2.3. Zeman(1973) gives a detailed description of the use of tableaux proof techniques in theorem proving and validity checking.

Forward and backward chaining techniques are described in more detail in section 2.4

2.3.1 First Order Predicate Calculus

It was argued at the beginning of this chapter what the ability to model a wide class of "objects" and their inter-relation in a suitable, high level representation language would be required in a process design environment. Objects will range in nature from representations of plant items to abstruct concepts describing the design activity itself. Relationships will be required to express design constraints as well as higher order information such as functional expertise, event and situation descriptions.

The previous section outlined the use of propositional logic. It was seen that propositional logic as a representation scheme is far from adequate in that it can only express true or false propositions. It is useful, however, in that any formal axiomatic system can be described in similar terms. In particular the notions of normal and clausal form for expressing logical formulas were introduced along with the three variations on theorem proving; namely purely syntactic theorem proving, purely semantic (truth tables) and refutation based procedures. The difference between data-driven, forward-chaining techniques and top-down, goal driven proofs was described. It was seen that there is an arbitrary choice as to the choice of universally valid axioms and inference rules, resulting in either Hilbert or Gentzen like proofs.

Many of these topics will be discussed again in various guises throughout the rest of this chapter.

The purpose of this section is to introduce first order predicate calculus. Predicate calculus can be used to describe objects, state relationships between objects and generalise those relations which hold over similar classes of objects. The concepts

involved in the extension of propositional calculus to predicate calculus are described in Appendix B. In particular, the use of variables, predicate constants, functional constants and the existential and universal quantifiers is introduced. First order logic is a simple yet extremely expressive representation mechanism. It is widely used in A I research as an analysis tool to understand and compare different knowledge representation techniques, e.g. see Levesque and Brachman(1985). It is used for comparison purposes throughout the rest of this work and familiarity with this subject is assumed. Excellent introductions can be found in Kowalski(1979), Thayse(1988) or Frost(1986). Much of the terminology used is given in Appendix B.

As a brief introduction, examples of some of the aspects of knowledge representation needed in a design environment, as described in section 2.1, are now given.

One-place predicates, i.e. predicates with one argument, can be used to classify individuals into sets, sorts or types. For example, the predicates pump(x), reactor(x) etc. specify certain types. Any individual constraint, eg. P101, satisfying such a predicate could be considered to be of that type, eg pump(P101) has truth value TRUE. Some sorts can include other sorts such as pump(x) and centrifugal_pump(x), while others form disjoint sets.

Predicates can be used to express a variety of statements about individuals, either about themselves or their relations to other individuals. For example, "temperature(r101, 298)" states the attribute value for reactor r101 to be 298. "upstream-of(r101, p102)" could express the relation that p102 is upstream of r101. Relations and attribute values can be referred to specific situations, eg. design case studies, by use of an extra argument to name the situation.

The predicate "temperature(r101, 298, design_study1)" refers to the temperature of r101 in situation "design_study1". Similarly, extra arguments can be used to refer to specific times or events.

Complex descriptions called, well formed formulas (wff's), can be constructed using the logical connectives {and, or, not} or $\{\wedge,\vee,\neg\}$, e.g. pump(X) $\wedge$ operating_pressure(X,P) $\wedge$ greater_than(P,10).
The existential and universal quantifiers permit general statements about classes of individuals to be made. These are typically used in conjunction with the material implication connective ($\rightarrow$) to express inference rules. For example,

$$\forall X,S.\ (\text{alkane}(X) \wedge \text{cubic_equation_of_state}(S)) \rightarrow \text{model}(X,S)$$

might be a simple inference rule in a physical property modelling system. Many

other inference rules could be formulated and applied, as required, by theorem proving techniques similar to those described earlier for propositional logic.

Functional constants are introduced for notational convenience in first order logic. Functions, unlike predicates, return values other than true or false when applied to their arguments,
eg. upstream of (r101), could return the unit upstream or reactor r101. Functions can be composed, as in upstream_of(upstream_of(r101)), and can be used as arguments in predicates,

connected_to(x, upstream_of(x)).

Functions do not extend the range of what can be expressed, merely how it is represented.

Finally, a logic is said to be first order if it permits quantification over individual constants but not over predicate or functional constants. This means reasoning cannot be performed on the predicate constants themselves. This is unlikely to be a major restriction in many simple process engineering applications.

It is worth considering how information described by sets of well formed formulas is related to the representation of data in traditional process design databases. If wffs are viewed as relational structures then the structures can be expressed as quadruples, U, where

{U = (E, N, R, Fn) , and}
- E is a set of entities $\{e_1,e_2,..\}$ representing the domain of U
- N is a set of distinct individuals $\{n_1,n_2,..\}$ and is a subset of E
- R is a set of relations $\{r_1,r_2,..\}$ defined on E
- Fr is a set of functions $\{f_1,f_2,..\}$ defined on E

eg. U given by -

E = set of plant items in benzene hydrogenation plant
N = {he101, p201, r301, stream1, stream2, ...}
R = {upstream_of, controls, has_design_pressure,...}
Fn = {constraint_function1,...}

where,

upstream_of = {(p101, he101), (p102, r301)., ...} etc. The relations and functions can be of any arity, the "constraint-functions" being any design constraints, algebraic expressions etc.

There is an obvious mapping, therefore, between any of the database types, ie.

hierarchical, network or relational, and a logic based representation. Since relational tuples can be expressed directly as sets of wff's, there would be little difficulty representing the data in conventional process design databases, such as PRODABAS (see section 1.4.5), in a logical framework. This is an important point. The use of the logical quantifiers and connectives in sets of wff's means, in fact, that the variety and types of statements that can be made in a first order logic system is much more extensive than is conveniently possible in conventional database systems. Complex relationships are more easily defined and domain dependant design heuristics, or inference rules, can be handled in the same representation system as the design "data". It is also straight forward to represent aspects of the design process itself. For example, individual constants can refer to design team members just as well as plant items, situations and events in the design can be handled directly by naming individual scenarios, e.g. identify each case study, name data release updates etc. Check on the use of update and self-consistent data can be enforced by defining explicit constraints, or relationships, describing the use of data in different scenarios.

For example,

$$\text{can_use_data}(X,S1) \wedge \text{derived_in}(X,S0) \wedge \text{precedes}(S0,S1)$$
$$\text{authorised_release}(S0)$$

might express the constraint that data X can be used in the current situation S1 if X was derived and authorised in an earlier situation S0. X could be replaced by a specific functional constant, say temperature(r101), and the constraint checked in an application program or by the engineer.

2.3.2 Reasoning Techniques In First Order Logic

It is clear that first order logic can be used to represent a wide variety of process engineering design situations. Specific problem facts can be represented by a set of wffs. General heuristic information and guidance can be expressed using quantifiers and inference rules. The choice of actual predicates and functions used to represent the problem is at the discretion of the program designer. In order to derive or deduce new facts from those given, or answer specific design queries, then some reasoning mechanism must be provided.

This section introduces important two techniques used in theorem proving, namely the use of unification (used in pattern matching or variable substitutions) and the use of heuristics to control and guide the theorem proving process. The theorem proving techniques used in first order logic are direct extensions of those described earlier for propositional logic.

2.3.3 Unification And Pattern Matching

An extremely important method used in symbolic computation languages is that of "matching" one symbolic data structure with another. In the case of first order logic this is equivalent to finding substitutions of terms for variables that makes well formed formulas identical. This substitution process is called unifacation. For example, the query "temperature(r101, x)" can be matched with the fact "temperature(Y, 298)" by the substitution, s, where s = {r101/Y, 298/x}.

The concepts of substitution, composition of substitutions and most general unifiers (mgu's) are described in some detail in Appendix B. A recursive algorithm, called UNIFY, is also presented which can be used to find the most general unifier of a set of unifiable expressions $\{E_1, E_2,.. E_n\}$, see Nilsson(1982). The algorithm is of linear complexity in the size of the terms to be unified. The algorithm includes an "occurs-check" which is used to trap the case in which a variable, x say, tries to match against a term which contains that same variable, x, but which does not reduce to x. This check is often omitted for efficiency reasons in actual implementations.

The algorithm can be used, or modified, to perform most pattern matching tasks. It assumes, however, that the formula be expressed in list notation of the form (Predicate Arg1 Arg2 ... ArgN), e.g. pressure(p101, x) becomes (pressure p101 x).

Similarly, an "if ..... then ....." type heuristic can be expressed as a Horn Clause, see Appendix B, if there is a single consequent C, i.e a single positive literal, and any number of antecedents $A_1 \ldots A_N$.
e.g. if $A_1 \wedge A_2 \wedge .. \wedge A_N$ then C, becomes the set

$\{\neg A_1, \neg A_2, ..., \neg A_N, C\}$. Any part of the rule can then be matched as required. For example, a typical heuristic used in separation sequence synthesis is "don't separate close boiling components first". This might be written as

```
if fluid(A) ∧ fluid(B) ∧ boiling_point(A,Ta)
∧ boiling_point(B,Tb) ∧ less_than(10,abs(difference(Ta,Tb)))
then separate(A,B,priority(low)).
```

The antecedents or consequent can be matched depending on whether or forward or backward chaining theorem prover is being used.

2.3.4 Theorem Proving Techniques In First Order Logic

This section outlines the principle theorem proving, or inference, techniques currently used in first order logic systems. The remainder of this thesis assumes a good appreciation of the methods involved, and their limitations, since they are the only means of deriving new facts or checking the consistency of sets of facts in a logic

based representation system. As such, they would form an integral party of any knowledge based process design system which attempted to support the concepts of integrity of data and knowledge, see chapter 1. In addition, it will be shown later that most of the representation styles used in knowledge based systems can be re-expressed in a logical framework. This is important as the advantages or limitations of any one style can then be compared against any other. Of particular importance to the design of a process engineering knowledge based system is the types of deduction that can be performed and their efficient implementation.

Historically, resolution refutation based theorem proving has been the most widely used inference technique. It's implementation and problems are described below. More recently, development has concentrated on the use of non-clausal and natural deduction systems. This style of proof technique is not yet fully developed but is well worth mentioning.

The subject of automated theorem proving in first order predicate logic is a large one. Detailed discussions cannot be given here, for reasons of space, but references to important papers are provided.

Section 2.3 introduced the main styles of theorem proving used in formal deduction systems in the context of propositional logic. In particular, the resolution inference rule was defined and an algorithm presented to perform the resolution refutation technique, (see Figure 2.10). This technique is a simple application of the "deduction principle". This was expressed by the statement,

$$\{H_1, ..., H_n\} \models C \equiv \{H_1, ..., H_n, \neg C\} \models F,$$

which states what proving a conclusion C logically follows from a set of hypotheses is equivalent to showing that addition of the negation of C to the original set of hypotheses results in an inconsistent set of wff's.

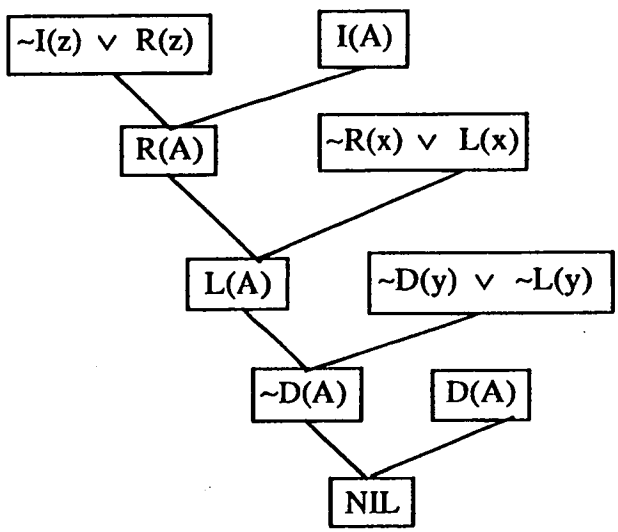
The purpose of this section is to indicate in more detail the practical problems of applying the resolution refutation technique to realistic problems. The problems encountered are also symptomatic of many other styles of theorem proving, as summarised at the end of this section.

Resolution refutation is a uniform proof method which requires that the proper set of axioms be translated to a standard form, namely clausal form. This transformation is described in Appendix B. It simply involves the elimination of universal and existential quantifiers, the latter involving the replacement of existentially quantified variables by Skolem functions whose arguments are universally quantified variables. The variables chosen as arguments are those whose scope includes the scope of the

existential quantifier being eliminated. The Skolem function represents the function that would return the individual that "existed". Once in prenex form the formula can then be expressed in clausal form. ie. a finite set of wffs each of which is a disjunction of literals. The transformation process is used to simplify the implementation of theorem provers and is a simple mechanical process involving repeated use of a set of rewrite rules. Nilsson(1982) provides an extended worked example.

Once a set of axioms have been transformed to clausal form they can be used as the clause, S, in a refutation system, Figure 2.10. For present purposes, it is useful to describe the refutation method in terms of a resolution refutation tree. This structure is simply derived from the "derivation graph". The nodes in a derivation graph are labelled by clauses, with nodes existing initially for every clause in the base set of wff's. New nodes are created when a resolvent r_{ij} is produced from two parent clauses c_i and c_j. Links are created between the new node and both its parents nodes. The resolution refutation tree is simply that part of the graph involved in the resolution process. Figure 2.12 shows a simple refutation tree with the root node labelled by NIL, the empty clause.

Figure 2.12: A Simple Refutation Tree



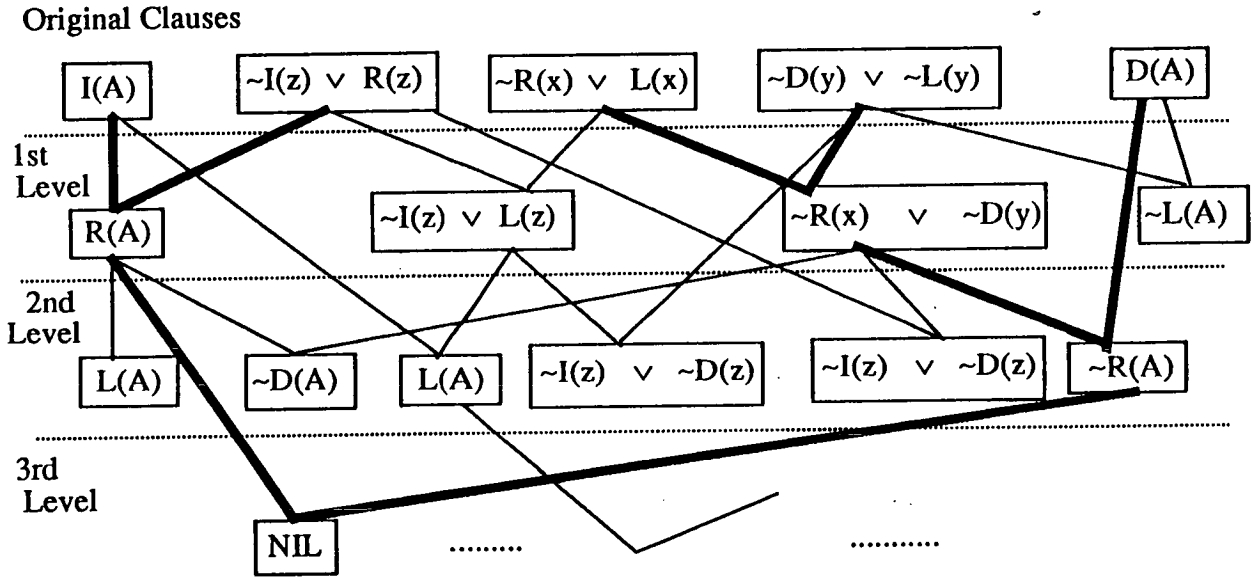
Refutation trees are useful in that they allow the inference technique to be described in standard graph theoretic terms. Implementation of a good control strategy is essential to limit the rate growth of the refutation tree and avoid unnecessary

resolvents being generated. A refutation control strategy is said to be complete if it will produce a contradiction whenever one exists (the use of "complete" here is not the same as meant by logical completeness, see Appendix B).

The choice of control strategy, as in general heuristic graph search procedures or specialised branch and bound optimisation methods, is critical to the efficiency and practical use of any inference technique. Some well known domain independent heuristics used in refutation system and associated control strategies are now presented.

The simplest control strategy to adopt is breadth first resolution. All first level resolvents are computed first, then second level and so on. An i-th level resolvent can have parent clauses at any previous depth but one parent clause must be at level i. As in heuristic graph search, the breadth first approach will guarantee to find the optimal "solution path length" if a solution exists at all. It is a complete strategy but too inefficient for practical use due to the combinational explosion, even in small problems. Part of a typical derivation graph is shown in Figure 2.13

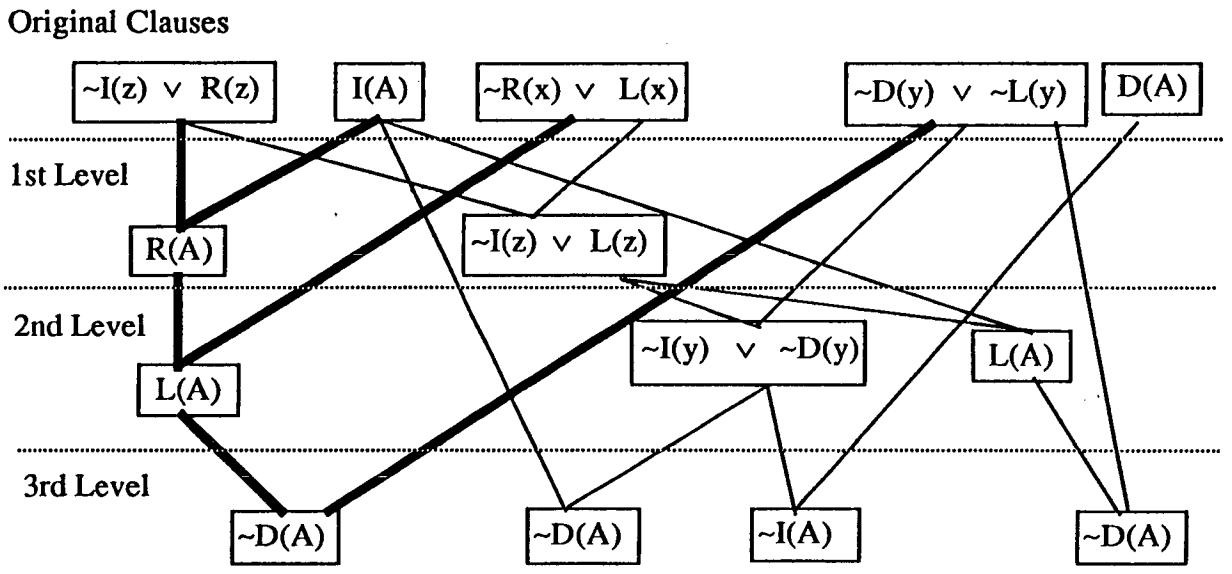
Figure 2.13: Breadth First Derivation Graph



Two simple, domain independent restrictions can be used to limit the rate of growth of the derivation graph. The first requires that in each resolution step at least one

parent clause is chosen either from the clause set resulting from the negation of the goal wff or from their previously derived descendants. The derived descendants are called the set of support, as is the overall strategy. Set of support refutation can be made to be complete by guaranteeing that it search for all possible set of support refutation in a breadth-first manner. Figure 2.14 show part of a derivation graph.

Figure 2.14: A Derivation Graph Using A Set of Support Strategy



The second restriction involves modifying the set of support strategy to try to always select a parent clause involving a single-literal clause, or "unit". The strategy is called unit preference resolution and helps the search to reduce the number of literals available at each resolution step.

Book the unit-preference and set-of-support strategies will normally increase the depth at which the empty clause is found but will significantly reduce the rate of clause set growth. Set of support is an interesting strategy in that it follows a backword-chaining, goal driven solution path, i.e. by using a clause originating from the goal wff or its descendants, but permits forward reasoning steps in the same system. The forward steps correspond to resolutions which involve a parent clause which does not descend from the negation of the goal wff.

One of the simplest refutation control strategies is called linear-input resolution. In

this case any literal can be used for unification provided that the most recently derived resolvent is one of the parent clauses and the other is chosen from the input clause set. The term linear is used because as the most recent resolvent is always on of the parents of the next resolution, the refutation steps can be described by a linear path. Linear-input resolution however, is only complete for Horn Clauses.

A widely used variation of the linear-input strategy is LUSH resolution, Hill(1974). (The acronym LUSH comes from "Linear Resolution with Unrestricted Selection function for Horn Clauses"). The added restriction here is that the literal to be selected for resolution be chosen from the most recently derived resolvent in a pre-defined order. The order is arbitrary, but the left most literal which has a complement in the input clause set is usually adopted. The technique is only complete for Horn clauses but is of added interest as it forms part of the theoretical basis of the programming language Prolog, see section 2.3.6.

In order to provide a complete strategy for non-Horn clauses, the input restriction must be relaxed but the linear constraint can be kept. The resulting strategy is called SL resolution, selected literal resolution, see Kowalski and Kuehner(1971). The literal selected must still come from the most recently derived resolvent, the linear constraint, but ancestor resolution is allowed. Ancestor resolution is simply where one clause is resolved with one of its ancestor resolvents in the derivation graph, the ancestor resolvent not being present in the original set of input clauses.

2.3.4.1 Constructing Solutions In Resolution Refutation Systems

Many problems in process design can be viewed as a process of satisfying or minimising constraints, eg. design of a reactor which achieves some intended aim. In terms of theorem proving this is a variation on the theme of proving a wff such as $(\exists x)W(x)$ where $W(x)$ is some complex wff describing, perhaps, the properties of x. It is not sufficient to merely prove that x exists, as would be derived by the strategies described earlier. Rather, a constructive proof is required which will produce the satisfying instance of x.

Fortunately this is straightforward to achieve following the method of Luckham and Nilsson(1971).

1. Construct a resolution refutation tree as usual. Label or mark the unification sub-sets of the clauses used in each resolution.
2. Replace any Skolem functions occurring in the clauses that result from the negation of the goal wff by new variables (see explanation below).
3. Convert the clauses that resulted from the negation of the goal wff into tautologies by appending to them their own negations.
4. A modified proof tree results which models the structure of the original refutation tree. The unification sets used are determined directly from those in

the refutation tree and will be the same unless Skolem functions have been renamed (step 2).

5. The clause at the root of the tree will now be the answer or solution to the goal wff.

Figure 2.15 shows a trivial example of a refutation tree and its conversion to a modified proof tree. The justification for the method is simple. By converting every clause resulting from the negation of the goal wff into a tautology, then the modified proof tree is, in fact, a resolution proof that the clause at the root of the tree logically follows from the axioms used plus the tautologies. Since tautologies are universally valid, the clause must also logically follow from the axiom set alone. Thus, the unifaction sets used in the proof tree, and the parent clauses used in each resolution, "justify" or "explain" the construction of the solution statement. The proof tree essentially describes "how" or "why" a particular solution was arrived at.

Step 2 in the answer extraction process requires some explanation, i.e. replacing Skolem functions by new variables. If a goal wff contains a universally quantified variable, eg. $(\exists x)(\forall y)(\exists z)$ [illegible] then it will become existentially quantified in the negation process and replaced by a Skolem function constant, P(x,g(x),z).

Luckham and Nilsson(1971) showed that these Skolem functions can be renamed as new variable without invalidating the proof. Any new variables merely trickle down to appear in the answer statement. In fact, the substitution sets will normally be more general as a result than those used in the original refutation tree.

Finally, it is obvious that the form of the answer statement depends on the refutation tree constructed. If there are several solution paths then some answers may be identical whilst others may be more general expressions in terms of variable occurrence. There is no way of knowing, without complete search enumeration, if the most general answer had been obtained. This is similar to the problem of ensuring a global optimum solution in heuristic graph searches except the problem is worsened by the undecidability of predicate calculus. It is not always possible to know whether all the possible proofs for a goal wff have been found. In practical applications, therefore, search will terminate when a satisfactory solution is found.

2.3.4.2 Simplifying The Search Strategy

The choice of control strategy is not the only means of reducing the amount of search. Problem descriptions, facts and heuristics expressed as sets of wff's can often be simplified by the elimination of certain literals or formulas. There are four simple mechanisms that can be used.

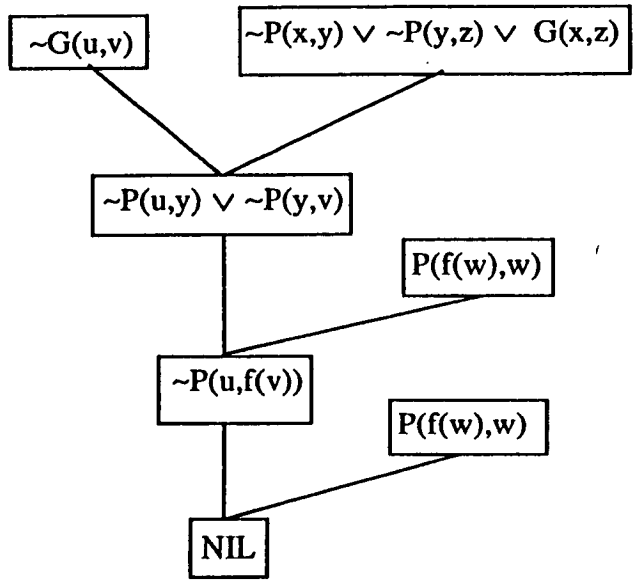
1. Elimination of universally valid formulas. Tautologies can be removed since an

Figure 2.15: Converting a Refutation Tree to a Modified Proof Tree

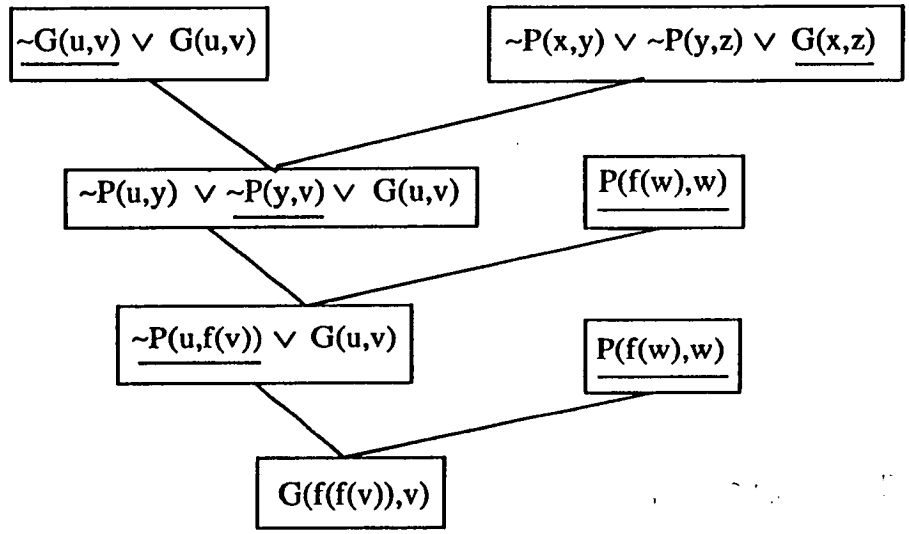
Given the Axioms : $\forall x\ \forall y\ \{[P(x,y) \wedge P(y,z)] \Rightarrow G(x,z)\}$
$\forall y\ \exists x\ P(x,y)$

Prove the goal : $\exists x\ \exists y\ G(x,z)$

Refutation Tree :



Modfied Proof Tree: Resolvents Underlined



unsatisfiable clause set will remain unsatisfiable after these formulas are removed.

2. A clause C can be deleted from an unsatisfiable set of clauses S if it contains some literal L that could not be resolved on by another clause in S. This is called the purity principle, see Robinson(1965).
3. A clause C can be deleted without affecting the unsatisfiability of a set of clauses S, if C is subsumed by another clause in S. A clause C is said to be subsumed by clause D if there is a substitution θ such that $C \leftrightarrow D\theta \vee E$
   e.g. y = x is subsumed by Y = x as anything that can be derived from the variable free term can also be derived from the more general term
   $(y=x \leftrightarrow (Y=x)\{y/Y\} \vee f)$

The fourth mechanism available is called procedural attachment. This technique is of great importance to the practical application of most knowledge based representation systems, including those based on logic, and is a topic that will be often be discussed in the rest of this chapter. In logic based systems, procedural attachment usually refers to the use of evaluable predicates to establish the truth value or literals rather than include the literal and their negations in the base clause set. The most obvious example is the use of the predicate constant "equals", eg. equals(4,3). It is much simpler to evaluate the truth value of equals(4,3) than litter the clause set with a large number of ground instances, i.e. variable free instances, of the form "equals(x,y)" and "$\neg$equals(x,y)".

The precise nature of procedural attachment is worth considering in more detail here. Well formed formulas in predicate calculus are simply linguistic constructs that denote truth values, individuals, functions or relations in a given domain. These formulas are useful in that they can be interpreted with reference to a model. This model associates the linguistic entities with the appropriate domain entities which are being modelled, and associates the truth values true or false with sentences constructed in the predicate calculus language. In practical applications, models and their interpretation process are not, however, finite. Infinite models, for example, include those which represent natural or real numbers. As a result, partial models must be used instead, procedural attachment being one feature of such a model. Consider a computer program, i.e. evaluable procedure called Eval_equals, say, which tests for the equality of two numbers within the finite domain that can be represented in the machine. The code Eval_equals is said to be attached to the predicate symbol equals. Similarly, in the wff equals(4,3), the linguistic entities 4 and 3 (ie. the numerals) are attached to the numbers, or data items, 4 and 3, and Eval_equals(4,3) is attached to the expression equals(4,3). A literal is said to be evaluated when it is interpreted in the partial model by running the attached procedure. The attached procedure will return a value which indicates whether a truth value of true or false is to be associated with the literal.

Procedures can also be attached to function symbols in exactly the same way as described for predicate symbols. Simple arithmetic functions such as plus(x,y), times(x,y) etc., could be evaluated in this way. Where necessary, or desirable, higher level functions such as boiling-point(x,pressure) could have attached procedures.

It is certainly not the desire, to attach procedures to every predicate or function symbol. The technique is used purely to help reduce the size of the proof search space. If a literal in a clause evaluates to true then the entire clause set can be eliminated without affecting the unsatisfiability of the remaining set. If it evaluates to false, however, then only the occurrence of that literal can be eliminated from the clause.

In summary, the essence of procedural attachment is that it is an implementation specific aspect of the more general interpretation process of a partial model. As such, it is just one of many forms of extended semantic checking that can be used in theorem proving to reduce the required search effort (see Bundy(1983, chapter 10) for an extended discussion on the use of semantic information to guide a proof search).

2.3.4.3 Paramodulation

The introduction of the "equals" predict above provides a convenient opportunty to discuss some of the problems arising from the use of such general predicates. Another typical example is the use of and connotations associated with, the "is-a" predicate, e.g. is-a(x,y). The problem lies not in logic but mainly in the direct, and varied usage of these terms or concepts in the English language. When two objects are said to be equal, or one object is another, the precise semantics intended often depends strongly on the context in which the statement was made, i.e. taking a statement to be true or false depends on the context and purpose of why the statement was made. For example, in process design problems it is usual to simplify a model of a unit operation, say, for specific reasons, data constraints etc. Sentences of the type "r101 is a plug flow reactor" can be used to represent this situation, but are known not be true in a wider context.

Even within a self-consistent context notions of equality etc. can prove troublesome. For example, in mathematics equality is represented by the = symbol. Five axioms are used in its definition.

1. Reflexivity : $X=X$, i.e. an object equals itself
2. Symmetry : $X=Y \rightarrow Y=X$, i.e parameter order is unimportant
3. Transitivity : $X=Y \wedge Y=Z \rightarrow X=Z$, i.e. equality is inherited.
4. Substitution : substitution of an n-ary function f
   $X_1=Y_1 \wedge .. \wedge X_n=Y_n \rightarrow f(X_1,..,X_n)=f(Y_1,..,Y_n)$
5. Substitution : substitution of an n-ary predicate p

$$X_1=Y_1 \wedge .. \wedge X_n=Y_n \rightarrow p(X_1,..,X_n)=p(Y_1,..,Y_n)$$

If these axioms are directly used, however, in a proof then the branching rate of the proof search tree will dramatically increase. The problem is simplified by introduction of a new inference rule of logic called paramodulation. It is defined as

$$c(X) \wedge X=Y \rightarrow c(Y)$$

where c(x) is a clause containing a term X and Y is some other term. If it happens that c contains more than one occurrence of X then only the first distinguished occurrence of X is replaced, multiple occurrences being replaced by repeated application of the rule.

This is indicated by square brackets c[X]. It is also usual to allow the substitution literal to be on either side of the equality, i.e.

$$c[X] \wedge ((X=Y \vee Z) \vee (Y=X \vee Z)) \rightarrow c[Y] \vee Z,$$

where Z can be the empty clause.

Finally, the occurrences of X are made unfiable rather than identical,

$$c[X'] \wedge ((X=Y \vee Z) \vee (Y=X \vee Z)) \rightarrow (c[Y] \vee Z)\theta\,,$$

where θ is the most general unifier of X' and X.

The paramodulation rule can simplify a proof. Its use is important in any sort of proof involving equations or constraints involving variables since these cannot be evaluated by an attached procedure. All the theorem prover can do is maintain consistent unification sets. The problem is further compounded by the semi-decidability of unification involving associative operators indicated earlier,
eg. x + (y+z) = (x+y) + Z or X + z = z + Y.

The main point to note is that attached procedures cannot always be used to evaluate certain predicates or functions. In those cases the paramodulation inference rule, or similar, can be used to help simplify the proof rather than revert to use of low level axioms resulting in rapid combinatorial growth. Bundy(1983) describes the practical problems of constructing proofs involving mathematical models, sets of equations, inequalities etc. These issues are of direct relevance to a process design environment which attempts to maintain consistency throughout a project design, plant models changing from simple to complex in nature as the design progresses.

2.3.4.4 Alternative Theorem Proving Methods

Before leaving this general introduction to theorem proving in first order logic it is useful to review more recent research since many of the techniques described so far date from the late 1960's.

A topic of much interest in the 1970's was the development of non-clausal resolution theorem provers, eg Wilkins(1974). These systems do not convert formulas to clausal form as this can destroy useful information implicit in the structure, or layout, of the original formula, e.g. there is often an intended "direction" or flow of information in formulas of the form "if A and B then C, as opposed to $\neg(A \wedge B) \vee C$. Murray(1982) is a good example of this approach using the non-clausal inference rule. To understand this inference rule consider again binary resolution, section 2.3.4., $\{(X \vee Y_1), (Z \vee \neg Y_2)\} \models (X \vee Z)\theta$, where θ is the most general unifier of Y_1 and Y_2. This is the same as replacing the matched positive and negative literals by "false" and "true" and forming the disjunction of the resultant clauses. The true or false literals can then be removed on truth grounds, i.e. they both evaluate to false in a disjunction. If the formulas A and B contain atomic formulas $L_1, ..., L_n$ such that there is a most general unifying substitution θ, $L_1\theta = L_2\theta = ... L_n\theta = L$, then if L is positive in A and negative in B the non-clausal resolvent of A and B is simply $A\theta\{L/false\} \vee B\theta\{L/true\}$.
Murray lists six simple rules to establish the positive/negative polarity of a wff and eight further "reduction" rules to simplify the resolvent clauses, eg. Let A, B and C be wffs, and C be a sub-formula of B. One polarity rule states that if C is positive/negative in A then C is positive/negative in the forms $A \vee B$, $A \wedge B$, and $B \rightarrow A$.

NC resolution still suffers from the problem of deciding upon a good control strategy but information may be available from the original structure of the formula set. In addition, NC resolution is generally more complex to implement efficiently although it should be less redundant than clausal resolution.

Another technique developed for simplifying proof procedures was the development of connection graphs. They are worth describing in one detail as variation of this method arise in many other knowledge representation formalisms.

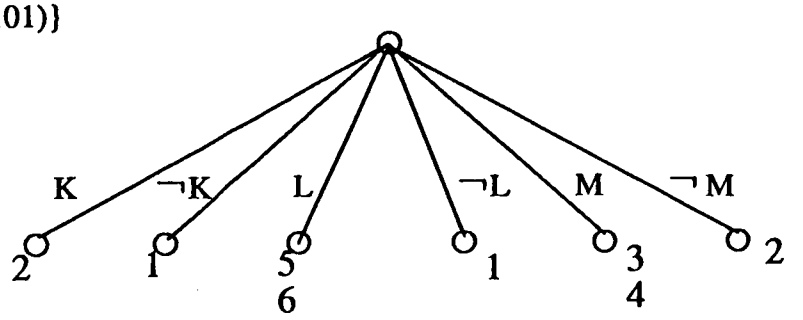
The simplest idea is that of "classification trees" Kowalski and Kuehner(1971). The tree is used to index clauses in the base set according to the literals which they could resolve on. The literals are named on the tree links and the clauses with which they could be resolved label the nodes. Figure 2.16 shows a simple example. The reason

behind this "pre-analysis" is to reduce the runtime effort of identifying potential resolvents. This can amount to significant cputime in large clause sets.

Figure 2.16: A Simple Connection Tree

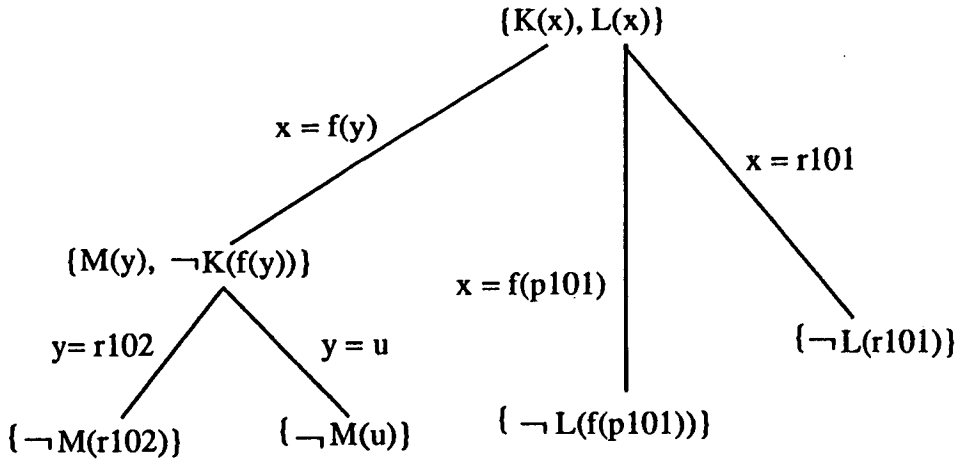
Given Clause Set :

{K(x), L(x)}
{M(y), ¬K(f(y))}
{¬M(r102)}
{¬M(u)}
{ ¬L(f(p101))}
{¬L(r101)}



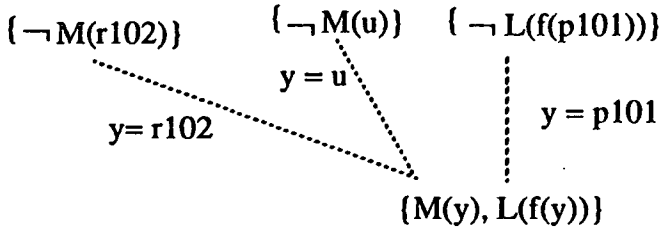
Boyer and Moor(1972) extended this idea to the notion of a connection graph. Each clause in the input set is used to label a node. More precisely, the idea is that every literal occurrence in the original set S of a theory be identified with a node. The arcs in the graph then indicate the possible resolutions and are labelled by the most general unifying substitution between the two nodes. Figure 2.17 shows the example of Figure 2.16 as a connection graph.

Figure 2.17: Connection Graph Representation



There is a unique resolvent associated with each arc which can be generated by resolving the clauses at either end of the arc. In the process, a new node will be created representing the resolvent. It in turn will have arcs between itself and the other nodes that were connected to the two parent clauses as long as the substitutions are compatible. Figure 2.18 shows the graph of Figure 2.17 after two of the nodes have been resolved. Detailed descriptions of a connection graph proof procedure can be found in Kowalski(1975, 1979).

Figure 2.18: Effect of Adding a Resolvent to a Connection Graph



There are several advantages in using such an approach. First, much of the computation effort can be reduced by careful pre-processing of the input set. Secondly alternative control strategies can be easily implemented or simulated. Finally the effort involved in deriving subsequent proofs or refinements of earlier proofs is much reduced in that new problems can continue from where previous

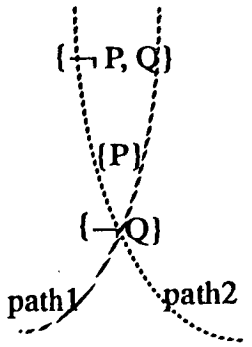
problems finished, assuming of course that the connection graph data structure is not descarded.

Stickel(1983) is a recent example of the development of a non-clausal connection graph theorem prover. In this case arcs connects nodes which have potentially unifiable literals of opposite polarity.

The last alternative technique considered here is the matrix connection method, see Bibel(1976,1983) Andrews(1981). The method is straight forward and can be applied to clausal or non-clausal forms. The basic idea is best described by a simple clausal-form example in propositional logic. To prove that Q is a theorem of the formulas $P \rightarrow Q$ and P,

1. Convert to a set of clauses S, $\{\neg P, Q\}$, $\{P\}$
2. Negate the goal formula to clausal form and add to S, $\{\neg Q\}$
3. Identify all "paths" through S such that each path passes through one literal in each clause, see Figure 2.19.
4. S is unsatisfiable if each path contains one or more complementary literals. In Figure 2.19 $\neg P$ and P occurs in path 1 and $\neg Q$ and Q in path 2. This S is unsatisfiable and Q is a theorem of the original clause set.

Figure 2.19: Paths in the Matrix Connection Method



The method essentially consists of generating all truth assignments for the clause set. If no satisfying truth assignment can be found, ie. complimentary literals exist in all possible assignments, then the clause set is unsatisfiable. A complimentary pair of literals in a path is called a connection. The spanning set of connections is one for which each path in the set of clauses has a connection named in the spanning set.

At first sight the method seems practically infeasible as it amounts to exhaustive numeration. In fact, it can be implemented quite efficiently by realising that a path P, and all other paths containing P as a sub-path, can be ignored as soon as a

complementary pair of literals is found in P. A brief outline of the approach is given in Appendix B. Full details can be found in Bibel(1982, 1983). The method is more complicated in predicate calculus since the procedure must also attempt to find a compatible set of substitutions for variables in each path. Bibel provides more detailed discussion on the advantages of the matrix connection method compared to both resolution and natural deduction techniques.

2.3.5 The Control Problem in Logic Theorem Proving

This chapter has shown how first order predicate calculus can be used to represent a wide variety of problems. In order to make useful deductions from these problem descriptions a variety of theorem proving techniques have been discussed. The uniform proof techniques require that logical formulas all be transformed to a single format while non-uniform methods work with facts and inference rules in their original forms. The problem with all theorem provers when faced with a realistic, non-trivial problem, is the choice of control strategy required to curb the combinational explosion in the size of the proof search space. Simple implementations will blindly apply inference rules, eg. resolution or paramodulation, to clauses in an arbitrary order. Given the inherent size of the search space, this is unlikely to result in useable systems for use on large or complex problems. Bundy(1983) is a good example of the practical problems that can arise. This is exactly the same situation as described earlier in the use of uninformed heuristic graph search methods. Indeed, by representing resolution based proofs as refutation trees, and the more general use of derivation graphs, it is clear there is a direct correspondence between theorem proving and heuristic state-space searches. State-space transformation operators simply correspond to the application of inference rules. Algorithms such as A^*, AO^*, show that heuristic estimates of the effect involved, cost of solution path etc., are required to improve the efficiency of depth first, breadth first or backtracking procedures.

The control problem in resolution based theorem proving stems from the choice of clause used to resolve a literal away. The order in which literals are resolved is not really a problem since they must all be eventually resolved. The choice of parent clause is, however, a "real" choice in that one path may lead quickly to the empty clause while others introduce new, redundant variables and increase the search space. The basic problem comes down to knowing, or guessing well, which rules, (or operators) to apply, when, and to what facts (or nodes), and embodying these heuristic estimates in the control strategy.

Strategies such a linear-input LUSH or SL resolution provide simple domain

independent heuristics over clause choice. Extended forms of semantic checking, including procedural attachment, can further reduce the number of choice options for each application of an inference rule within a given strategy. It is the general consensus, however, that domain dependent heuristics must also be used to provide much strictor search control. Bundy(1983) discusses in some depth the use of heuristics in theorem proving in the context of equation solving and mathematical modelling in general. A listing of a concise heuristic theorem prover, written in Prolog (see section 2.3.6 below) is also provided.

A more detailed discussion of the use of domain dependent heuristics and meta-information, i.e. information on how to use other pieces of information, such as when to apply an inference rule, is given in section 3.2.

2.3.6 Logic Programming and Prolog

Logic programming is a style of programming based on the ideas introduced in this chapter. Ideas such as automatic inference, pattern matching, inference rule and fact representation are taken to be programming primitives out of which more complex systems can be built. One of the main proponents of this style is Kowalski(1975). The use of deduction systems to perform computations is appealing as the control strategy and its implementation can be cledarly separated from the representation of information.

Other than specialised logic programs, the most widely used, practical programming language is PROLOG, originally developed by Colmerauer et al in the early seventies and most notably by Warren(1977). Prolog is basically a Lush Resolution theorem prover employing a depth first, backbracking control strategy. As such, a prolog program is a set of Horn clauses defining a set of predicates. As with other symbolic languages, most notably Lisp, there is no distinction between program and data. All "programs" are data structures and are evaluated by an external interpreter, in Prolog's case the particular theorem prover in use. In order to make Prolog a practical programming language several built in predicates are supplied which are not satisfied by resolution but are evaluated and then deleted from the clause. A simple example is the "write" predicate which displays Ascii strings on a terminal. Higher order logics can be simulated as predicates are provided to analyses predicate structure, i.e. the name and arguments of the predicate, to construct new predicates, collect the set of instances of any given predicate schemata etc. A "cut" symbol (! is used) is provided to irrevocably prune of parts of the search tree, thus augmenting the basic control strategy.

As Prolog handles Horn clauses it can represent the following types of clause.

1. Facts and Assertions: General facts, data statements etc. are represented by n-ary predicates, e.g.
   temperature(r101, 298). is a prolog "fact".
   Variables are expressed by upper case letters and predicate or functional constants by lower case letters. The underscore symbol " " is the special "don't care" variable, e.g.
   temperature (r101,_), r101 has some temperature.
2. Implications Clauses: $A_1 \wedge A_2 \wedge \ldots \wedge A_n \rightarrow C$, which express a consequent, C, given a number of antecedents $A_1 \ldots A_n$. This can be used to represent heuristic "if ..... then ..." rules, or in a procedural context, express how a goal C is proved, or computed, in terms of subgoals. In Prolog this is written as
   goal-name (Arg1,...., ArgN):-
   sub-goal 1 (.....),
   sub-goal 2 (.....),
   ...,
   sub-goal N (.....).
   The "if" operator, :-, can be replaced by any sequence of allowed letters and symbols. For example, to express heuristic rules it can be written using the pair of strings "if" and "then",
   if antecedent 1 (Arg1,....),
   antecedent 2 (ArgA,....),
   ....,
   antecedent N (ArgN,....)
   then consequent (ArgC,....).
   The logical "and" connective in Prolog is the comma, "," symbol but this can also be redefined if necessary.

To compute a Prolog program, or prove some goal clause, a goal statement is taken by the interpreter and "proved" by Lush resolution plus evaluation of any special predicates.

Since Lush resolution based on depth first search is a weak theorem proving technique, most Prolog applications use the built-in facilities to construct more powerful systems on top. This is the approach taken throughout this thesis.

Prolog is widely used as a programming language in AI. Detailed introductions can be found in Clocksin and Mellish(1984), Sterling and Shapiro(1986). The use of Prolog in constructing database systems can be found in Gardarin and Gelenbe(1984).

For a comparison of Prolog and Lisp as general symbolic evaluation systems see Warren and Pereira(1977). The choice between the use of Prolog or Lisp is somewhat irrelevant within the context of this thesis. The important point is a commitment to a clean separation between data representation and its manipulation and interpretation rather than commitment to any particular implementation.

2.4 Generalised Production Systems

One of the major research areas in Artificial Intelligence over the last fifteen years has been the development of domain independent production systems. Many of these early systems were referred to as "expert systems". The term is not used here as it is considered to be both somewhat inappropriate and imprecise. An assessment of what may or may not constitute an "expert system" is given at the end of this section.

The purpose of this section is to describe the underlying structure representation and interpretative mechanisms available in this class of system. In order to understand the nature of these systems in a more precise way, their relationship to techniques developed in heuristic graph search and logic based theorem proving is made clear whenever possible. The potential contribution of the use of production systems in the development of a process engineering design environment, in terms of how suitable they are for representing certain types of information, is then discussed.

Problem solving activities in the area of heuristic graph search, whether optimising or satisfying searches, consist of three basic characteristics.

1. A symbolic structure or code which represents each candidate object or sub-problem in the search space.
2. Computational "operators" or "functions" that transform the encoding of one object to that of another.
3. A control strategy, or scheduling algorithm, to perform the object transformations via the application of operators. It is generally desirable to reach the desired goal or object state with near minimum search effort.

These solution technique characteristics are mirrored exactly in logic based theorem proving systems.

1. A set of logical axioms.
2. A set of inference rules for deriving new wffs from previous wffs.
3. A theorem proving or control strategy used to order the application of the inference rules to arrive at or prove the desired goal statement with minimum effort.

Research in both areas has indicated that good, domain or problem dependent heuristic estimates are required to satisfactorily guide the search. State- space heuristic search based on the A^* or AO^* class of algorithm, including the special case of branch and bound optimisations, require assessment of the expected activity of operator application. The order or schedule of operators to be used must be continually revised as the solution progresses. The same is true in problem reduction formulations in which the order of sub-problem to tackle next must be constantly reviewed using some appropriate heuristics. Domain independent heuristics, such as limiting depth search to a certain level then reverting to a uniform cost or breadth like

strategy, do generally help but are insufficient in large or complex problem spaces. Experience in the area of heuristic process synthesis algorithms shows that this approach is unlikely to succeed, when applied in isolation to many process design problems. Much of the problem appears to be due to an inadequately low level of representation and the consequent inability to reason about the search control strategy at a high enough level of abstraction.

Logic based deduction systems offer a much richer form of problem representation than do many early heuristic systems. Experience shows however, e.g. Bundy (1983), that their practical application to complex problems is unlikely to succeed in the absence of good heuristics to guide the selection and scheduling of inference rules and their application.

These problems gave rise to the formulation and development of more generalised "production systems" which incorporate many characteristics of both heuristic search and theorem proving techniques. One of the early systems developed was in the area of psychological modelling, Newell and Simon(1972). Information expressed in the form of "if... then..." statements were referred to as productions. The layout and operation of a typical production system, as described below, generalises the work of Newell and Simon and applies to the majority of systems developed to date. The approach taken have, following that of Nilsson, is to make clear the relationship between production systems and the equivalent logical formalism. The problems involved is using such representation systems in a process design environment can then be more clearly appreciated.

Corresponding to the three basic components of heuristic search and theorem proving systems, the components in a production system are:

1. A global data base.
2. A set of production rules, and
3. A central system.

The term "global database" is used to refer to the central data structure(s) used in any particular implementation to store both problem specific and invariant data/information. The word database does not indicate actual use of a database system, although it does not exclude it, but any data storage mechanism appropriate to the situation. The data structure is not fixed in size but grows in size as problem data is supplied, new deductions are made, etc. As such, it effectively provides one working memory for a production system. In logic terms, the global database is simply a set of wff's, clauses, facts etc. The set of clauses is constantly modified as theorems are proved, contradictions are found and so on. The advantage of a logic

based data representation is that a precise semantics can then be defined for its subsequent interpretation and manipulation.

The set of production rules in use define the operations that can be performed or applied to the global database. Following Newell and Simon(1972), production rules are invariably expressed as "if.....then....." constructions, typically

```
if Premise1 and Premise2 ... and PremiseN
   then Conclusion1 and Conclusion2 ... and ConclusionN
```

This, of course, is exactly the same as any inference rule in logic representations.

$$P1 \wedge P2 \wedge \ldots \wedge PN \rightarrow C1 \wedge C2 \wedge \ldots \wedge CN$$

or,

$$\neg(P1 \wedge P2 \wedge \ldots \wedge PN) \vee (C1 \wedge C2 \wedge \ldots \wedge CN).$$

A number of terms are used to describe these rules. The left hand side of a rule refers to the antecedent or set of conditions or hypotheses of the rule, while the right hand side of the rule refers to the consequent or conclusions of the rule. If the right hand side of the rule represents an action to take, as is the case in problem reduction formulations, then the rules are referred to as "condition-action" or "situation-action" rules,

```
e.g.          "If appropriate point in design has been reached
               then solve separation sequencing problem"
```

Many systems express the rule conditions as "object-attribute-value" triples, e.g. (r101 temperature 350), which indicates that the attribute temperature of object r101 has value 350. This is exactly how information is represented in logic via the use of predicates. When writing generic rules or heuristics the actual values and object constants will be replaced by variables. If variables are represented by capitalised words, a simple generic rule could be,

```
if (R temperature Tactual) and
(R safe-operating-temperature  Tsafe) and
Tactual > Tsafe
then (R condition dangerous).
```

At any given instant the antecedent of a rule may or may not be "satisfied" or "matched" by facts, given or derived, in the global database. It is normally the case that any one of a number of rules could satisfy their antecedents. It is then the responsibility of the control strategy to decide which rule to apply to the global database. This is very similar to the process of deciding which inference rule to apply in resolution based theorem proving and is discussed in more detail shortly. The overall aim of the production is to arrive at or deduce some goal state or answer

statement which it does by repeated rule application governed by the control strategy. The goal state is recognised by providing suitable termination conditions, either explicitly or system defined, describing parts of the global database, eg existence of a certain set of facts with acceptable object-attribute values. Figure 2.20 shows an outline algorithm for any production system.

Figure 2.20: General Algorithm For a Production System

```
1)  Assign Data Set to the initial data,
    eg problem details and general facts.

2)  While termination conditions are not satisfied by Data Set

    START

    3)  Select a rule R that can be applied to Data Set.

    4)  Update Data Set having applied R to Data Set.

    FINISH
```

The algorithm is too general to be of real use but indicates the essential operations, more detailed implementations being described below. It is worth comparing, however, with the graph search algorithm, Figure 2.4, and the resolution refutation algorithm, Figure 2.10. The control strategy in use determines the rules to be selected in Step 3. As in heuristic search, the strategies employed can be classified into one of two broad categories, namely irrevocable and tentative. When a rule is applied in an irrevocable regime its application and effects cannot be reconsidered.

In a tentative strategy, however, a mechanism is provided to reconsider the use of a rule or any other possible alternatives at a later point in the search. This can either be effected by means of backtracking or via general graph-search approach as discussed earlier in section 2.2.

The mode of operation of a production system, as distinct from the style of control strategy, is normally classified as either forward or backward chaining in nature. The concepts of forward, data driven bottom-up strategies and backward, goal drive, top-down strategies were briefly introduced earlier in the context of theorem proving. In a rule based system forward chaining simply refers to process of chaining forward from known conditions in the global database which match against rule conditions towards allowable conclusions. An intermediate conclusion can be used to match part of another rule condition, hence the use of "forward" :

```
Given fact y : deduce z ...

    if y then x.

    if x then z
```

In contrast, backward chaining refers to the reverse process of hypothesising some goal conclusion expressed as a rule consequent, and trying to find terms to match against the rule conditions. A single rule condition within an overall conjunction can become an intermediate or sub-goal which can be used to match against the consequents of other rules :

```
Hypothesise goal z, verify fact y ...
    if y then x.

    if x then z
```

Before describing specific systems in more detail and their application to chemical engineering design problems, the operation of a production system shall be re-expressed in a logic based framework. Informal descriptions of production systems can be found in several introductory texts, eg Hayes-Roth,Waterman, and Lenat(1983), Barr and Feigenbaum(1981), Jackson(1986).

2.4.1 Production Systems in a Logic Based Framework

This section briefly outlines the relationship between certain aspects of production system operation and logic based theorem proving. This is important for two reasons :

1. It permits comparison of a number of ad-hoc, implementation specific details found in a variety of systems in a common representation.
2. The precise nature of the operation of production systems can be understood more clearly.

If production systems are to be used in decision making procedures in process design it is essential that the decision procedure and validity of any subsequent conclusions or decisions made be clearly understood. Logic offers a precise semantics for this analysis task.

2.4.1.1 Direct Representations, Kowalski Form and Prolog

Most production systems do not convert rules and facts to normal clausal form but use them in a form equivalent to the original statement. These systems are termed direct representation systems. The idea is to preserve order information implicit in the sequencing of rule promises or consequents which can help guide the search. This is the same approach taken by non-clausal and natural deduction theorem provers.

Rather than transform an if-other rule of the form

"if P, and P2 ...Pn then C"

to the clausal form,

$$\neg P_1 \vee \ldots \vee \neg P_n \vee C \ ,$$

the equivalent Kowalski form can be used

$$P_1 \wedge P_2 \wedge \ldots P_n \rightarrow C.$$

The arrow is often drawn the other way round, giving

$$C \leftarrow P_1 \wedge P_2 \wedge \ldots P_n ,$$

which can be directly represented as a Horn clause in Prolog,

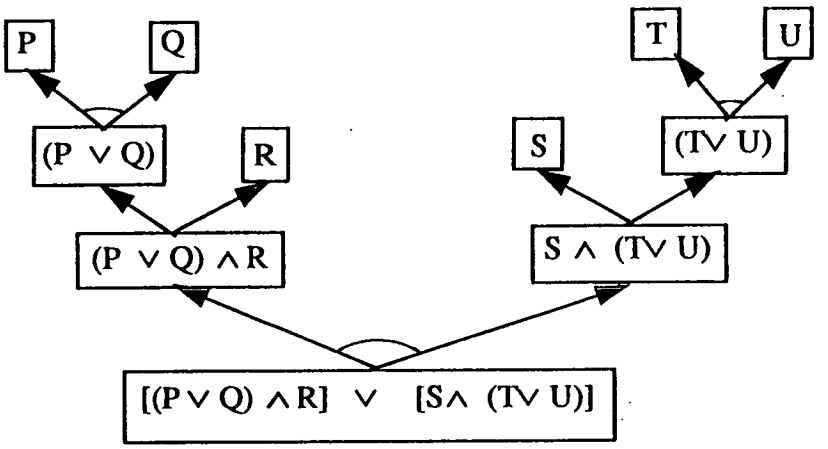
```
e.g. c :- p1, p2, pn.
```

The use of symbols such as ":-" and "," is a syntactic convention and can be changed back to "if", and "then" if desired.

General facts can be directly represented by predicates in first order logic. Section 2.3.4 illustrated representing such facts in Prolog. These facts are stored in an internal database in the prolog interpreter environment. In the context of production systems, Prolog's internal database is the global database structure discussed earlier.

Nilsson(1982) generalises "facts" to be "fact expressions" of an arbitrary AND/OR form that can be represented by AND/OR graphs, see section 2.2. Conjunctive sub expressions E_1, ..., E_n of an expression, i.e. $E_1 \wedge E_2 \ldots \wedge E_n$, are represented by individual descendant nodes connected to their parent node by a 1 connector, see Figure 2.21.

Figure 2.21: Representing Fact Expressions



Disjunctive sub expressions, $E_1 \vee E_2 \ldots \vee E_n$, are represented by descendant nodes connected by a k-connector to their parent. The leaf nodes of a graph represent the literals in the expression, the root node being the entire expression. The reason for reversing the role of k-connectors and 1-connectors, normally associated with AND

nodes and OR nodes respectively but reversed here to OR nodes and AND nodes, is to more easily describe possible solution proof graphs as now explained.

The usefulness of an AND/OR graph to represent a fact expression is to understand clearly the process of applying rules in a production system to a global database. In this case, the entire AND/OR graph represents the state of the global database, e.g. a set of facts represented as a complex expression. The following example is given in Nilsson(1982).

Suppose a simplified rule of the form "literal $\rightarrow$ wff", e.g. $S \rightarrow ((X \wedge Y) \vee Z)$, is applied to the fact expression in Figure 2.21. Given a fact expression F(L) and a rule $L \rightarrow W$, the fact expression F(W) can be derived by substituting all occurrences of L in F by W. This is shown in Figure 2.22. The result is that the original graph is transformed to a new graph which also represents F(W). Alternatively, the set of solution graphs terminating in leaf nodes correspond to the set of clauses in the clause form of (F(W). To see this more clearly consider the resolution process steps. The clause form of the rule $S \rightarrow ((X \wedge Y) \vee Z)$ is $\neg S \vee X \vee Z$ and $\neg S \vee Y \vee Z$, and the clauses resulting from the clause form of $[(P \vee Q) \wedge R] \vee [S \wedge (T \vee U)]$ that would resolve on S with the rule clauses are $P \vee Q \vee S$ and $R \vee S$. The complete set of resolvents that could result are
$\{X \vee Z \vee P \vee Q, Y \vee Z \vee P \vee Q, R \vee Y \vee Z, R \vee X \vee Z\}$
Figure 2.22 shows that all these clauses are represented in the solution graphs of the transformed graph, i.e. it includes the entire set of resolvents produced by performing all the possible resolutions on L between the clause forms of fact F(L) and rule $L \rightarrow W$

The essential point to note is that in this formulation the process of rule application achieves the same results as the resolution process but in an efficient manner. Applying a rule to an AND/OR graph results in a transformed graph which represents both the original fact expression and new inferred facts. Rule application is represented by a "match" arc or link. The original node, to which the rule was applied, is no longer a leaf node but can still have other rules applied to it if necessary. The order of rule application and the nodes selected, ie those nodes which match the left hand side or right hand side parts of the rule, depends on the control strategy in use.

The process of "matching" parts of rules, either the antecedent or consequent in forward or backward chaining respectively can now be more precisely defined. The use of variables in direct representation systems, however, must be introduced.

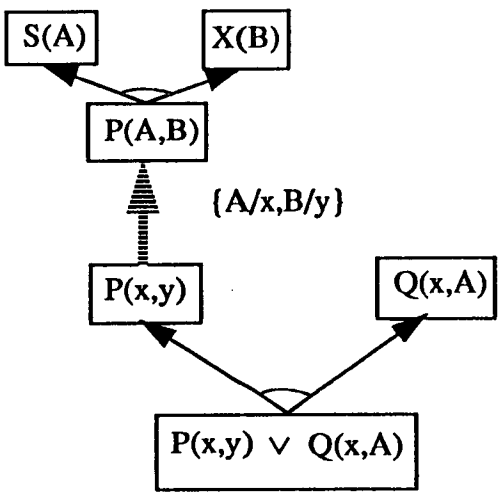
Generic rules will usually be written in terms of variables that have implicit universal quantification, e.g $\forall x$ reactor(x) $\rightarrow$ vessel(x). In order to handle both universal and existential variables in direct representation systems a dual version of the Skolemisation process is used. Universally quantified variables are now replaced by Skolem functions in goals within the scope of existential variables. The existential quantifiers are then dropped. The variables that remain in goal expressions or wffs are assumed to be existentially quantified. This is to be expected in a direct system since the application of rules will result in matches derived from specific facts in the global database referring to named individual constants.

Consider a typical object-attribute-value triple represented by a fact, e.g. temperature(r101, 1000). A generic rule may refer to the temperature in the form,

$P_1 \wedge \ldots \wedge$ temperature(x,tval) $\wedge$ greater_than(tval,900)
$\rightarrow$ condition(x,overheated) $\vee \ldots \vee C_n$.

Figure 2.23 shows in general terms how a rule like this would be matched agains a fact expression using a graph representation. The thick arrow represents matching the fact with a rule.

Figure 2.23: Matching Facts with Rules Involving Variables



Associated with each link is a set of variable substitutions. A consistent solution path in a graph now involves finding the most general unification, or unifying composition, of the set of substitutions labelling each of the terminal leaf nodes in the graph. Application of rules which result in inconsistent variable substitutions will not lie on solution paths within the graph. The unification procedure required is exactly the same as that described earlier in theorem proving, section 2.3.

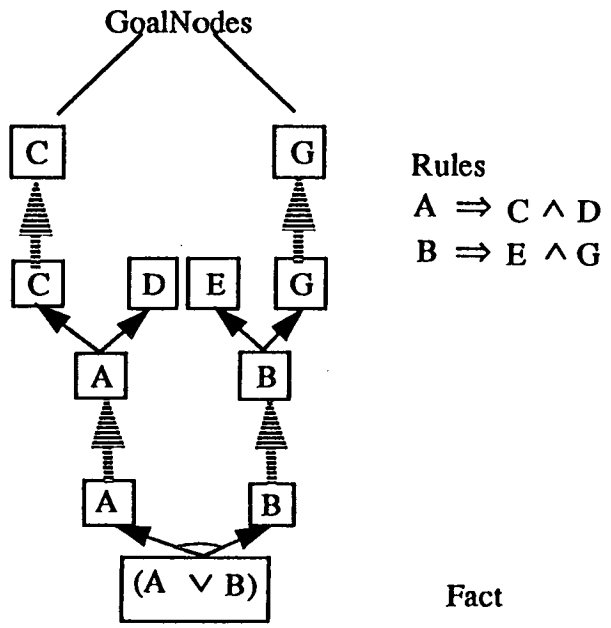
Thus, pattern matching parts of rules against facts in the global database is equivalent to the process of finding consistant unifying compositions if the pattern matching is viewed in the global context of finding a solution path in an AND/OR graph representing the application of rules to fact expressions.

2.4.1.2 Forward Chaining Deduction Systems

The process of repeatedly applying rules to nodes in a fact expression AND/OR graph, as described so far, is usually called forward chaining. More precisely, the search is driven by pattern matching between the rule antecedents and the assertions and axioms represented in the global database. The termination condition for a forward deduction system is to prove some goal wff that can be derived from the facts and rules available.

Consider a goal wff whose general form is a disjunction of literals, i.e. $A \vee B \vee ... \vee C$. Each literal in the goal wff can be simply represented in the AND/OR graph by a match arc between the goal literal and a suitable matching node. The forward search will end when the AND/OR graph is transformed to one containing a solution graph terminating in goal nodes. Figure 2.24 illustrates a simple example from Nilsson(1982).

Figure 2.24: Representing Forward Chaining



The reason for using k-connectors with OR nodes is now clear. If a complete record of the final solution graph is desired and it is unknown whether A or B are true,

Figure 2.24, but only that the disjunction A ∨ B is true then the solution graph must include all the descendant nodes of a disjunctive node n. This represents the "reasoning by cases" approach: first assume A is true then assume B is true, and so on.

It is clear that the operation of forward chaining production systems can be re-expressed in standard theorem proving notation. The use of direct representation systems and AND/OR graphs to represent fact expressions makes clear the processes involved.

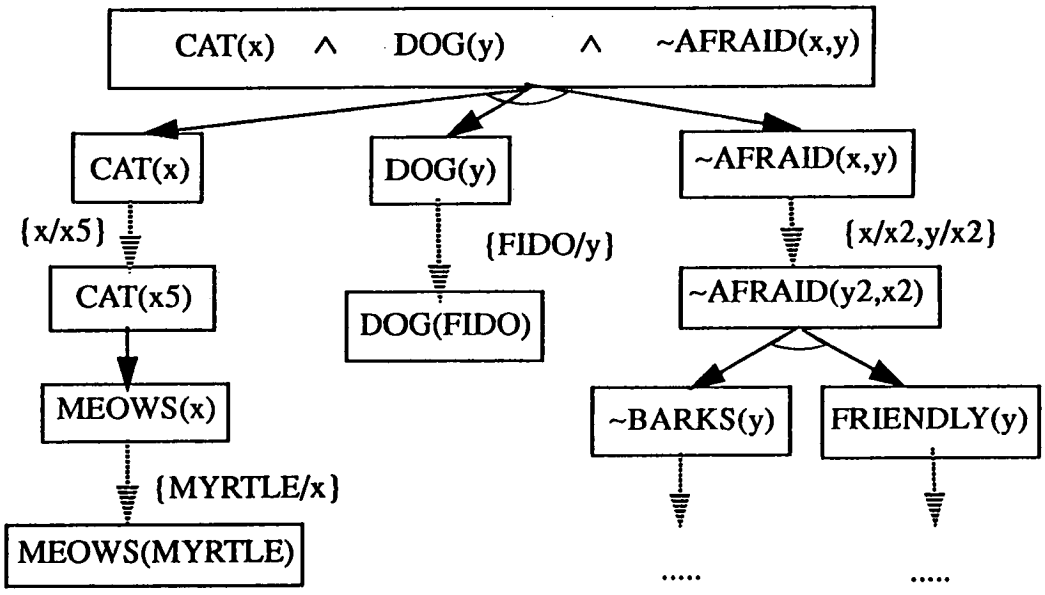
Nilsson provides more detailed discussion on how various other extensions to direct representation systems are handled, e.g. resolutions between nodes in a graph, conditional resolution, repeated use of a fact in a proff involving renaming of variables etc.

2.4.1.3 Backward Chaining Deduction Systems

The process of rule selection and application in backward chaining systems can be described in a similar way to forward chaining systems. The essence of backward chaining or top-down deduction is that the search starts from the conclusion of the theorem, i.e. the goal wff, and works backwards towards the hypotheses and axioms of the theorem.

In this case, it is simpler, and more natural, to express the desired goal wff in the form of an AND/OR graph. The goal wff is processed as in the forward chaining case, i.e. implication symbols are rewritten, universal variables are Skolemised and the remaining variables are assumed to be existentially quantified. A simple example is shown in Figure 2.25. The goal expression establishes whether there is a cat that is not afraid of a dog.

Figure 2.25: Representing Goal Expressions and Backward Chaining



Notice that k-connectors are used to connect descendants of conjunctions, not disjunctions as in forward chaining. The reason, as before, is to use the AND/OR graph to represent the solution graph. Descendant nodes in backward chaining systems are usually referred to as subgoal nodes. Finally, the clause form of the goal wff is simply the disjunction of the clauses represented by the leaf nodes of the graph.

The process of selecting and applying rules is the dual of that for forward chaining. Consider a rule of the form $W \rightarrow L$, where L is a literal and W is an arbitrary wff. The rule can be applied if the rule consequent L unifies with a node L' in the AND/OR graph. If so, a new node is generated which represents the most general unifier of L and L', the match arc between these nodes being labelled by the unifying substitution. Descendants of this node represent the new subgoals given by W, see Figure 2.25.

From a theorem proving point of view all that has happened is that the rule $W \rightarrow L$ has been negated and disjunctively added to the goal wff. The negated form is $(W \wedge \neg L)$ and the transformed graph includes a representation of the set of resolvents that can be derived by performing all resolutions on L between the goal wff and $(W \wedge \neg L)$. These clauses will be represented on consistent solution graphs of the final transformed AND/OR graph.

The backward search terminates when the leaf nodes descending from k-connectors

can be matched with correspond facts literals. A consistant solution graph is one terminating in fact nodes and where the substitutions labelling the match arcs have a unifying composition.

The accounts of forward and backward draining have been somewhat simplified here. Various extensions are required to enable certain types of "intra-graph" resolutions to be performed. For example, it may sometimes be necessary to perform a resolution between nodes representing complementary but unifiable literals, P(x,A) and ¬P(B,y) say, to find a solution graph. This process is called restricted goal resolution, RGR, the match arc between the nodes being labelled by the required substitution as usual, {B/x, A/y}. Joining partial solution graphs in this way can result in a candidate solution graph if all the other nodes are goal nodes, or nodes participating in RGR matches, and the unifying substitutions are consistent.

In direct representation systems mechanisms are required to handle resolutions that would otherwise be carried out if the formulas were multiplied out into clausal form. An example of this is the use of conditional substitutions. Consider a goal wff of the form $[P_1(x,y) \vee P_2(x,y)] \wedge P_3(x,y)$ and suppose that $P_3(x,y)$ can match with either $P_3(A,B)$ or $P_3(B,B)$ in the global database. Applying inference rules to the node $P_1(x,y)$ may result in the node Q(x,B) whilst the path leading from $P_2(x,y)$ may lead to ¬Q(A,y) Now the alternative paths lead to two complementary literals, Q(x,B) and ¬Q(A,y) which have the most general unifier {A/x, B/y}. Applying this substitution to Q(x,B) yields Q(A,B), ¬Q(A,y) yielding Q(A,B). Only one of these can be true so further search can be terminated and can be represented by a conditional substitution labelling the match arcs with nodes representing True. That is an "if" condition is used to label a match arc which points from some given node to a node that is labelled True. Nilsson(1982) describes some of these mechanisms in more detail. Loveland(1978) is a useful discussion on the relationship between AND/OR graph representations and resolution. The subject of conditional substitutions is treated in Manna and Waldinger(1979) in the context of automated synthesis of computer programs.

2.5 Implementation Aspects of Production Systems

The previous section described in theorem proving terms the operation of forward and backward chaining production systems. Given this understanding it is now possible to briefly discuss the implementation of three well known production systems, namely MYCIN, AL/X (based on Prospector) and OPS5. The purpose is to highlight the representation and control mechanisms available in these and similar systems and how they can be applied to various aspects of process design.

Consider the algorithm describing the application of rules in production systems, Figure 2.20 earlier. This is often described as a three phase recognise/act cycle:

1. The Match Phase - identify the rules whose conditions hold in the context of the current global database. This may involve backtracking to a previous choice point in some control strategies.
2. The Conflict Resolution Phase - Select one of the rules identified in the Match phase. If no rules are applicable then the search will terminate.
3. The Act Phase - Perform any actions indicated by the rule. If the goal is achieved then the search can end, otherwise the recognise/act cycle is repeated.

The recognise/act cycle has a clear relationship with the application of inference rules in theorem proving systems. This correspondence is now used to compare the production system exemplars.

2.5.1 The Mycin System

The Mycin production system was developed for the medical diagnosis of bacterial infections. It is one of the most well known systems and gave rise to a large number of similar development in many unrelated research areas, including engineering design. The system dates from 1972, a comprehensive discussion being given in Shortliffe(1976). Mycin is worth discussing as it is typical of a large class of consultation type programs which perform diagnoses and offer advice.

The basic purpose of Mycin is to produce a set of possible hypotheses concerning the likely identity of the bacteria causing the infection and subsequently recommend drug therapies based on these hypotheses. The three main areas of interest are

1. The form of production rules and static data structures used to represent the specialist domain information.
2. The control strategy
3. The dynamic data structures used in arriving at conclusions.

These are now briefly discussed in turn and then assessed from a logic based point of view.

2.5.1.1 Representation of Facts and Rules in Mycin

Representation of simple definitional facts in Mycin make use of the facilities of the list processing language, InterLisp, in which it is written. Three important data structures can be identified:

1. Simple lists denoting the basic data types known to the system, eg. all the known ORGANISMS and STERILESITES (upper case words are MYCIN terms).
2. Tables of records of values of clinical parameters under various circumstances.
3. A classification structure that denotes which parameters are relevant to Mycin "contexts". Mycin contexts can be thought of sub-problem contexts which are labelled patient, culture, organism etc., and are explained below. So this information states whether a parameter is an attribute of a patient, organism etc.

Much of Mycin's information is expressed in the Lisp property lists describing the parameters associated with each context type (A property list in Lisp is essentially a vector which can be accessed by symbolic indices, eg. "age", "name", rather then by numerical indices). For example, the context type PERSON has a set of parameters including NAME, AGE, SEX, ALLERGIES, SEVERITY, BURN etc. Each parameter in turn is classified as one of :

- A Yes/No parameter. Context types either have this attribute or not and questions seeking a value of this parameter expect a yes or a no as an answer, eg. the patient attribute FEBRILE, ie. has fever.
- A single valued parameter. Values of these parameters exclude all other possible values, eg. the identity attribute, IDENT, of organisms is treated this way.
- A multi-valued parameter, eg. a patient could have more than one infection, the INFECT parameter.

A number of other properties are used to further describe each parameter and help guide their use during search:

- The EXPECT property which indicates the range of possible values for the parameter.
- The LABDATA property. This indicates whether the parameter represents a data value which the user may know. This is further classified as ASKABLE or NON-ASKABLE. Mycin will ask for a value, if ASKABLE, before trying to deduce a value.
- The PROMPT and TRANS properties. The former is a question to be used when asking for the value and the latter is a phrase describing the parameter in English. The TRANS property is used for displaying rules in an English like format.
- The LOOKAHEAD property lists all the production rules that mention the parameter in their premises.
- The UPDATED-BY property is the dual of LOOKAHEAD and lists all rules that refer to the parameter in their action or conclusion part.

As a simple example here is what the BURN parameter associated with the PERSON context might look like:

```
(BURN       (MEMBEROF:    PROP-PT)
            (VALUETYPE:   BINARY )
            (EXPECT:      (YN)
            (PROMPT:      (IS * A BURN PATIENT?))
```

```
(LABDATA:      T)
(LOOKAHEAD:     (RULE_047 RULE_048))
(TRANS:       (* has been seriously burned)))
```

The *'s in the PROMPT/TRANS get replaced by suitable words when used, eg. the patient's name.

The production rules in MYCIN refer to these parameters and their values to express the domain specific knowledge and heuristics that the system has access to. The rule format used is straightforward. A rule consists of a PREMISE and an ACTION. The premise can be any number of condition forms in a conjunction, $AND being used as the connective and conditions being of the form :

(predicate context parameter value)

or

(predicate context parameter)

Predicates refer to system functions such as SAME, CONCLUDE etc. Any number of conditions can be connected by the $OR connective within a single disjunctive condition. An example rule might be

```
(RULE_048 (PREMISE ($AND (SAME CNTXT SITE BLOOD)
                         (NOTDEFINITE CNTXT IDENT)
                         (SAME CNTXT STAIN GRAMNEG)
                         (SAME CNTXT MORPH ROD)
                         (SAME CNTXT BURN T)
          ACTION (CONCLUDE CNTXT IDNT PSEUDOMONAS TALLY 0.4)))
```

There are two points to note about Mycin rules. First, rules are always categorised according to the context in which they are most appropriately used. This will be the lowest context type in the classification hierachy. In the example above the BURN parameter is associated with the PERSON context, SITE with the CURCUL context, and the others with the CURORG context, i.e. the current organism. As CURORG is the lowest type in the context hierarchy for this rule, the rule is associated with this context type.

The second point to note is that the rule makes a conclusion regarding the identity, IDENT, of the current organism, CURORG, with a certain degree of confidence, the TALLY value. Handling imprecise information or making guesses in this way raises complex issues which are discussed later, chapter 4.

2.5.1.2 Mycin's Control Strategy

Mycin uses a simple backward chaining search which only varies slightly from ordinary depth first search. There is a single top level rule associated with the PATIENT context:

```
IF      there is an organism which requires therapy,
        and consideration has been given to the
```

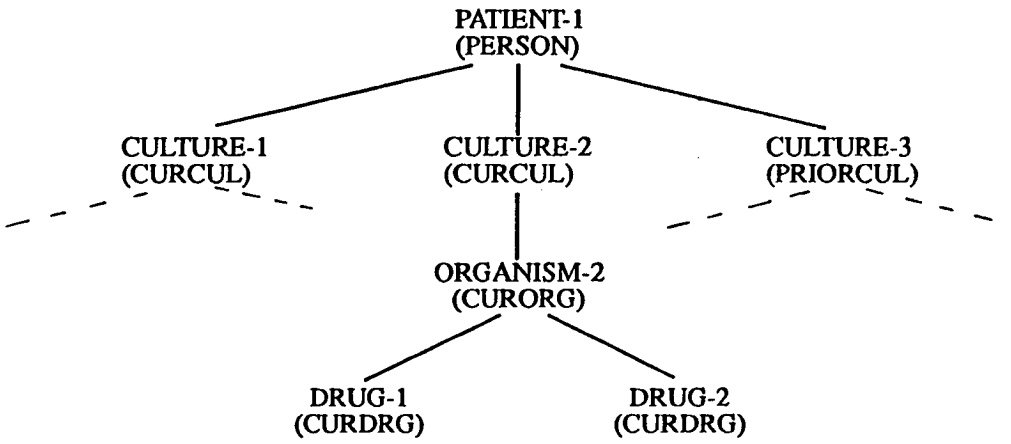
```
        possible existence of additional
        organisms requiring therapy
THEN    compile a list of possible therapies,
        and determine the best therapy in the list,
```

i.e. the rule states that the best drug therapy is to be found which will treat the identified organism and take account of the existence of other possible organisms not yet identified in laboratory samples.

The rule is applied to the PATIENT context. In order to find a therapy the conditions in the rule premise must be evaluated. This involves finding an organism which requires therapy. This can be determined by directly asking the user or by involving rules relating to the organism and culture contexts, CURORG and CURCUL respectively. These rules will gradually work back towards user provided facts and the tables of values stored for chemical parameters in different contexts. The choice and application of rules is guided by the LOOKAHEAD and UPDATED-BY properties of parameters described earlier.

The search, or proof, is represented by a "context tree" which is dynamically extended during the search. The top node in the tree is the PATIENT context. New contexts are added to the tree when they are referenced by parameters in the antecedents of rules associated with another context. Figure 2.26 shows an example of a context tree, leaf nodes indicating alternative DRUG contexts. The use of CONTXT in the rule above is a variable referring to the currently active context node.

Figure 2.26: Fragment of a Mycin Context Tree



The selection and application of rules is implemented by two procedures called MONITOR and FINDOUT. MONITOR is used to evaluate the truth of the rule antecedent by evaluating each condition in turn. If a condition is false or indeterminate, due to lack of positive or negative evidence, then the rule application

fails and the next rule on the list of possible rules is tried. If the rule premise evaluates to true, or possibly true, then the rule conclusion is added to another dynamic data structure called the "consultation record". Mycin then returns to previous choice points in the context tree and expands them in the usual depth first fashion. The FINDOUT procedure is used to find information that will count for or against a particular value in a rule premise. If the parameter is classed as LABDATA/ASKABLE then the user is asked and control returns to MONITOR to evaluate the next condition in the rule antecedent. Otherwise the rules that reference the parameter in their conclusion, given by the UPDATED-BY property, are listed and returned for use in MONITOR's rule list.

There are two differences from simple depth first search:

1. If a condition in a rule premise refers to a particular parameter value, eg. the identity of the organism is Salmonella, then the subgoal that is actually set up is the generalised form of the proposition, eg. what is the identity of the organism, rather than prove the organism is Salmonella. This effectively "broadens" the depth first approach.
2. Every rule relevant to a goal is applied unless a rule is concluded with absolute certainty. Certainty is represented by a certainty factor ranging from +1.0, "definitely", to -1.0, "definitely not". If the evidence available about a parameter value results on a confidence factor between -0.2 and +0.2 the evidence is taken to be inconclusive and the parameter value treated as unknown.

Once the search for alternative diagnoses has terminated the second action in the top level goal can be performed, ie. select the best therapy from the alternatives. This is, in fact, performed by a set of Lisp functions which use static information about preferred drugs stored in therapy rules. The rule given earlier proposed the identify of the organism to be Pseudomonas. The therapy rule below shows the recommended drugs for this organism:

```
If      the identity of the organism is Pseudomonas
then    recommend therapy from the following drugs:
              1 - COLISTON          (0.98)
              2 - POLYMYXIN         (0.96)
              3 - GENTAMICIN        (0.96)
              ........ etc.
```

The numbers given with each drug are probabilities that the organism will be sensitive to the drug. The actual drugs selected are chosen to maximise sensitivity and minimise the number of drugs to be taken.

2.5.1.3 Explanation in Mycin

Mycin was one of the earliest systems to offer simple explanation facilities. The "consultation record" structure is continuously updated after rule applications recording the sequence of rules applied, the conclusions made, questions that were asked and how values of clinical parameters were obtained.

After a diagnosis has been made and a drug selected Mycin can, therefore, be queried as to HOW it arrived at its final conclusion, or any intermediate conclusion for that matter in terms of rule applications. During a consultation the user can also ask WHY a question is being asked. Mycin will respond by stating that it is trying to evaluate a condition in a rule to establish a certain conclusion. The user can recursively search back up the context tree in order to find out why the system is trying to establish a specific goal if so desired.

Much has been made of Mycin like explanation facilities in other rule based systems. These claims are not accepted here, in the context of knowledge integrity in a process design environment, and are discussed later in chapter 4.

2.5.1.4 Handling Imprecise Information in MYCIN

Mycin was one of the first production systems that attempted to handle imprecise information. Users of Mycin can reply to questions with definite answers or indicate uncertainty by qualifying the reply with a number between 0 and 1. The number is supposed to represent the user's "degree of belief" in the answer. Mycin rescales the [0,1] range to a [-1,1] range indicating negative or positive evidence for parameter values. A value in the range [-0.2,0.2] is taken to mean that the evidence for a parameter value is inconclusive.

Consider a rule applicable to organism contexts with the premise.

```
($AND          (SAME CNTXT     MORPH ROD)
               (SAME CNTXT     AIR AEROBIC))
```

Suppose the following information is known about the current organism, ORGANISM-1,

```
MORPH    =   (ROD 0.8)   (COCCUS 0.2)
AIR      =   (AEROBIC 0.6)  (FACUL 0.4),
```

i.e The morphology of ORGANISM-1 is rod like with a certainty of 0.8 and coccus with a certainty of 0.2, and the aerobicity is aerobic with a certainty of 0.6 and facultative with a certainty of 0.4.

The overall certainty factor for the rule antecedent is taken to be the minimum of each of the certainty factors in the conjunction, ie. 0.6 in this case. If the rule antecedent were a disjunction then the maximum certainty factor would have been chosen as the overall value.

Once the confidence factor for the rule premise is known the confidence factor for the conclusion is calculated as

$$CF(\text{conclusion}) = CF(\text{premise}) * CF(\text{rule})$$

. The certainty factor for a rule is given by the TALLY figure in the rule definition, see earlier example, representing the strength of the heuristic rule. If the TALLY had a value of 0.8, the certainty factor for the conclusion would be 0.6*0.8=0.48.

Once the certainty factor for a given conclusion is known it can be updated by combining it with other CF's as follows:

```
CF = CF1 + CF2 - CF1*CF2  if both CF1,CF2>0
   = CF1 + CF2 + CF1*CF2  if both CF1,CF2<0
   = (CF1 + CF2)/(1-min(CF1,CF2))  otherwise.
```

Mycin's certainty factors are not probabilities but are derived from Bayesian probabilities. Several authors have pointed out various flaws and problems in their use. Discussion is delayed, however, until chapter 4 when comparison with other possible approaches is made. For the moment it is useful to consider them as "scores" which can help Mycin rank alternative hypotheses.

2.5.2 Mycin from a Logic Point of View

Although later developments of Mycin became more complex, the initial Mycin system was essentially that described above. As such, Mycin is a straightforward implementation of a production system that can be easily characterised in logical terms.

Mycin's static data structures can be directly expressed as simple facts and predicates. The lists of known organisms, drugs etc. are simple 1-place predicates denoting types, e.g. drug(colistin), organism(e.coli), etc. Similary, the property lists describing clinical parameter values can be expressed as a set of binary predicates or a single n-ary predicate containing all the record information.

The growth of Mycin's context tree is exactly analogous to the process of applying inference rules to produce modified AND/OR graphs in direct representation systems. This was described earlier and the relationship to resolution based theorem proving was made clear. The control strategy of early versions of Mycin was simple and differed little from simple backward chaining theorem provers developed earlier. The use of the LOOKAHEAD and UPDATED-BY properties of parameters to reference possibly useful rules is, of course, just a variation on the use of connection graph techniques in logic based systems. The only real difference between Mycin and a direct representation system is its ability to handle uncertain information. This can also be accommodated, however, in a logical system. Associating a confidence factor with a predicate is similar to associating an extra argument to identify the situation or time etc. A practical implementation would have to represent this argument as a functional constant which had a suitable procedure attached to it to evaluate how the

certainty factors were to be updated on rule application, see the discussion on procedural attachment section 2.3 earlier.

The consultation record used in Mycin does not provide any additional functionality. Links between nodes in an AND/OR graph are normally only labelled by substitutions but the labelling information could easily be extended to identify the rule used, use of any attached procedures etc. This information would be used to provide the HOW/WHY facilities of Mycin.

Subsequent developments of Mycin resulted in a more sophisticated control strategy and are described briefly later. The exhaustive rule application approach was only feasible due to the guidance provided by the rule indexing mechanism and the relatively shallow level that rules could be expanded to (in the domain of interest covered by Mycin). In a wider problem domain in which problem parameters are highly inter-dependent, the typical situation in many design problems, such a simple approach is likely to suffer from performance problems similar to those experienced with general purpose theorem proving systems.

A version of Mycin was produced that was stripped of any medical information and was called Emycin (Essential or Empty Mycin), see van Melle et al.(1981). Emycin has its own rule language syntax which is more like Algol than lisp. As a result the rules are concise, using symbols like '=','>','and','or','if','then', and are easy to read. In addition, some debugging tools are provided to help the developers of systems. General production systems like Emycin are often referred to as rule-based "shells".

2.5.3 The Prospector-AL/X Systems

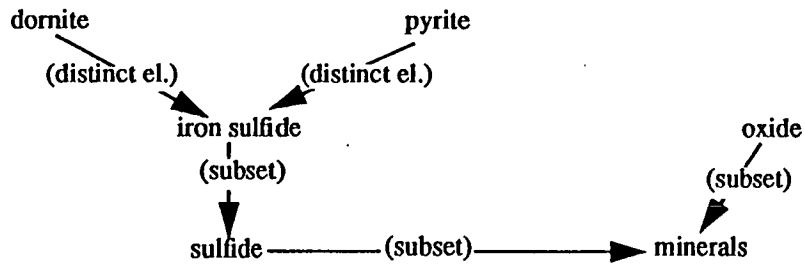
Prospector, like Mycin before it, was an interesting rule based base application which gave rise to many imitations and more complex developments. An account of the system can be found in Duda, Gasching and Hart(1979). AL/X is a general purpose system modelled closely on Prospector, see Jones(1984) for an analysis.

Prospector was written to assist geologists involved in hard rock mineral exploration to assess the likelihood of finding exploitable ore deposits in a given region. It is interesting from two points of view:

1. The data structures used to represent the general geological data and domain specific rules.
2. The mechanism used to handle imprecise or uncertain data.

The first data structure of interest is the graph, or "associative net", used to represent taxonomies of geological knowledge. Figure 2.27 shows a typical fragment of the graph.

Figure 2.27: An Associative Net in Prospector



The nodes in the graph denote the geological entities that can be referred to in rules similar to the lists of organisms, cultures etc., in Mycin. The links, however, are used to state taxonamic information relating these entities. This type of information is not used directly in Mycin but it is in Prospector. The links are labelled by terms such as subset, element, disjoint subset, distinct element, etc. These links are used to partition classes of rock, eg. sulphides, into subsets, eg iron sulphides, and particular elements, eg. dornite, in a hierarchical manner. This information is used by Prospector to drive the search. For example, if the user provides information regarding the presence of a particular sulphide, eg. dornite or pyrites, Prospector will not only forward chain on this fact but will forward chain on the assumptions that iron sulphides and sulphides are also present.

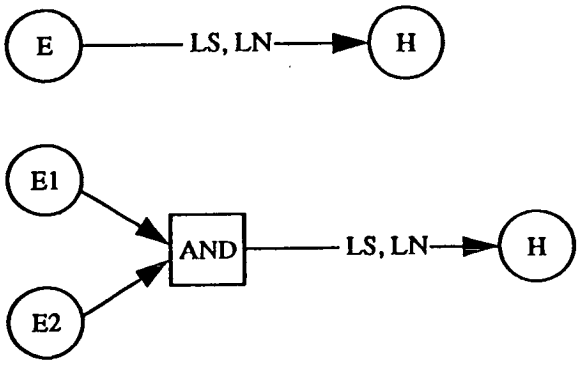
The hierachy links are also used to check the consistency of user supplied data in that

the user's confidence in the existence of a particular mineral, eg. pyrites, can be no higher than the confidence associated with its parent class, eg. sulphides.

Prospector's associative net is a simple example of what is more generally called a "semantic net". Semantic nets and object class hierarchies are discussed in detail in chapter 4.

The second interesting data structure used in Prospector is the "inference net" used to represent the inference rules. An inference net is similar to an associative net except that nodes represent individual assertions or logical connectives and the links represent implications. For example, the simple rules "if E then H" and "if E1 and E2 then H" represented as shown in Figure 2.28. The degree of belief in the implication is represented by two numbers, LS and LN, which label the implication links. These will be explained shortly.

Figure 2.28: Inference Nets in Prospector



Prospector maintained an inference net for each class of ore deposit, each net being called a "model'. In AL/X a model is called a "space". This is an important point as it means the rule set is now partitioned into a number of smaller subsets. In AL/X, each space can in turn be treated as a complex node which can be linked to other complex nodes. New problem spaces, or models, can then be built up in a modular fashion.

The other interesting aspect of Prospector, apart from the way it represents information, is how it handles uncertain data. Unlike Mycin, Prospector works directly with probabilities and odds, under certain simplifying assumptions, which are converted to certainty factors for user interactions. A certainty factor is a number in the range [-5,5]. This number gets rescaled internally to a probability. The way in which Prospector propagates probabilities through the inference nets is most easily

described by following the overall operation of the system.

1. The user is asked to volunteer observations. These can be qualified by a certainty factor in the range -5 to 5. For each piece of evidence Prospector has available to it a prior probability, P(E), estimating the likelihood of this fact. These prior probabilities are usually subjective estimates provided by an experienced geologist. What is required is an estimate of the probability of E given the user information, ie. p(E|E'). Rather than ask the user for this value directly a simple linear interpolation formula is used. A value of -5 corresponds to p(E|E')=0 and a value 5 corresponds to P(E|E')=1. Values in between are rescaled to P(E|E') using

$$\text{certainty value} = 5 * \frac{P(E|E') - P(E)}{P(E)} \quad \text{, negative evidence}$$

$$= 5 * \frac{P(E|E') - P(E)}{1 - P(E)} \quad \text{, positive evidence}$$

   The system uses these formulae to convert probabilities back to certainty factors when informing the user of any conclusions.
2. For each piece of evidence, and any subsequent assumptions derived from the associative net, probability adjustments are propagated through the inference nets by forward chaining from these facts towards the topmost goal in each net. These nodes will represent the possible presence of a financially worthwhile ore deposit. After propagation, one of the system models or spaces will seem most likely and is chosen as the current candidate.
3. The system then backward chains from the most likely node to try and increase its associated probability. The backward chaining path through the model is dynamically selected by a heuristic function which tries to select links that will contribute most to the probability of a node being confirmed or rejected. In the case of AND and OR nodes, the child nodes having the smallest and largest probabilities respectively are chosen. Backward chaining continues to a leaf node when the user is asked for information.
4. The user's reply is then forward chained.
5. The system then assesses the most promising model. If it is the same one then the system continues backward chaining towards another leaf node. If a new model is preferred then backward chaining is applied to the new model.
6. The search of a model terminates when all the relevant facts, ie. node assertions, have been processed.
7. The system eventually terminates when it has investigated all the relevant models. At any point in the search the user can interrupt to provide extra information or force attention to be switched from one model to another.

Probabilities are propagated in the intermediate forward chaining passes by using the two numbers, LS and LN, which label each inference link. LS, the level of sufficiency, is used to adjust probabilities when the evidence is positive and LN, the level of necessity, is used when the evidence is negative. These numbers are standard statistical likelihood ratios and indicate the effect of a piece of evidence, E, on a given hypothesis, H. Their definition and use in combining information is given in Appendix C. Given their definition it is seen that a large positive value for LS indicates that evidence E strongly suggests hypothesis H. LN indicates how odds are changed by observing a lack of the evidence and a value close to zero suggests that

evidence E is highly necessary for hypothesis H, since if E is not observed then the odds for H fall to zero. The possible combinations of LN and LS must be one of

1. LS>1 and LN<1, the odds on H increase if E is observed
2. LS<1 and LN>1, the odds on H decrease if E is observed
3. LS=LN=1, odds on H are independant of E.

The values for LS and LN must be supplied by geological experts when new rules are added to models in Prospector. They are subjective estimates which may require adjustment in the creation of inference nets.

In order to simplify the probability calculations the designers of Prospector used a piecewise linear function to infer the effect of the modified probability of evidence E, i.e P(E|E'), on the probability of hypothesis H, i.e. P(H|E'), see Appendix C.

This means the individual items of evidence must be assumed to be conditionally independent to allow the incremental update of the probability of hypothesis H. The implications of this independence assumption are discussed later.

Finally, Figure 2.29 shows a simple example of a space in AL/X. The 'prior' entry is the prior weight assigned to the space. The 'logical definition' entry indicates that the degree of belief in space A is a function of spaces B,C and D. The rule antecedents indicate the weightings on inference links to other spaces that provide evidence for this space. A rule consequent entry provides links to other spaces for which this space is evidence.

Figure 2.29: A node in a AL/X inference Net.

```
space A
text description  /* some comment */
inference
prior 0
logical definition
     and (B,C,D)
rules antecedents (I pw 10 nw-5)
rules consequents (J pw 20 nw-2)
```

2.5.4 Prospector from a Logic Point of View

Prospector has two main data structures, namely an associative net describing taxonomies and an inference net to represent production rules. The associative net can be directly represented in first order logic by simple predicates and facts, e.g. class_element(iron-sulphide, domite), class-member(sulphides, iron_sulphide).
Alternatively, forms of the Modus Ponens inference rule can be used rather than enumerate all the class members, eg

sulphide (x) :-iron-sulphide(x) OR.....

along with facts such as iron-sulphide(pyrites).

The inference net representing the production rules is simply a convenient form of an AND/OR graph and, when combined with the fact expressions above, comprises the global database structure. The operation of Prospector is essentially a backward chaining strategy. The propagation of probabilities in forward chaining mode simply serves to provide guidance for the top level control strategy as to which models should be continued to be searched. This is exactly the same style of control provided in the general GRAPHSEARCH algorithm described earlier, section 2.2. There is nothing fundamental, therefore, that would not allow Prospector to be implemented in a theorem proving system, where the inference process is clear, with the proviso that use of inference link weights be used in rule application and selection. Additional labelling of nodes with probabilities and match arcs with rule weights are simple housekeeping operations.

In the introduction to logic based production systems only backward and forward chaining strategies were mentioned. Nilsson(1982) shows that it is perfectly straightforward to describe mixed or combined forward and backward chaining systems in a theorem proving framework. The essential operation is finding unifying compositions between the leaf nodes in goal expression graphs (backward chaining) and the leaf nodes in fact expression graphs (forward chaining). The implementation of any particular control strategy, and the choice of rule application and selection, is an important but arbitrary choice. The essence of the effect of imprecise data in both Mycin and Prospector is to influence or preclude the selection of certain rules. This simply means that the selection of literals, in a resolution based system, must take account of any arguments in well formed formulae representing uncertainty measures. Literal selection algorithms, e.g. SL or LUSH, would have to be modified accordingly.

The introduction of uncertainty measures to the control of search for proofs is identical to the introduction of probability measures in heuristic graph searching. Although this was not discussed in the introduction to graph searching algorithms it is clear that ranges of uncertainty can be associated with heuristic evaluation functions rather than single point values. Calay (1985) is a good discussion on how the B^* algorithm is extended to use probability measures and make use of the upper and lower bounds on heuristic estimates. Similarly, Pearl(1984) describes the use of probabilities in versions of the alpha-beta game playing search methods. The measures of upper and lower bounds of the utility or worth of nodes are essentially used to further prune paths from the search space.

2.5.5 The OPS5 system

OPS5 is an interpreted language specifically written to allow production rules to be easily expressed and efficiently interpreted. The system is implemented in Lisp. Full details can be found in Brownston et al.(1985), but see Forgy (1981) and Forgy and McDermott(1977) for earlier developments. A brief description of the system is now given and then discussed in the context of logic theorem proving systems.

The general form of an OPS5 production rule is

```
(p <identifier>
      <condition>*
   -->
      <action>*)
```

where "identifier" is just a word to identify each rule and the asterisks indicate one or more conditions or actions. Each condition is basically an object-attribute-value triple or vector of attribute-value pairs associated with the object. For example, the temperature and pressure of heat exchanger he101 could be expressed as

```
(p rule1
   (heat exchanger ^name he101
                   ^temperature 298 ^pressure <Pvalue>)
   ...
```

The first entry identifies the object type, each attribute being preceded by "^" and followed by its value. Variables can be used to retrieve attribute values by using angled brackets as with Pvalue above. The variable can then be used in other conditions of the rule. Constraints can be placed on the values of attributes by placing them in curly brackets. For example, suppose a heat exchanger is to be identified with a pressure greater than the value of <Pvalue> above. This would be expressed as

```
   (heat-exchanger ^pressure {> <Pvalue>})
```

Finally if a condition is preceded by a minus sign then the truth value of the condition is negated, i.e. -C is only true if nothing can be found to match C.

Rule consequents in OPS5 are primarily intended to be actions with rules expressing situation-action pairs. There are three main types of rule action:

```
(make <identifier> <^attribute value>*)

(remove n)

(modify n <^attribute value>*)
```

The make instruction adds a new vector of attribute-value pairs to the working memory. The 'n' in modify and remove actions refers to the nth conditions that were matched in the rule antecedent. For example, (remove 1) will remove the vector described in the first rule condition from working memory. The modify action is

similarly used to modify attribute-value pairs in working memory.

OPS5 is an efficient implementation of a forward chaining production rule system. The overall control algorithm follows the classic three phase recognise/act cycle described earlier, i.e. a match phase, followed by conflict resolution, followed by an act phase. The widespread use and success of OPS5 has benefited directly from the efficiency of the algorithm developed for use in conflict resolution. This algorithm, the RETE algorithm, represents the control strategy of OPS5 and is worth discussing in more detail. Many other production systems have subsequently copied or built upon the ideas expressed in the RETE algorithm.

2.5.5.1 Problems in Pattern Matching and the RETE Algorithm

The left hand sides of OPS5 rules are simply pattern templates to be matched against the elements of working memory. In many production systems pattern matching takes place between two collections of lists, namely the production rule antecedents and the contents of working memory. Experience in the development of production rule systems, eg Forgy(1981), shows that simplistic pattern matching algorithms can lead to prohibitively lengthy computation times in deciding which set of production rules can be matched against working memory at any one time. This type of problem is usually referred to as many- pattern/many-object matching. The computational cost can be due to a variety of reasons:

1. A naive implementation may try to match all the rules against each element of working memory on every recognise/act cycle. Even if only selected parts of working memory are matched against each rule, there will still be N iterations per cycle for a system of N rules. For large rule sets and/or a large working memory this will become a very time consuming activity.
2. It is often the case that the same elements of working memory and rule pattern templates will be repeatedly matched on every recognise/act cycle. This is referred to as "between-cycle iteration".
3. Production rule antecedents often share many of their individual conditions, i.e. the same condition appears in more than one rule. A simple implementation would result in a shared condition being matched N times against working memory given N occurrences of it in a rule set. This is referred to as "within-cycle iteration".

The RETE matching algorithm was designed to address both within-cycle and between-cycle iterations. Within-cycle iteration is reduced by compiling production rule antecedents into a sorted network which takes account of both intra-element features and inter-element features. Intra-element features, i.e. values of attributes in a single rule condition, result in a linear sequence of nodes to test for each required attribute. Once intra-element features have been tested, inter-element features are then considered.

Inter-element features are those parts of rule conditions which have shared attribute values due to the use of the same variable in different conditions, e.g. ^Pressure <P> being used in more than one rule condition. Inter-element nodes have two inputs. The first input tests whether shared variables have compatible values while the second tests for the rest of the attributes in the rule condition. A "terminal" node is used to represent the production rule as a whole and a rule will be added to the current conflict set, i.e. the set of rules that could be matched against working memory, if the network can be traversed without failure to a rule's terminal node. Recognising inter-element features extends to the use of similar conditions in different rules. In this case, shared network nodes will have links to terminal nodes for each rule involved.

The output of the RETE algorithm is the current rule conflict set. This is represented by a set of ordered pairs of the form

```
<production, matched left handside elements>
```

for each successful instantiation of the rule variables with working memory. What makes OPS5 reasonably efficient is that the input to the algorithm is not the working memory and compiled network but rather a set of tokens describing changes to working memory and the compiled network. Once a set of production rules have been tested against a set of attribute values represented in working memory there is no need to re-test all of them on each recognise/act cycle. The only ones that need to be tested are those that refer to attributes of objects or class types, the attribute values of which have been modified as a result of executing the actions in the right-hand side of a rule in the previous recognise/act cycle. All that needs to be checked, therefore, is which instantiations in the conflict set are no longer valid and which new rule matches have become valid as a result of any changes.
As a consequence, between-cycle iterations are greatly reduced. Within-cycle iteration can only be reduced to the extent that sharing of variables within a rule, or parts of conditions across more than one rule, occurs in any particular rule set.

Once the conflict set has been generated there still remains, of course, the problem of deciding which rule to fire. There are two control, or conflict resolution, strategies that can be used in OPS5. The default strategy is called LEX and is a variation on the means-end analysis approach, MEA. There are five basic steps in MEA:

1. Eliminate from the conflict set any instantion that fires, i.e. the same instantiation can't fire twice. This prevents one kind of unwanted looping and is referred to as refractoriness. Put simply, it means that a particular rule can't fire twice if the variable bindings are the same.
2. Preferentially select those rules, the first conditions of which match up with the most recently created working memory elements.

3. Order the other rule conditions on the basis of the recency of the matching working memory elements. This is referred to as "recency". The idea is to follow a single line of reasoning for as long as possible and delay the evaluation of alternatives, similar to bottom-up, forward chaining search strategies.
4. If no rule is preferred order the competing rules according to how specific the rule antecedents are, i.e select the most specific rule. The rule will either fail or, if successful, prune the search space by the greatest amount since they take more data into account.
5. If a rule has still not been selected then make an arbitrary selection from the remaining contenders.

(The LEX strategy does not use step 2, step 3 applying to all matching elements of working memory. A more complete discussion of each strategy can be found in Brownston et al(1985)).

2.5.5.2 Partitioning Rule Sets and Context Layering

The means end analysis strategy in OPS5 is directed by giving preference to those rules whose first condition in the antecendent match some part of the working memory. This is an important feature as it is the only mechanism available in OPS5 to allow partitioning of a large set of rules into smaller, related subsets of rules.

Consider, for example, describing the task of designing a heat exchanger in the wider context of a process plant design. It is obviously pointless matching up rules which pertained to the design of other equipment items, such as pumps, columns etc. Rules directly relating to heat exchanger design can be selected by having a common first condition such as

```
(rule 1 (designing_heat_exchanger
        ...)
```

Similarly, rules for the design of heat exchangers can be further partitioned according to the level of design detail that is desired, e.g. rough sizing, type selection, tube sizing, baffle placement etc.

This technique, often referred to as context layering, was put to very good use in the development of the R1 production system, see McDermott(1980, 1984).

R1 is one of the most successful commercially developed production systems. It is used to configure VAX computer systems by first checking that an order is complete and then determines the spatial layout of the computer components. The system contains over three thousand rules describing some five thousand components and has reportedly processed more than eighty thousand orders.

Of interest here is the way in which R1 makes use of contexts to impose a structure on the way it sequences the various problems to be solved. The approach used is exactly analogous to how rules for heat exchanger design can be layered according to

the level of detail that is required. High level rules fire, the actions of which create elements in working memory which match the first condition of a lower level set of rules. Similarly, the action parts of these rules create elements which fire yet lower level sets of rules, and so on. The overall effect is to allow top down, decomposition problem solving strategies to be conveniently expressed in production rule format. The design task that R1 solves is in many ways analogous to specific parts of process plant design.

2.5.6 OPS5 From a Logic Point of View

The OPS5 system is not very interesting from a representational point of view since its syntax is a very restricted form of what can be expressed in first order logic. What is important is the control strategy used in the RETE algorithm.

It is clear from preceding sections that the application of rules in OPS5 could be described in a straightforward way in terms of resolution refutation based procedures. Testing for valid instantiations corresponds directly to the task of finding compatible unifications in a proof tree. The essential difference between OPS5 and standard resolution proof techniques is the mechanism used to decide on which rule to apply. The idea of compiling the production rules into a network is essentially the same as that behind the use of connection graphs in theorem proving but efficiently implemented. Where it differs is how new potential resolvents are identified as a result of addition or deletion of clauses to the global database. There is no reason, however, why a refutation control strategy could not be implemented which more closely followed the RETE algorithm.

Chapter 3
Development of Representation Techniques For Process Design

This chapter describes three prototype systems developed to investigate the use of different representation techniques for use in a process design environment. The areas covered are rule based production systems, blackboard architectures and frame based systems. Each of these are covered in separate sections below.

3.1 Development of a Heat Exchanger Selection System

This section describes the development of a general production system applied to the problem of heat exchanger selection in process design. The purpose of this work was to understand more fully the problems associated with this form of knowledge representation. In particular, the suitability of this style of knowledge representation for use in different aspects of a process design environment, outlined earlier in chapter 1, was considered.

This work represents the first of a sequence of developments in different styles of knowledge representation discussed in the remainder of this thesis.

3.1.1 Background

It is clear from the success of the R1 production system in configuring computer systems, see previous section, that production systems can be used to represent knowledge about certain aspects of design. The initial aim of this work was to develop a rule based production system that could be applied to selected process design problems. The problem chosen for study was the initial selection, sizing and costing of heat exchangers. This problem is typical as regards the initial design of other items of process equipment, e.g. pumps, columns etc. The resulting system, therefore, could in principle be applied to these other design tasks if so desired. The use of context layering, described earlier, also means that the description of individual design tasks can be embedded in rules describing higher level aspects of the design process if appropriate control strategies are provided.

It was decided to develop the production system using the Prolog language, see section 2.3 earlier, for the following reasons:

1. It is simple to develop alternative control strategies on top of the Prolog interpreter. Since much of this thesis is concerned with developing suitable representation mechanisms for use in a process design environment, flexibility is an important issue.
2. The relationship between the search techniques used and logic theorem proving techniques remains clear.

3. OPS5 and other Lisp based systems were not available or supported on the hardware available at the time.

Details of the version of Edinburgh Prolog used are given in Prolog(1986).

The problem domain chosen was the initial selection of type of heat exchanger suitable for a given duty. The initial aim was to represent a basic design procedure in a production system comprising of a set of production rules, a working "database" and associated interpreter. The design steps followed in the selection of heat exchanger type roughly follow those in the I.Chem.E User Guide, see Linnhoff et al(1982). The first step in this procedure involves ruling out types of heat exchanger which are definitely unsuitable and identifying those which seem most promising. This is done on a purely heuristic basis taking into account the following considerations:

1. Maximum pressure - high pressure process conditions immediately rule out the use of certain types of heat exchanger.
2. Temperature Range - as with pressure, many classes of exchanger only operate within a limited temperature range.
3. Fluid Limitations - important limitations in the use of a heat exchanger are the range of fluid viscosity that can be handled and corrosion problems with the particular process fluids and materials of construction used.
4. Normal Size Range - size limitation is a soft constraint in that multiple units can always be installed but this is often undesirable due to additional piping costs, flow distribution problems, etc.

In order to further discriminate between exchanger types it is necessary to perform some rough sizing and costing calculations. Linnhoff et al(1982) provide a number of tables relating heat duty, estimated overall heat transfer coefficients and cost factors (£/(W/K)) to different types of cold and hot side fluids for a number of heat exchanger types. The tables are typical of costing information found in database systems. Given the process conditions and fluids involved, the feasible exchanger types can be ranked according to cost and size.

If a shell and tube type heat exchanger is selected then Linnhoff et al give a further set of heuristics to determine the TEMA type configuration. In practice, this would have to be confirmed by running commercially available heat exchanger design programs to check for the locally optimum configuration.

3.1.2 Description of the Production System

This section describes the structure of a simple, but flexible production system. In particular, it describes how it was applied to the problem of initial selection of a suitable heat exchanger type given the process conditions and working fluids. It is

important to note, however, that the purpose of the exercise was not to develop a detailed heat exchanger design system. The main aim, instead, was to investigate the problems that could arise if such a rule based knowledge representation system was applied to similar design tasks in an integrated process design environment.

3.1.2.1 Production Rule Format

In the introduction to Prolog, section 2.3 earlier, it was indicated that the programmer can specify the syntax of clauses to be read, or "consulted", into the Prolog interpreter. Consider a clause representing a production rule, the rule consisting of an antecedent and a consequent. This could be written in a variety of forms, e.g. in the default Prolog syntax,

```
Consequent :-  Antecedent.
```

or in triple form,

```
rule(Label, Antecedent, Consequent).
```

or in some infix, easy to read format,

```
RuleLabel:  if Antecedent
            then Consequent.
```

The style chosen here is similar to the infix style of the last example and is modelled on a version of the KS-300 rule based system from Teknowledge Inc., see Ross (1985).

The general format of a production rule is

```
RuleLabel: if Antecedent
           then Consequent because Explanation.
```

RuleLabel is simply a word, rule 15 say, used to identify the rule and is referred to for explanation or debugging purposes, see below.

Antecedent can either be a single clause or a conjunctive clause or a disjunctive clause, e.g.

```
if C1 then ..., or
if C1 and C2 ... then ..., or
if C1 or C2 ... then ...
```

Disjunctions and conjunctions can be nested within one another. The individual clauses, C1, C2 etc, are written in infix format and a number of special forms have been provided. The basic form used to refer to the value of some attribute is

```
if THING = VALUE, as in
if problem_type = heat_exchanger_selection
```

Two other useful ways of referring to "values" is to use the "is" and "has" forms i.e.

```
if THING is VALUE, or, if THING has VALUE
```

Negations can be expressed by using the corresponding "is not" and "hasnt" forms as follows,

```
if THING is_not VALUE, or
if THING hasnt VALUE
```

Negation is taken to mean "failure or proof" as is usual in Prolog applications, i.e. not (X) is true if X can't be proved given the current set of clauses in the Prolog database.

One of the useful features of OPS5 is the ability to name variables in a rule condition and refer to them in other rule conditions, typically to ensure some constraint is satisfied. This effect is easy to achieve in a Prolog system. Consider the following fragment of a possible rule:

```
if type_of_pump is Ptype
   and Ptype is one_of(centrifugal, reciprocating)
```

As soon as Ptype is established in the first condition, the unification mechanism in Prolog ensures that all other occurrences of variable Ptype elsewhere in the rule are immediately "instantiated" to the same value. (Instantiation refers to the process of unifying two prolog structures).

Virtually every aspect of process plant design involves calculating numerical information at some point in the process, checking allowable ranges of values etc. Most of the early rule based production systems were not written with numerical applications in mind. Indeed, subsequent discussion will indicate that expressing essentially procedural information in a production rule format is both unnecessary and undesirable. Nevertheless, there is no reason why simple calculations should not be expressed within rule conditions if desired. The rule syntax used here allows both simple numerical tests and arbitrary calculations to be performed in a rule antecedent. The format is:

```
if THING < VALUE or
   THING =< VALUE or
   THING   VALUE or  >
   THING    VALUE or >=
   THING is ^predicate(VALUE,Arg1,Arg2,...)
```

Of particular interest is the last form which allows an arbitrary Prolog predicate to be evaluated in a rule condition. In a procedural context, this could be used to perform some relatively complex calculation as Prolog supports both floating point arithmetic and a simple interface to the C programming language. The unification mechanism in Prolog means that if any of the predicate arguments get unified to some value in the evaluation of the predicate, then these values can be referred to in subsequent rule conditions just as for other variables in the rule antecedent.

3.1.3 Overall structure and Control Strategy

The overall structure and control strategy used in the production system is interesting in that it combines the rule interpreter's own strategy along with the default depth first backtracking search strategy used by the core Prolog interpreter.

The top level structure is, in fact, a set of Prolog clauses, the definitions of which are dynamically loaded depending on the problem context, e.g. heat exchanger selection, pump selection etc. Figure 3.1 shows an abbreviated form of the top two clauses, programming details having been omitted for clarity.

Figure 3.1: Top Level Clauses of a Rule Based Production System

```
select THING:-
    load_definition(THING)
    choose THING

choose THING:-
    ask_for_details,
    (details_available --> get_problem_data;
                           get_vague_details),

    (details_available --> to_associated_calculations;
                           true),
    search THING,
    rank_alternatives THING
    report_answer THING
```

The call to the rule interpreter is the line "search THING" in the definition of "choose THING". The clauses either side of this call are loaded dynamically depending on what is being "selected" at the top level clause, e.g. select heat_exchanger_type etc.

The search strategy for these other clauses therefore relies on the default Prolog interpreter. The same general framework could be applied to a number of different problem domains. Domain specific solution styles can be catered for by loading alternative definitions of the top level calls without affecting the search strategy of the rule interpreter itself. Brief details of the definitions used in the case of selection of types of heat exchanger are given later, section 3.1.3.2.

The rule interpreter itself is basically a backward chaining system whose rule syntax was described earlier. Figure 3.2 outlines the search strategy used:

Figure 3.2: Search Strategy for Heat Exchanger Selection Rules

```
1)   If already searched for THING then return.

2)   If the domain knowledge base indicates that
     the user can be questioned regarding
     the value of THING then ask user

     2a) If user replies with "why, help, describe"
         or issues another Prolog query
         then do as requested and repeat from 1)

     2b) Otherwise, record user's answer and return.

3)   Find a suitable rule referring to THING
     in the rule consequent.

     3a) For each condition in the rule antecedent
         referring to ThingN repeat from 1).

     3b) Repeat from 3) for THING

4)   If all rules referring to THING have been considered
     then return.
```

There are several points to note about this search strategy :

1. Step 2b) is similar to the approach of Mycin, i.e. if the user can offer an answer then accept this rather than perform further search. If subsequent search results in failure then the algorithm can backtrack to this point, delete the user's answer and search for itself if so desired.
2. The rule conditions in step 3a) are evaluated in left to right order. It is simple, therefore, to layer rules, i.e. make them context dependent, by referring to problem context details in the leftmost rule conditions. If these conditions evaluate to false then the rule will fail and cause another rule to be selected.
3. The rule syntax allows arbitrary prolog predicates to be evaluated. This facility can simply be used for procedural calculations or can be used to implement various forms of procedural attachment, semantic checking etc., see sections 2.3 and 2.4. In particular, complete proofs can be embedded within the rule interpreter's search strategy as control is returned to the Prolog interpreter during these embedded calls.

The overall structure of the production system is a general set of clauses used to establish some top level goal. Search at this level is driven by the Prolog system, i.e. a variant of LUSH resolution. The control strategy is switched to that shown in Figure 3.2 when domain specific rules are used to guide the search. Interpretation of these rules, however, allows Prolog type proofs to be performed when evaluating the truth conditions of individual rule premises. Alternatively, procedural calculations can be evaluated as part of a rule premise and this information can also be used to help guide the search. Context layering within the rule set is achieved by using the first, or leftmost, rule conditions to refer to the context in which the rule is applicable. Contexts are created by simply adding the appropriate clause to working memory in the last condition of the last rule in the higher level set of rules. This ensures that all rules at a higher level context get fired before rules at a lower level are considered.

The system as developed uses the internal database of Prolog to store the clauses which represent the working memory. No attempt was made to develop a specialised structure such as the context tree of Mycin, or the inference nets of Prospector.

3.1.3.1 Simple Help and Explanation Facilities

In advertising literature much has been made of the ability of production systems, or "expert system shells", to explain their reasoning in terms of answers to "why?", or "how?", questions from the user. These claims, in fact, are greatly overstated. As shown in step 2a of Figure 3.2, the user can ask why a question is being asked at any point in the session. The heat exchanger selection system simply responds, as do most other systems, that it is to help establish the truth value of the antecedent of the current rule, as identified here by the RuleLabel. The user can ask the system to "describe" the rule, in which case the rule is printed out, or he/she can ask why this rule is being investigated. The system will reply that it is being used to establish the truth value of some premise in a "parent" rule.

This parent rule, and any of its ancestors, can be displayed in turn. What is being displayed, therefore, is simply the stack of rules invoked by the interpreter. While this may be helpful to a user interested in tracing or debugging the system, it can scarcely be considered an explanation of what the real expert originally had in mind when constructing the rule set. This is evidently so when programming tricks such as context layering of rules, procedural attachment etc., are used to implement a more efficient search. The system has no idea of the relative status of individual premises in a rule antecedent and why they have been ordered in a certain way. It is precisely this implicit information, however, that would be given by a human expert in explaining his solution to a problem.

Upon selection of a type of heat exchanger the user can ask "how" a certain conclusion was arrived at. Once again, all the system can do is retrace the search path in terms of the rules invoked in arriving at any given conclusion.

If the user asks for help in response to a question the system will display any help text relevant to that question that has been provided by the programmer. Alternatively, if the user is a competent Prolog user he may trace any Prolog command to establish some item of information before replying to the question. This facility could be used to retract a previous answer and cause the interpreter to fail and backtrack to a previous question or choice point.

3.1.3.2 Details Specific to the Heat Exchanger Selection Program

This section outlines the specific definitions used in the heat exchanger selection system to define the top level clauses of the production system shown in Figure 3.1 above. These clause definitions are dynamically loaded replacing any previous definitions in use. This means that the same system can be specialised to behave in different ways for different problem contexts without changing the overall structure or rule syntax of the system.

The first clause in the definition of "choose THING" is called "ask_for_details". In this case the system asks the user whether a point in design has been reached where full details of the hot and cold streams of the exchanger are known, i.e. heat loads, physical properties etc., or whether only vague problem details are known, e.g. the types of fluid involved, maximum operating temperatures and pressures that are to be expected. If full details of the hot and cold streams are available then the system will subsequently be able to perform approximate sizing and costing calculations and use this information when recommending a type of heat exchanger. If this data is not available then only a rough guide to the types of heat exchanger to consider will be possible.

In either case, the user must at least classify the type of fluid in use in both the hot and cold streams. On the cold side the categories used are low pressure gas, high pressure gas, treated cooling water, low viscosity hydrocarbon, high viscosity hydrocarbon, boiling hydrocarbon and boiling water.

The hot side fluid is classified as one of low pressure gas, high pressure gas, process water, low viscosity organic liquid, high viscosity liquid, condensing steam, condensing hydrocarbon and condensing hydrocarbon with inerts. The categories used are by no means exhaustive but are ones for which cost data is available, Linnhoff et al (1982).

If details of each stream are available then the basic heat balance calculations are performed.

Once the initial problem context has been established the rule interpreter is invoked to identify the most likely type of heat exchanger to use. This involves an exhaustive backward chaining search over the set of production rules, see Figure 3.2 above. The types of heat exchanger referred to in the production rules are shell and tube, double pipe, gasketed plate, spiral, lamella and air cooled. Figure 3.3 shows some typical rules.

Figure 3.3: Example Production Rules for Heat Exchanger Selection

```
rule1 : if max_pressure < 307 and max_temp < 575 and min_temp > -200
        then heat_exchanger = 'shell&tube' cf 900 because
        [' a popular choice, well tried and tested,
           and if Pressure and Temperature are
           acceptable then you''re only really limited
           by materials of construction '].

rule2 : if rule1 hasnt worked and
        max_pressure < 307 and max_temp > 575 and min_temp > -200
        then heat_exchanger = 'shell&tube' cf 800 because
        [' although Pressure is acceptable,
           you''re above the safe operating
           temperature for ordinary steels so you''ll
           need special costly materials '].

rule5 : if max_temp < 600 and min_temp > -100
        then heat_exchanger = 'double_pipe_bare' cf 800
          and heat_exchanger = 'double_pipe_finned' cf 750 because
        [' usually more expensive than other types
           but come in standard modular design,
           and can be very attractive at low Reynolds number '].

rule12 : if use = heat_rejection_system
         and allowable_type = 'shell&tube'
         and hot_fluid = pw
         then heat_exchanger = 'ctst' cf 800 because
         [' often a fairly cheap acceptable alternative
            in a heat rejection system ,
            as opposed to site-cooled cooling water '].
```

There are several points to note about the form of the rules not yet explained:

1. In rule 1 the rule consequent is of the form
   Conclusion cf Score because text
   The confidence factor, or score, is a subjective value, in the range 0-1000, placed on the relative degree of confidence in selecting a particular type of heat exchanger. Text can be used to further describe the rule and is used for explanation purposes only.
2. Rule 5 shows the use of a more complex conclusion.
3. The condition "use = heat rejection system" in rule 12 is an example of context layering, i.e. partitioning the rules into identifiable sub-problems.

The use of a confidence factor in a rule consequent is simply a heuristic means of representing some degree of uncertainty in the conclusions of the rule. Confidence factors are combined in a straightforward way. The assumption made, as in Mycin, is that each item of evidence can be treated independently as follows:

1. If a rule consequent consists of a proposition which itself has a confidence factor, then the overall confidence factor is obtained by multiplying the values together and dividing by 1000.
2. Given an existing proposition P of confidence factor C1 and a new assertion that P has confidence factor C2, the resulting confidence in P is

$$C3 = C1 + C2 - (C1*C2/1000)$$

This is obviously a very simple weighting function and is subject to the same

criticisms made regarding the treatment of uncertainty in MYCIN and PROSPECTOR earlier. In particular, it does not account for "negative" evidence as the expression in 2) means that C3 >=C1. A more sophisticated technique would probably be required if the heat exchanger problem was analysed in greater detail as many interactions between rules might arise. The function used, however, is adequate for demonstrating how uncertainty can be represented in rules and the problems that arise. The discussion of how uncertainty might be handled in a design environment is delayed until section 4.3.

The final aspect of the heat exchanger selection system to be considered is how the alternatives are ranked once an initial selection has been made. There are two possible cases:

1. If details of each stream are available then rough costing calculations are performed. Both costs and confidence factors are then presented to the user.
2. Otherwise, the alternative types are ranked purely on the values of the confidence factors.

The cost data is taken from Linnhoff et al(1982). For each type of heat exchanger and combination of hot and cold side fluids, values of the cost factor C are given for specified intervals of $Q/\Delta T_{LM}$ i.e. heat transfer per temperature difference (W/K), and the value of U (W/m^2K) used. The units of C are (£/(W/K)).

The table of cost data for each exchanger type is represented as a set of clauses in Prolog's database. Figure 3.4 shows some typical entries for spiral heat exchangers.

Figure 3.4: Sample Clauses used to Represent Cost Data

```
cost_data(shell_and_tube,1000,lpg,condh+inert,86,3.14,single_shell).
cost_data(shell_and_tube,1000,hpg,lpg,93,3.76,single_shell).
cost_data(shell_and_tube,1000,hpg,hpg,300,1.70,single_shell).
```

Estimating the cost involves searching this database to locate the appropriate fluid categories (lpg - low pressure gas, hpg - high pressure gas, etc.) and the values of $Q/\Delta T$ (e.g. 1000 in Fig. 3.4) which form the closest interval around the value of $Q/\Delta T$ calculated previously. Once a suitable interval has been found, logarithmic interpolation is used to calculate a value for the cost factor. When multiplied by the actual $Q/\Delta T$, an estimate for the cost of the exchanger is obtained, see Linnhoff et al for details of worked examples.

3.1.4 Discussion

A general production system was written and applied to a typical design task, namely the initial selection of a type of heat exchanger. The production rule interpreter is embedded within a general framework which can be specialised in different ways in

different problem domains. The framework allows procedural aspects of design to be introduced in a convenient manner, both prior and subsequent to any search performed by the rule interpreter. This is a common feature of many design procedures in which

1. Initial calculations can be performed to provide guidance or bounds for the subsequent selection/search phase.
2. Further calculations are often performed to verify a particular decision but which are too costly to perform for every option available.

The rule syntax used is both flexible and easy to understand. Two features of the Prolog interpreter are exploited which greatly enhance the degree of complexity of "data" that can be represented in the rules:

1. Prolog variables can be used in rules and instantiated to arbitrary complex Prolog structures stored in the internal database or working memory.
2. Calls to Prolog predicates can be made within rule conditions. This means that either Prolog like proofs or procedural calls can be nested within the rule interpreter's control strategy.

These features can be used to reduce the amount of search that would otherwise occur using the default, backward chaining control strategy. Reducing the amount of search is also one of the reasons for allowing initial "calculations" or search to be done before calling the rule interpreter. This approach attempts to overcome the drawback of backward chaining systems in that they do not make full use of any problem information at an early point in the search, unlike forward chaining systems which are data driven. The approach works if assertions are made which are referred to in the first or leftmost conditions of a rule antecedent since these are interpreted first in this system.

The demonstration application also showed the ease in which database facilities can be accessed. In this case heat exchanger cost data was stored in predicate form in the internal database available in Prolog. Previous discussion, section 2.3.6, indicated that access to external mass storage databases can be achieved in much the same way.

The development of the rule based production system was the first of four related investigations into the suitability of different knowledge representation formalisms for use in a process design environment. A discussion, therefore, of the relevance of rule based representations to process design activities is delayed until these other representation styles have been described, section 3.5.

The rule based approach was initially studied since it appeared to be the most successfully used and well understood of the AI programming styles available at that

time. Given the insight gained in the development of the heat exchanger selection system, in addition to increased awareness of other relevant research in the area of AI, it was realised that there were many problems associated with the direct application of rule based systems in process design. Two general classes of problem were identified:

1. Practical Problems - the problem of constructing and maintaining large, consistent rule sets in a complex, highly inter-related domain such as process design would be extremely difficult.
2. Theoretical Problems - how could rule based systems be used to tackle the problem of maintaining integrity of knowledge that was identified earlier, chapter 1, as being fundamental to any advanced design environment with capabilities beyond those found in current database systems?

The potential problems identified in using a simple production system, such as the heat exchanger selection system described above, are introduced in the next section.

The underlying question in point 2) above is one that fundamentally challenges any representation system that is claimed to be "expert" or show "expertise" in any problem domain, namely to what degree can any system be said to "understand" its problem domain in a meaningful and useful way?

This question is not discussed, however, until other research done on "Blackboard" and "Frame Based" representation systems has been presented and, at which point, useful comparisons between styles can be made.

3.2 Problems concerning the Use of Rule Based Production Systems

This section describes some of the basic problems associated with the use of rule based systems, assuming that suitable design applications can be found. The question of the relevance of production systems in a design environment and to which aspects of the design process they can be most usefully applied is answered in section 3.5.

3.2.1 Implementing Control Strategies

Considerable explanation was given earlier in this chapter regarding the workings of logic based theorem proving systems and their correspondence to both forward and backward chaining production systems. It has already been stated that the major problem facing general theorem provers is development of good control strategies which can select and apply inference rules for efficient solution of problems. Experience indicates that information specific to both the problem domain and the particular problem context must be used if this goal is to be achieved. Generalised approaches, such as the resolution control strategies described in section 2.3, only partially reduce the amount of search required and are ineffective on complex problems.

Given the close correspondence between theorem proving systems and production systems, as described so far, the obvious question to be asked is whether production systems can be used to implement more efficient search control strategies.

Consider again the OPS5 search strategy, see section 2.5. The first stage in this strategy is to compile the set of rules in use into a network of nodes, representing individual rule conditions, that can be efficiently searched when deciding which rules can be selected on each cycle of the recognise act loop. This provides no information, however, on which rule out of the set of applicable rules to invoke. This is exactly the same situation that arises in the use of either the connection graph approach or the matrix connection method in resolution based refutation systems. Indeed, the compilation methods of the latter two techniques are more complex in that alternative solution paths are also (implicitly) identified at this stage.

The selection of a particular rule from the set of all possible rules in OPS5 is then determined by the domain independent RETE algorithm. The rule selected depends on recency of bindings, rule complexity etc. The selection decision is slightly more complex than, but identical in principle to control strategies found in resolution systems. The only way for the programmer to help guide the search and make good use of problem specific information is to use the programming trick of asserting propositions referred to in the first condition of other rules. These rules will then be

preferentially selected provided the programmer uses the means-end analysis, MEA, version of RETE. This "trick" could easily be implemented in a resolution system if additional management and recording of changes to the global database were made.

Two approaches that have been widely used to improve the use of domain dependent control knowledge during search are now described. The first approach involves embedding control information in the rules themselves and can be thought of as an extension of simple procedural attachment,see section 2.3.4 The second approach involves the use of meta-level information, i.e. information about information such as rules describing the use of other rules. The ideas presented below are taken from a number of sources, rather than any one prototype system or language, and are described in common logic-like terms.

3.2.2 Embedding Control Information in Rules

The discussion of theorem proving and production systems has centred, so far, on two types of domain knowledge: declarative knowledge and procedural knowledge. Declarative knowledge is a term loosely used to refer to declarative type statements such as facts and inference rules that are used to describe a problem area. Procedural knowledge is usually taken to mean knowledge of "how" things are achieved, e.g. how to perform certain calculations, etc, and is often identified with the procedures that perform these calculations. Both types of information can usually be handled in up-to-date systems. The heat exchanger selection system combines both declarative and procedural knowledge in a particularly flexible way.

It has been recognised for some time, however, that the domain specific control information on how to combine the use of declarative and procedural knowledge in solving problems must be represented and manipulated in some way in order to improve the performance of theorem provers or production systems.

Winograd (1972) is an important example of criticisms made concerning the types of problems not addressed by classical theorem proving systems available at that time.

A number of systems were reported in the period 1970-1980 which attempted to allow control information to be specified in much the same way as declarative knowledge. In particular, the PLANNER system, Hewitt (1972), and the AMORD system, de Kleer et al (1977) were important developments. Other important contributions, such as CONNIVER [McDermott and Sussmann (1972)] and QLISP [Rebok et al 1976] are reviewed in Barr and Feigenbaum (1980). Moore (1975), however, is a good discussion of the logical inadequacies of many of these systems.

The essential idea used in each system was to allow control information to be specified in either forward or backward chaining rules. In order to retain the forward or backward chaining character of the production system the control information is typically specific, or local, to each fact or rule. In this case, the global control strategy of a production system operates as before. When a rule is invoked, global control is temporarily suspended and localised search takes place according to the additional control information supplied. Once this has been completed the global search strategy regains control. If the local control checks fail for some reason, the system can backtrack to a higher level choice point depending on the overall search strategy in use.

The most common style of checking is simply associating statements of the form.

if <condition> Fail

with a rule, where <condition> is an arbitrary clause or procedure that evaluates to true or false.

More complicated side effects can be achieved when a rule is fired by asserting new facts into the global database or requesting that additional sub-goals be established first. Each "assert" or "sub-goal" command can list a number of rules that should be tried first during any subsequent search, i.e. locally overriding any global rule selection criteria. The sub-goal requests may simply involve searching for current instances of predicate forms in the global database. In the PLANNER system potentially useful theorems can be individually named or general "classes" of theorems can be recommended.

A more sophisticated feature of some systems, e.g. PLANNER, is the ability to recommend whether an inference rule should be used in a forward or backward chaining mode. For example, consider the rules:

$$\text{Heat_exchanger}(x) \Rightarrow \text{unit_operation}(x)$$

$$\text{Stripping_column}(x) \Rightarrow \text{unit_operation}(x)$$

If there were lots of rules of this type it would be more efficient to use them in a forward chaining style in attempting to prove that a particular item was a unit_operation rather than try and establish that it was a heat_exchanger, stripping_column, etc, in turn. By using rules in a direction that minimises the number of alternatives the amount of search will usually be reduced.

Another feature is to recommend the form of the rule to use. For example, to prove that x is not a stripping_column then it is more efficient to establish first that x is not a unit operation. In a backward chaining system the contrapositive form of the rule.

$$\neg\, \text{unit_operation}(x) \Rightarrow \neg\, \text{stripping_column}(x)$$

should be used.

Finally, most systems also allow the order in which sub-goals in a conjunction or disjunction are to be tackled to be stated, e.g. in the rule.

$$P_1 \wedge P_2 \wedge ... \wedge P_N \Rightarrow C ,$$

the order in which P1 to PN are to be proved could be left to right.

The AMORD system introduced two important features not found in earlier systems:

1. The rule definition language allows the dependence of a conclusion on other beliefs to be made explicit using a "SHOW" facility, see de Kleer et al (1977) for details. The important point is that this controls the particular forms that can be generated by application of an inference rule. For example, given facts A and B the consequent introduction rule introduces $A \wedge B$.
2. This does not prevent redundant conjuncts such as $A \wedge A$ being introduced. Such problems are easily avoided in AMORD.
3. New facts are asserted into working memory using the form
   (ASSERT <Statement> <Justification>)
   where <justification> is a specification of the reason for belief in the fact represented by <statement>.

The justification in point 2) consists of the name of the rule used and the "factnames" that index the assertions on which the belief depends, as made explicit in point 1). The use of explicit justifications for each assertion in working memory allows a more sophisticated form of search to be performed, namely dependency-directed backtracking. Dependency-directed backtracking, as introduced by Stallman and Sussman (1976), makes use of lists of justifications to avoid unnecessary backtracking. For instance, if an assumed fact is contradicted during a proof and removed by backtracking, then the negation of the fact can be assumed and supported by the justification underlying the contradiction. The development of this technique and its use in "truth maintenance systems" is described in section 4.3. Pereira and Porto (1980) describe a general, selective backtracking strategy for use in any Horn clause program written in Prolog. The latter technique maintains contexts of bindings used in unification rather than justification lists. In both methods, the set of possible goals selected as backtrack goals, i.e. choice points, are re- evaluated whenever a sub-goal fails, taking into account the reasons for the failure of the sub-goal.

In summary, a number of techniques have been proposed that allow control information to be embedded or associated with both declarative and procedural knowledge in production systems. Most of the methods have been successfully applied in simple example problems but there is, as yet, no general consensus on their suitability for use in large scale applications. Theoretical AI research has concentrated on redescribing these techniques in a formal logic framework, see Hayes (1973), Moore (1975), McDermott (1978), Levesque and Brachman (1985).

3.2.3 Using Meta-Level Information

As an alternative to embedding control information directly in rules, the control strategy problem can itself be considered to be a problem to be solved by another system. In this case an "object-level" system would retain the original declarative and procedural knowledge describing the problem domain while a "meta-level" system would reason with declarative and procedural knowledge relevant to the control of the object-level system. The term "meta" is used here in a very general sense, i.e. information about information.

Two forms of meta-level information have already been described in the previous section regarding the PLANNER and AMORD systems, namely recommending sets of rules to try in specific situations and explicitly defining the dependency of conclusions on other beliefs.

The use of meta-level information is best exemplified by development of the MYCIN system into EMYCIN and NEOMYCIN, see Clancey and Letsinger(1981) and Clancey(1983). The features developed have been copied in many other systems.

In order to improve the performance of the search strategy in the original MYCIN system a "focussing" mechanism was added to the program. This used domain specific information to prune the set of possible rules applicable at any point down to a smaller set of, hopefully more relevant, rules to focus on. The information was described in the form of meta-rules i.e. rules on how to use other rules. Full details and examples of such rules are given in Buchanan and Shortliffe (1984). The meta-rules were somewhat ad hoc in nature and described different types of control information. For example, a number of meta-rules stated which types of rules should be subsequently used or ruled out given some intermediate conclusion. A rule type is often identified by those referring to specific attributes in the antecedent or consequent.

Other meta-rules were used to modify the order in which certain classes of rule should be applied. Finally, some very general, domain independent rules were used, e.g.

```
If there are rules, R1, which do not mention
        the current goal in their premise,
    and

    there are rules, R2, which do mention
         the current goal in their premise
    THEN investigate R1 before R2.
```

These facilities are similar in nature to those described in PLANNER above and were made accessible in the EMYCIN system. (EMYCIN, as described earlier, is a

domain independent system based on MYCIN).

Careful consideration of the problems experienced in the use and development of MYCIN led Clancey and Letsinger (1981) to reimplement MYCIN in a more general framework called NEOMYCIN. The motivation for this work is well described in Clancey (1983).

NEOMYCIN attempts to make explicit the links between factual knowledge and reasoning or control knowledge in a relatively domain independent way. The static information expressed in MYCIN's property lists was re-expressed as a disease taxonomy in NEOMYCIN. This was implemented as a "frame based" hierarchy of diseases (frame based systems are discussed in section 3.4). Nodes in the hierarchy stored information concerning the extent of disease, symptoms etc. Information at each node representing a disease is stored in so called "slots", each disease being described by a number of such slots.

The control strategy in NEOMYCIN is governed by the "reasoning" rules. These rules are themselves grouped into a hierarchy of nodes:

```
make_diagnosis                -> identify-problem,
                                 establish_hypothesis_space
                                 process_hard_data

identify_problem              -> get_initial_data,
                                 find_chief_complaint
establish_hypothesis_space    -> review_hypotheses_list,
                                 group_and_differentiate
establish_hypothesis_space    -> explore_and_refine
...etc...
```

An example of a reasoning rule associated with the "group_and_differentiate" node is

```
If   there are two hypotheses that differ in some feature
then ask a question that differentiates between them.
```

There are two sorts of "factual" rules in the system, namely trigger rules and data-hypotheses rules. Trigger rules are used to set up new hypotheses if certain conditions are found to be true, e.g. "if stiff neck and ... suggest meningitis".

Rules of this type were described in the previous section on embedded control information in rules. Data hypothesis rules simply assert associations between data and hypotheses but cannot create new hypotheses.

The system works by processing the reasoning rule hierarchy. This results in different sets of factual rules being invoked, trigger rules being responsible for activating intermediate, domain specific hypotheses.

Apart from making the reasoning strategy much more apparent to both the program developer and user, the structure of NEOMYCIN also enabled better explanation facilities to be written. For example, a "How" question could be answered by stating the completed meta-rules used in arriving at a particular conclusion. What is reported

is a summarised explanation of the overall strategy rather than a recursive trace of every rule that was invoked in the search.

3.2.4 Control and Meta Information in the Heat Exchanger Selection System

Despite its apparent simplicity, a number of features regarding control knowledge outlined in the previous two sections can be handled in the heat exchanger selection system:

1. The order in which conditions in the rule antecedent are evaluated is specified by the layout of any individual rule.
2. "If <conditioning> Fail" and "Match <pattern>" instructions can be directly embedded into rules. These are special cases of procedural attachment. The heat exchanger system allows any sort of attached procedure to be evaluated, including those involving nested proofs within Prolog.
3. Facts can be asserted into or retracted from the working memory whenever necessary.

These features would allow considerable control information to be embedded into any working set of rules. The system does not provide, however, an explicit way to recommend sets of rules to use via the use of meta-rules. This would be straightforward to implement, meta-rules being written in the same syntax as before, but the simplicity of the application did not warrant this addition. Individual rules can be referred to by name, to avoid duplicating effort or writing overly complex rules, e.g.

if rule1 hasn't worked and ...then ...

This feature assumes that rule1, say, has already been tested. This is guaranteed if the rules, R1 say, that refer to other rules, R2, are loaded into the Prolog interpreter after the R2 rules. In addition, both R1 and R2 rules must refer to the same entity in the rule consequent. This works because depth first, exhaustive backtracking is used to select each rule referring to the current goal in turn.

The overall structure of NEOMYCIN is simple but attractive. Working with a generic reasoning framework is mimicked in a simpler way in the heat exchanger system. The approach used was to provide a single framework for use, see Figure 3.1 earlier, but allow the specific definitions used to be dynamically loaded in "procedural" form rather than declarative meta-rules. Different "procedures" can be loaded in different problem domains thus achieving different behaviours or reasoning strategies. The procedures can, of course, be Prolog clauses, i.e. Horn clauses, that can be used to express the meta information used in NEOMYCIN like reasoning rules.

Unfortunately, this "trick" relies on the ability of the programmer and his knowledge

of Prolog.

At present, a rule is simply selected by analysing the content of a rule consequent and checking whether it refers in some way to the current goal. The only selection criteria used is that non-recursive rules fire before recursive rules. This checking makes excessive use of Prolog's unification algorithm and would be prohibitively slow with large rule sets. Ross(1987) is an example of a Prolog rule interpreter that compiles rule sets into an internal form that can be searched much more efficiently. Such an addition would be necessary in a more complicated application of the system.

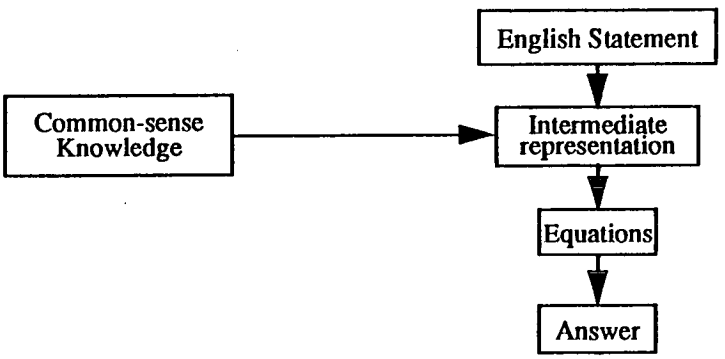
3.2.5 Meta-Level Inference in the MECHO System

MECHO is a sophisticated Prolog program used for the formation and solution of mathematical models from informal specifications in the area of Newtonian mechanics. The system is detailed in Bundy et al (1979a, 1979b). The system is briefly described here for the following reasons:

1. Meta-level inference is implemented in a well defined, logical framework unlike many other AI systems.
2. The formation of mathematical models and their solution is very important in process design problems.
3. The overall structure, approach and complexity of problem tackled by MECHO influenced other developments in this thesis.

The overall problem solving strategy of MECHO is shown in Figure 3.5 below.

Figure 3.5: Problem Solving Strategy used in MECHO



Translation from a statement of a problem in English into an intermediate representation is not of immediate concern here. CHAT-80, see Pereira(1983) is an example of one such system written in Prolog and used to interface to a geographical database system.

The intermediate representation is a set of predicates. The predicates represent the objects involved and the relevant relationships. Consider a pulley system consisting of four objects: two particles, a string and a pulley. Object types are defined with an "isa" predicate, e.g.

```
isa(p1, particle)
isa(p2, particle)
isa(s1, string)
isa(p3, pulley)
```

The relationship defining the configuration of the pulley system might be defined as

pulley-sys(p3,s1,p1,p2).

Various other predicates, such as end-point, contact, incline, concavity etc, are used to further define the pulley system. A number of predicates are also used to further define each object. For example, an object has a unique mass but this may be measured in various units. The same is true for other physical quantities associated with each object. These are represented as follows:

```
mass(p1, mass1)                    accel(p2, acc, 90)
measure(m1, kg, mass1)             measure(a, ft/sec ^2, acc)
```

Thus, a problem statement is converted to an appropriate intermediate form. The system uses a "given" predicate to state which quantities are initially known e.g. given(mass1), and a "sought" predicate to record the quantities to be solved for, e.g. sought(acc). A number of inference rules are used to infer additional information from the problem information, e.g.

```
accel(PartB, Acc, DirB) ∧
pulley_sys(Pull, String, PartA, partB) ∧
extensibility(String, 0) ∧
end (String, EndA, left) ∧ incline(String, EndA, DirA)
→ accel(Part A, Acc, Dir A)
```

This rule states that the acceleration, Acc, of one particle, PartA, in a pulley system is the same magnitude as that of PartB provided the string is inextensible. It can be used in a forward mode if information about PartB is available.

MECHO also uses special axioms to provide default information. The axiom
pulley_sys(Pull, String, P1, P2) → extensibility(String, 0)
states that strings are inextensible by default. These axioms are specially handled since they must not be used if a contradictory conclusion is already known. (The PLANNER system also uses axioms of this type).

The main problem for MECHO is to form mathematical models, i.e. sets of equations, from this intermediate representation. It does this by use of the meta-predicate "is_formula". This predicate relates variables in a formula to allowable propositions in the intermediate form.

Consider the formula F = MA, i.e. Newton's resolution of forces formula. This is represented as.

```
is_formula(resolution_of_forces, situation(Object, Dirctn),
           F = M*A,
           mass (Object,M) ∧ accel_compt(Object,A,Dirctn)
           ∧ sum_forces(Object, F, Dirctn)).
```

The first parameter is just an identifying name, the second describes the problem situation, the third is the formula itself, and the fourth relates the second to the third by a number of propositions to be satisfied. In satisfying these propositions MECHO may have to use inference rules like the earlier example or

```
accel(Object,Acc,Dir1) ∧ A = Acc*cos(Dir-Dir1)
-> accel_compt (Object, A, Dir)
```

to evaluate the component of acceleration in direction Dir.

MECHO still has the problem of knowing which equations to generate for any particular problem. The algorithm used is quite simple and a variant of means end analysis. The basis principle is to try and choose equations which solve for the "sought" quantities in terms of the "given" ones while minimising the number of unknowns introduced. The algorithm steps are

1. Determine what sort of quantity is being solved for and the problem situation.
2. On the basis of 1), select a formula and situation.
3. Fill in any unknowns using inference if necessary.
4. In 3), if any inferences would introduce a new unknown then mark this node as a continuation point and reselect in step 2).
5. If an unknown must be introduced then resurrect a continuation node in 4) and note these unknowns as being sought quantities.
6. Check the equation is independent from earlier ones.
7. Repeat the whole process for any remaining sought quantities.
8. Use any measure propositions to translate variables into numbers in compatible units.

The output of this algorithm is a list of equations. These equations are then passed to the symbolic equation solving package PRESS for solution. PRESS itself is a logic based inference system also written in Prolog, see Bundy and Sterling (1981).

The algorithm as it stands is not very powerful as the large number of choices in steps 2) and 3) would result in large amounts of search. Further meta-predicates are introduced to control the search. The predicate "kind", for example, relates a particular physical quantity to its type and defining proposition:

kind(acc, acceleration, accel(p2, acc, 90))

The meta-predicate "relates" lists physical formulae involving a type of quantity in much the same way as meta-rules identify other rules to use in production systems, e.g.

relates(acceleration, <resolution_of_forces, const_accel1,...>)

indicates acceleration is used in the list of formulae in brackets.

A powerful meta-technique, not found in other system, is exploiting the fact that some parameters in a predicate are functionally dependent on others. Bundy (1983) gives the simple example of mass(part1, m1). Since every particle has exactly one mass, the parameter m1 is functionally dependent on parameter part1 and, hence, has the properties of uniqueness and existence in the logical sense. This can be exploited in three ways:

1. If mass(part 1, m1), say, is already known then there is no point in every trying to prove mass(part 1, m2), m2 ≠ m1. Any goal requesting this can be marked as unsatisfiable. This is called the uniqueness test.
2. Once a goal mass(part1, m) is satisfied by {m1/m}, say, then all other choice points can be discarded. This is termed "back-up prevention".
3. If the goal mass(part1, m) should ever fail, MECHO can create a skolem constant, m say, if necessary. This depends on the state of the problem solving process, e.g. if some unknown must be introduced for solution to proceed, and is termed "controlled creation".

Finally, there are high level meta-predicates which select the most suitable inference method, the "method" predicate, and specify the conditions needed to solve for some quantity, the "solves_for" predicate. These predicates are effectively high level strategies that can be applied at any point in the computation. This is in direct contrast to the "chunking" approach in other systems in which candidate rule sets are identified in advance. The meta-rules in MECHO are highly flexible and complete control can be placed on storage and retrieval of knowledge. The disadvantage is the cost of the extra theorem proving that must be performed.

The importance of the MECHO system to this thesis is that it successfully tackled a class of problem, similar in nature but simpler in content to certain aspects of process design. There is much to be gained, therefore, from an appreciation of the techniques it uses. MECHO is also interesting in that it is one of the few AI systems to be implemented in a fairly rigorous logical framework.

3.2.6 Maintaining and Extending Rule Sets

A potential problem arising from the use of production systems, with important consequences in an industrial context, is how to ensure that rule sets can be extended or modified whilst maintaining a degree of consistency with previous behaviour of the system.

New rules or knowledge will inevitably be added for a variety of reasons, eg:

1. Rules introduced to treat a wider class of "data" or situations.

2. Rules introducing new subtasks to make the analysis of the main task more precise. In complex problem areas it will be impossible to define all the relevant rules in advance.

It is also worth noting that the task of extending rule sets is unlikely to be ever completed in a system that claims to have a high degree of "expertise". Real experts are constantly faced with new research work and information which must be assimilated and understood in the context of previous information if the expert is to remain an expert in that field.

Unfortunately, there are no techniques currently available that can be used to guarantee that previous, correct behaviours will be guaranteed when new rules are introduced into the system or previous rules modified. There are a number of aspects to this problem that are worth outlining.

McDermott(1984) is a good account of the experience gained in developing the R1 System, see section 2.5 earlier. R1 is a large system consisting, at that time, of some three thousand rules. It was found that program development, especially by those not involved in the original development, often resulted in redundant rules and ad hoc solutions. This problem seems inevitable when dealing with large rules sets. The situation is exacerbated by a number of other contributing factors:

1. Embedding control information into rules often makes it much more difficult to understand the original intention of the programmer. Many systems have a complex rule syntax that requires considerable understanding if the intended effect of the control information is not to be misinterpreted.
2. The use of procedural attachment for control purposes can lead to problems if a specific order of evaluation of conditions in the rule antecedent was intended and subsequently modified.
3. The trick of expressing essentially procedural information by relying on the rule firing order is also highly problematical.

Problems such as 1) to 3) make maintenance of rule sets a non-trivial problem in large or complex problem areas. The whole idea of expressing calculations and procedural information in a rule based format is, indeed, a questionable one due to the implicit ordering information used to structure the rule base.

A problem specific to systems, such as Prospector, using subjective weightings on links in inference nets is how to modify these weightings to take account of new nodes in the net. The literature suggests that considerable tuning of these weightings is required to achieve the desired effects. This is true, however, to a certain extent in any system making use of some uncertainty measure in rule consequents.

The basic cause of these problems is that although productions systems were originally intended to make knowledge or expertise in a problem domain available in

an explicit form, the fact is that the actual reasoning strategy is still implicit, in most systems, in the form of the rules themselves. This has only been addressed to a limited extent by meta-level systems such as MECHO and NEOMYCIN. This is a fundamental point that is central to the problem of designing a system that can ensure integrity of knowledge throughout a process design and is returned to in greater depth in Chapter 4.

3.3 Blackboard Systems

A sophisticated process design environment, as envisaged in chapter 1, will be a large and complex system. The concept of a design methodology based on A.I.P., Appropriate and Incremental Parallelism, requires by its very nature communication and cooperation between a number of engineers, or systems, knowledgeable in different areas of expertise. Agreed design decisions and information must be managed and made available to the relevant parties throughout the design process. Given the problems experienced in applying production systems to very specific, small to medium scale problems it seems unlikely that these systems on their own can be successfully applied to the much larger and more complex problem of process design.

In the period mid 1970's to early 1980's, a number of systems were reported that specifically studied problems requiring relatively incompatible types and sources of expertise for their solution. They are generally referred to as "blackboard" systems since they are modelled on the idea of a number of experts cooperating to solve a problem, the solution being gradually developed on a blackboard. The blackboard is a common data structure and each expert is expected to make a contribution when the state of the solution suggests that he/she can perform a useful task. In many ways the overall structure is similar to developments in integrated database systems, see chapter 1.

At first sight, therefore, the general structure of a blackboard system seems well suited to a design environment based on an A.I.P. design methodology. Rather than describe any one system in detail, only the typical components of a system shall be given here. Three important systems discussed extensively in the literature are HEARSAY-II, Erman et al (1980), HASP, Nil and Freigenbaum (1982), and OPM, Hayes-Roth (1983).

(HEARSAY-II is a speech input/understanding system used to access a database developed during the 1970's. HASP is a system used to process signals from an array of hydrophones to detect the movement of ships and submarines. OPM is a flexible planning system used to sequence tasks to solve problems in a specified domain).

There are three principal components in a blackboard system: the blackboard, a number of experts, usually called knowledge sources (KS's) and a system scheduler. Each of these is now considered in turn.

The blackboard is a globally accessible, multi-dimensional data structure consisting

of a series of entries. An entry usually refers to some fact, hypothesis or piece of control information. One of the dimensions is usually "representation level" running from low level data up to high level abstractions. Another dimension is usually used to represent the number of alternative hypothesis available to use in some sub-task. Other control information is often recorded on the blackboard as well. The important point is that each dimension is usually partitioned or segmented into a number of levels, ranging from low level information to high level information. In order to record the dependencies of generated assumptions or hypotheses there are usually a lot of dependency links between statements at each level.

From a logical point of view, it is worth noting that most entries on a blackboard could be represented in a straightforward way as a set of predicates. The idea of proportioning the blackboard entries into layers along several dimensions can be achieved by suitable use of extra parameters indicating both layer level and dimension type. Nevertheless, the idea of a communal blackboard is a useful one but should be considered independently of any implementation details.

The individual knowledge sources, agents or modules cooperating in the system are often mini-production systems stripped of any explanation facilities. They could, of course, be any self contained program or procedure with a suitable interface to the blackboard. Normally, however, each KS consists of a precondition part and an action part. The precondition part typically specifies the conditions needed to make it worth activating the KS. This is usually in terms of specific entries or types of entry at a given level on the blackboard. In addition, a subjective estimate of the likely benefit, required resources, likely hypotheses etc, resulting from the use of the KS is usually reported to the system the first time the precondition is analysed.

The action part of the KS is the code that is evaluated when the precondition is satisfied and the KS has been selected to perform some tasks. In some systems the action parts of KS's are partitioned sets of rules interpreted by some common production system using the blackboard as its working memory.

The third and final part of a blackboard system is the scheduler which controls the operation of the system. Many schedulers are based on the following simple strategy:

1. Note any changes, e.g. new entries, on the blackboard.
2. Determine which KS's would act on these changes. Rather than poll each KS on every cycle control instructions are often generated by the scheduler the first time it inspects the precondition of the KS. Alternatively control instructions are directly programmed into the scheduler for each application.
3. Create an agenda of "knowledge source activation records", KSAR's. These are simply records of instances of variable bindings of the preconditions of each KS that might be invoked.

4. Decide which KS to invoke by ordering the agenda according to some algorithm.
5. Invoke the KS using as arguments the bindings recorded in the KSAR on the agenda.
6. Record any additions or changes to the blackboard requested by the KS and repeat from step 1.

The performance and control of the system depends crucially on step 4). KS's could well be applied in the wrong order, get into recursive loops etc. Excessive amounts of unnecessary search could be performed or, worse, failure of the system. The problem of deciding which KS to activate is just the same as deciding which rule or set of rules to activate in a production system. Indeed, if the action part of a KS is a set of rules then it is exactly the same problem.

The approach taken in the OPM system is to allow the process of sorting the agenda and activating the KS's to itself be determined by a further set of KS's which are solely concerned with reasoning about the control strategy. This is equivalent to taking the meta-level techniques of MECHO and MEOMYCIN, say, and expressing them in a slightly more general framework. It is worth briefly describing the various levels that OPM uses.

The levels used in the domain or problem description part of the blackboard are as follows:-

1. OUTCOME - This is the highest level and describes the problem information.
2. DESIGN - This records decisions about moving from one "cluster" of activities to another. Sub-problems to be solved are grounded together into "clusters". This level is concerned, then, with decisions regarding problem solving strategy.
3. PROCEDURE - Record decisions about moving "within a cluster".
4. OPERATION - This is the lowest level and is concerned with "single step" activities that comprise PROCEDURES.

The levels in the control part of the blackboard permit movement between the domain levels above. They are:-

1. PROBLEM - This is the highest level describing in general terms the problem to be solved, eg "a multiple task planning problem, starting time = ti, finishing time = ... etc".
2. STRATEGY - This level records guidelines on how to solve the problem, eg: TOP DOWN - Top Down, and is similar to choice of inference method in MECHO. The basic idea is to FOCUS on successive levels of refinement, top down style.
3. FOCUS - Records where to focus or restrict the system's attention, eg at the DESIGN level or PROCEDURE level or OPERATION level.
4. POLICY - General decision criteria to be used after a number of cycles, eg fast or detailed KS's, highest expected value, reliable, etc.

5. AGENDA - Records entries on the agenda on each cycle, ie the KSAR instances.
6. KSAR - The entry on this level is the KSAR from AGENDA to be fired on that cycle.

For each of the control levels, PROBLEM, STRATEGY, etc, there are a number of control KS's, see Hayes-Roth (1983) for full details.

The scheduler consists of three knowledge sources, an update KS, a scheduler KS and an interpreter KS. These KS's are defined, in abbreviated form, as:

```
KS : AGENDA-UPDATER
Precondition:
  There is a change to the most recent scheduled KSAR, KSAR-n,
  indicating it has executed.

Action :
  Create a new agenda = prior agenda minus KSAR-n.
  If KSAR-n produced changes on the blackboard
  then remove any KSAR's from the agenda
       whose preconditions have been invalidated by the changes
  and add a KSAR for any KS whose precondition is now satisfied.

KS : SCHEDULER
Precondition :
  There is a new agenda, Agenda-n.

Action :
  Choose a KSAR by applying an evaluation function
  to operative Foci and Policies for all KSAR's.
  Create a new scheduled KSAR and the factors
  influencing the decision to schedule it.

KS : INTERPRETER
Precondition :
  There is a new scheduled KSAR, KSAR-n.
Action :
  Interpret and execute KSAR-n.
  Change KSAR-n to record pointer to all the
  resulting changes on the blackboard.
  Change KSAR-n to indicate that it has executed.
```

Careful study of the SCHEDULER KS shows that responsibility for activating a specific KSAR has moved to an "evaluation function" of sorts. The selection criterion for this function are specified by a control KS at the STRATEGY level which might recommend "importance", say, as the criterion to use. The scheduler would then look for expected values of "importance" in the precondition parts of the DESIGN or PROCEDURE KS's depending on the current FOCUS. These values are subjective estimates provided by the programmer developing the domain specific KS's, e.g. a KS for an overall mass balance would have higher importance than that for a detailed reactor model in the domain of process synthesis.

3.3.1 Observations on Blackboard Systems

On first inspection the general principles used in blackboard systems seem to overcome some of the problems experienced with using production systems on multi-faceted problems. Rule sets can be partitioned into KS's on specific topics. The resulting KS's should be much smaller and easier to maintain than the original, large, unstructured rule set. Procedural code and declarative knowledge can easily be accommodated in the same environment by careful placement into appropriate KS's.

On closer inspection, however, it is clear that the whole emphasis on the use of meta-level information to directly guide the reasoning strategy has moved to a situation where this information is secondary to the planning and activation of KS's. In early descriptions of the OPM system, the meta-level information becomes quite localised, i.e. KS specific, and heuristic in nature, numbers used to estimate relative importance, reliability etc. The quality of meta level information being used on each cycle of the scheduler then, is less than it should or could be, ie: the heuristic estimates are static and do not take account of possible global interactions with other KS's in a particular problem.

A blackboard system, such as OPM, also requires a specific scheduler or planner to coordinate the activation of the KS's. This, in principle, should allow very flexible forms of reasoning to be carried out on the control and operation of the system. In practice, however, planning of tasks turns out to be a very complicated problem in its own right. A great deal of literature has been published on the development ot planning systems in A.I. Nilsson (1982) is a good introduction to the well known linear, hierarchical planning systems STRIPS (Fikes and Nilsson (1971)), ABSTRIPS (Sacerdoti (1974)) and WARPLAN (Warren (1974)) in the context of logic based theorem proving. Well known non-linear planners have since been developed, e.g. NOAH, Sacedorti (1977) and NONLIN, Tate (1977). (The difference between non-linear planners and linear planners is that the former plan tasks in parallel and do not impose task order until necessary, i.e. they work with partially ordered plans. Linear systems are fully ordered). Tate (1985) is a general introduction to planning with many relevant references.

Instead of trying to form a complete problem plan, OPM sidesteps the problem by adopting an opportunist "try it and see" approach. The KS activated on any particular cycle depends on the current entries on the blackboard resulting from activation of other KS's. Care must be taken with this approach, however, to ensure that the system progress towards a global solution and not get stuck in a depth first search.

Despite the potential control problems, work was started on developing a type of blackboard system for use in process design. This research is described in the following section.

3.3.2 Development of Designer's Assistant

The purpose of this section is to describe the initial development of a system, called Designer's Assistant, loosely based on a blackboard style architecture. It is clear from previous attempts to provide integrated database design environments and the requirements needed in a more sophisticated environment, see chapter 1, that

1. The resulting system will be large, complex and required to cover many different areas of design expertise, and
2. Complex/flexible interaction will be needed between these different areas of expertise.

The consensus in the literature, and from the experience gained in the development of the heat exchanger selection system, is that rule based production systems or theorem provers do not provide a suitable framework to tackle such a large problem. The general notion of an environment consisting of a number of cooperating agents or modules that communicate with one another, however, appears a more attractive proposition, and one that is amenable to practical solution. There are a number of basic advantages to this approach.

1. Complex design tasks can be broken down into identifiable sub-problems. Contributions to their solution can be made by a number of different modules competent in different areas of expertise, thus supporting the concept of an A.I.P. design methodology.
2. Practical consideration of usage of computer resources require that the system be partitioned in some way. Providing a set of distinct, independent modules greatly simplifies the problem of allocation and control of computer resource.
3. A number of different types of inference techniques will inevitably be needed to solve problems of a different nature. Some problems will be of a planning nature, others involving selection decisions or simply evaluation and analysis of procedural code. It is likely that specialised interpreters will be required for different purposes. Those can be conveniently embedded within any module in the system rather than have one all embracing interpreter.

The idea of having alternative inference techniques, and perhaps alternative representation techniques suitable to the particular problem being solved, seemed attractive at the time. It would mean that full advantage could be made of specialised interpreters already developed, such as the heat exchanger demonstration system. The work described in the following sections was started on this basis. Subsequent insights to the problems that this created radically altered this view and are discussed at length in the remaining chapters of this thesis.

Implementing a generic blackboard system is a significantly larger programming task than development of a one-off production system. Rather than spend a lot of time developing the overall framework, which is relatively uninteresting from a representation point of view, it was decided to initially concentrate on the development of a single module and its interaction with the blackboard. This work is described below. Development of the overall structure of an enhanced version of the system is described later, chapter 5.

3.3.3 Development of a Physical Properties Module

The selection of methods for physical property estimation is a fundamental part of modelling in process design. As such, it is typical of a module in a blackboard system that will be needed or called upon by many other problem solving modules. For example, the heat exchanger selection system requires estimates of specific heats, boiling points, etc. The user is currently prompted for these values but it would be desirable if a request could be posted on a "blackboard" and answered by another module in the system.

The choice of physical property method selection is interesting in that it brings into focus the important question of how process models are to be constructed in a design environment and how process models are to be constructed in a design environment and how information in the model is to be used by different modules. The development work is described in four parts: module language definition, model selection and interpreter strategy, model representation on the blackboard and interaction with the module.

3.3.4 Module Language Definition and Knowledge Representation

The selection of methods for physical property estimation is normally an important part of some other process design task. The physical properties module must address, therefore, not only the problem of representing and using expertise in this domain but also that of constructing or enhancing some process model to be used in the wider problem context. A partial model may already exist and be represented on the blackboard, in which case the physical properties module should take account of any relevant details or constraints. If there are no such constraints the module will be free to question the user when necessary in constructing details of a new model.

The problem of selection of appropriate method for physical property estimation is similar in nature to that of heat exchanger level but a lower level in the context of process design. It was decided, therefore, to use a rule based inference technique as the means of method selection. The question of model construction, however,

required that an enhanced rule definition language and interpreter be developed. The rule syntax is now described.

3.3.4.1 The Rule Head

The rule head is a term of the form.

ruleN(<PROPERTY>)

Where <property> is replaced by an actual property, e.g. viscosity, density etc. This term is used as an index by the interpreter to classify the rules into specific sets of rules relevant to each property known to the module. In fact, the rules can be further classified by replacing an atomic value for <property>, eg: "viscosity", by a structural term such as "viscosity(low(pressure))". The system would not consider such a rule if the pressure was known to be high in a particular problem. Each rule is uniquely identified by the value used for N in ruleN.

3.3.4.2 The Rule Entrance Guard

Each rule has an "entrance guard" associated with it. An entrance guard is an arbitrary logical formula that must be satisfied, i.e. evaluate to true, before a rule can be even considered for application. The truth conditions are strictly evaluated and cannot be "compromised" in any way. Each condition in the guard has the same syntax as statements in the rule antecedent, see below. A simple condition might be

stream(phase) = gas

indicating that the physical property model that follows only applies to a gas phase stream. The interpreter uses the condition guard of rules to further restrict the set of applicable rules.

3.3.4.3 The Focus of Interest

Each rule has a term of the form.

focus_of_interest <ObjectType> ,

where <ObjectType> is replaced by the specific type of object that the rule is mainly concerned with. For instance,

focus_of_interest stream

Indicates that the rule can be used to infer something about the <property> of the stream of interest. The means by which the interpreter establishes the focus-of-interest is described later. This is used both as a reference point for expressions referred to in the rule body and as another mechanism for focussing the system's attention on the relevant set of rules.

3.3.4.4 The Situation/Action Body of the Rule

The rule body is of the usual form

if <antecedent> then <consequent>

<antecedent> is some logical formula such as

```
If stream (class) is pure
           and pressure = < "5 bar"
           then ..
```

Each premise in the antecedent can be one of the following forms

```
X is    Y,    X is not    Y,
X has   Y,    X has not   Y,
X does  Y,    X does not  Y,
X are   Y,    X are not   Y
X = Y (or <, >, > =, = <)
```

An additional form that can be used is

```
Property of X is Y
```

This is used to refer to the property of an object in a rule when the object is not the current focus of interest. What is meant by an "object" is described later in section 3.4 and concerns entries representing objects on the blackboard data structure.

Functional or procedural terms can be evaluated by the forms.

```
result_of( <goal-expression> )
```

or

```
evaluate( <procedure-call> )
```

<goal expression> can be some arbitrary arithmetic expression or one of the special forms

```
largest(<thing>, in (<object>)),
smallest(<thing>, in (<object>))
```

These forms are used to find maximum or minimum values of individual properties of, say, a stream mixture, e.g. the largest pure component molecular weight.

The largest/smallest forms can be used in arithmetic expressions and a "result_of(...)" term can appear in a relation form,
e.g. "result_of(...) > X". Figure 3.6 shows a number of example rules.

Figure 3.6: Example Rules in the Physical Properties Module

rule2(viscosity(low(pressure))) :

```
strict_guard( one_of([ stream(phase) is gas,
                       stream(phase) is vapour ]),
              one_of([stream(class) is pure,
                      stream(class) is essentially_pure ])),

focus_of_interest stream
if    stream(class) is pure
      and pressure =< 5.0e5
      and component is_not polar
then  model(viscosity(low(pressure)))
            by "Enskog_Lennard_Jones_equation" cf 0.95
            references [pooled(pp(ref(1)))] .
```

rule7(viscosity(low(pressure))) :

```
strict_guard( stream(class) = mixture,
              one_of([stream(phase) is gas,
                      stream(phase) is vapour]) ),

focus_of_interest stream
if  pressure =< 5.0e5
  and result_of( largest(component(mw),in(stream))/
                 smallest(component(mw),in(stream)) ) =< 6
  and result_of( largest(component(viscosity),in(stream))/
                 smallest(component(viscosity),in(stream))) =< 6
then model(viscosity(low(pressure)))
     by "Chapman_Enskog_Wilke_equation" cf 0.95
     references [pooled(pp(ref(1)))] .
```

The consequents of the rules are all of the special form

```
model(<property>) by <technique> cf <confidence>
      references <information-sources>)
```

This form causes special entries to be entered in the active blackboard. Finally, Prolog variables can be used within rules, as in the heat exchanger selection system, to provide a unification mechanism.

3.3.4.5 The Applicability Compromise

The "applicability compromise" of a rule is the fifth constituent part of a rule. An interesting, and very useful, feature of the interpreter in the physical properties module is that even if any part of the rule antecedent fails then the interpreter will still try and use the model suggested in the consequent provided the bounds of applicability are not stretched too far. This is what the applicability compromise specifies, suggesting how the confidence in the model should be modified appropriately. The only restriction on making compromises is that the entrance guard must strictly evaluate as true without any compromises being made.

The following is a typical example of a rule compromise

```
applicability_compromise( rule2(viscosity(low(pressure))) ) :
[
 ( if stream(class) is essentially_pure change cf to cf-0.05 ),
 ( if pressure is within_range(5.0e5,20.0e5)
      change cf to cf - cf*((pressure-5.0e5)/(pressure+5.0e5))),
```

```
 ( if component is slightly_polar change cf to cf-0.05 ),
 ( if anything_else_fails change cf to 0.05 )
].
```

A compromise is simply a list of compromises that may be made for quantities referred to in the rule antecedent. The expressions are of the form

```
if <Test> change <Thing> to <New-Value>
```

<Thing> can refer to anything but it is usually the confidence factor used to score the merit of a model. The example entry also shows the use of two other forms. The first is the function "within-range" and is self-explanatory. The second is the use of the form.

```
if anything-else-fails change ...
```

This is a catch-all that can be used to compromise any rule premise not specifically mentioned.

N.B. The use of an applicability compromise is optional and need not be specified.

3.3.4.6 Making Data Compromises

The applicability compromise is used to make allowance for conditions on the use of a property estimation technique not being met on the assumption that the data being used is correct. It tends to be the exception rather than the rule, however, that good data is readily available for modelling purposes in process design. The "data-compromise" part of a rule caters for this common situation by indicating how use of estimates of data, rather than reliable values, affect the confidence of the use of a particular technique. Some typical entries are

```
data_compromise( rule2(viscosity(low(pressure))) ) :
[
 ( if hard_sphere_diameter is estimated_by Anything
       change cf to cf-0.1 ),
 ( if dimensionless_temperature is estimated_by "Neufeld_LJ_equation"
      and dimensionless_temperature is outside_range(0.3,100)
   change cf to cf*0.5 )
].
```

The quantities referred to are called the "parameters" of the method referred to in the "model" statement of the rule consequent. Details of how this data is accessed or estimated are given below.

3.3.4.7 Observations on Rule Format

It is hopefully clear from the example entries that recommending techniques for specific modelling purposes can be achieved by writing rules in an easy to read, natural format. The facilities provided, via the syntax of rules, in the physical properties module represent a considerable extension to those found in the heat exchanger selection system. Of particular importance is the use of applicability and data compromises. Many heuristics in process design are of a very approximate

nature and need some insight as to how they should be applied. The use of applicability compromises is a first attempt at providing a simple way of specifying how the context of a problem can be taken into account when applying some overly simplistic heuristic.

The ability to make compromises regarding the availability and reliability of data is another essential facility. If no data is available when a request for some value of a physical property is made, the system need not fail, as usually happens, but recommend instead some estimation techniques for the missing data with an appropriately lowered confidence in the use of any values generated for the physical property. A typical example would be estimating the acentric factor from critical properties for subsequent use in evaluating Z, the compressibility factor, from an equation of state.

An additional facility, not yet mentioned, is that any number in an expression can be expressed in quotes with appropriate units, eg: "5 bar". Expressions themselves are symbolic in nature, referring to various aspects of the focus of interest. All such expressions are symbolically evaluated and make use of a general units conversion technique written in Prolog. All units are converted internally to an S.I. system for consistency. Any numbers not appearing in quotes are assumed to be in the modules's units.

Finally, it is worth noting that the rule format is by no means restricted to description of physical property estimation techniques. The heat exchanger selection rules, for example, could easily be written in this layout. In different problem areas the type of object used as the focus of interest will determine the nature of the "data" that could be low level information, as in the physical properties module, or high level pieces of information. A "flowsheet" object as focus of interest, for example, might refer to unit models and their connectivity as the "data parameters".

There is a fundamental aspect of the rule syntax that has not been described, however, and that is the definition of a module language of terms that can be used in statements such as "if X is Y". This is described later once the other components of the module have been explained.

3.3.4.8 Structuring of Physical Property Domain Data

A feature common to many systems, e.g. NEOMYCIN, is the use of taxonomic hierarchies. These hierarchies are used to classify "types" that are of interest in the domain, e.g. types of medical disease. The use of hierarchies makes it easier to access and maintain "static" domain information/data in a structural way. Nodes in

the hierarchy are often used to store information appropriate to that level in the hierarchy. This aspect was used in the physical properties module to classify chemical components as now described.
(N.B. The methods used here were an early attempt to represent immediately useful aspects of object hierarchies in the physical properties module. Section 3.4 outlines the subsequent development of much more sophisticated techniques involving object hierarchies).

All chemical compounds known to the physical properties module are placed in one of two basic classes, namely organic or inorganic.
Organic chemicals are further classified under the three main groupings hydrocarbons, polymers and biochemicals. Only hydrocarbons are considered further. There are a number of ways in which hydrocarbons can be classified. The approach taken in the physical properties module was to classify them according to the dominant sub-groups which characterise the chemistry of the compounds. This results in any homologous series being kept as a class. The main class hierarchy, represented as a set of Prolog clauses, is shown in Figure 3.7

FIGURE 3.7: Compound Class Hierarchy

```
full_class(organic,organic).
full_class(inorganic,inorganic).

full_class(alkane,organic(alkane)).
full_class(alkene,organic(alkene)).
full_class(diene,organic(diene)).
full_class(allene,organic(allene)).
full_class(alkyne,organic(alkyne)).
full_class(alcohol,organic(alcohol)).
full_class(alkanal,organic(alkanal)).
full_class(alkanone,organic(alkanone)).
full_class(ester,organic(ester)).
full_class(alkanoic_acid,organic(alkanoic_acid)).
full_class(epoxide,organic(epoxide)).
full_class(alkyl(X),organic(alkyl(X))) :-
     member(X,[halide,sulfide]).
full_class(phenol,organic(phenol)).
full_class(aryl(X),organic(aryl(X))) :-
     member(X,[halide]).
full_class(arene,organic(arene)).

full_class(element,inorganic(element)).
full_class(allotrope,inorganic(element(allotrope))).
full_class(halogen,inorganic(element(allotrope(halogen)))).
```

Classes of inorganic compounds were not developed other than identifying common elements such as the halogens.

Each sub-class in the hierarchy is considered to be a node further described by default information common to all members of a sub-class. The following entry, for example, describes default characteristics of members of the alkane sub- class.

```
class_character(alkane,
                [aliphatic, unsubstituted, non-polar]).
```

Instances, or members, of each class at the next level in the hierarchy are specified by another set of predicates. Typical entries are as follows.

```
component_class(alkane,1,1,19,ppdata,
                compounds(methane("CH4"),ethane("C2H6"))).
component_class(alkene,2,20,39,ppdata,
                compounds(ethene("C2H4"))).
component_class(alkyne,5,80,99,ppdata,
                compounds(ethyne("C2H2"))).
component_class(alkanal,7,120,139,ppdata,
                        compounds(methanal)).
```

The last argument of each predicate lists the members of the sub-class named according to I.U.P.A.C. conventions. Synonyms and formulae can be provided in following brackets. The name of the relevant sub-class is the first argument. The second, third and fourth arguments are used for indexing classes and compounds numerically. The fifth arguments indicates where information describing the class members can be found, in this case a set of prolog predicates in a file "ppdata".

Each node in the hierarchy at the compound level is described by another predicate with information specific to each compound. The form of the predicate used is

```
component_data(macro_list(LIST),
   name_of(component),
   standard formula,
   class_of(component),
   molecule_type,
   'component phase at 1 atm,298 K',
   molecular_weight,
   freezing point (K),              [status,accuracy,references]
   boiling point (K),                    "       "          "
   critical temperature (K),             "       "          "
   critical pressure (atm),              "       "          "
   critical volume (cm^3/g_mol),         "       "          "
   critical Z value,                     "       "          "
   acentric factor,                      "       "          "
   dipole moment (debyes),               "       "          "
   (mean)radius of gyration (Angstrom)"        "          "
   stockmayer_polar_parameter            "       "          "
   association_constant                  "       "          "

, where status     = {estimated_by(model),experimental}
        accuracy   = +/- X , X being a percentage
        references = a list of reference papers etc.

Anything that is unknown is replaced by a question mark ?
```

In the terminology of object hierarchies the nodes in the hierarchy represent compounds and are called object instances. They are defined by a set of slots, or attributes, as listed above. The values of these slots are provided by instances of this generic predicate. The following entries are taken from the physical properties module:

```
component_data(
  macro_list([ X=[experimental,?1,[ref(physprops,1)]] ]),
```

```
  methane,"CH4",alkane,monatomic,gas,16.043,
  90.7,X,111.7,X,190.6,X,45.4,X,99.0,X,0.288,X,0.008,X,0.0,X,
  1.123,[experimental,?1,[ref(physprops,2)]],0.0,X,0.0,X).

component_data(
   macro_list([ X=[experimental,?1,[ref(physprops(1))]],
                Y=[experimental,?1,[ref(physprops(2))]] ]),
   methanol,"CH3OH",alcohol,monatomic,liquid,32.042,
   175.5,X,337.8,X,512.6,X,80.959E5,X,118.0E-3,X,0.224,X,0.559,X,
   1.71,Y,1.536,Y,0.5,[calculated,?1,[ref(physprops(1))]],1.63,Y).
```

The purpose of providing all this classifying information is to allow statements in the rule body to be expressed in a concise way. Consider, for example, a pure component stream, methane say, and the rule condition.

```
if component is not polar
```

The node information for methane makes no explicit mention of polarity but the node information for alkanes does. In the absence of other information the system will use this default information to evaluate the premise above to be true. The use of node information in a hierarchy in this way is a simple example of "inheritance", i.e. default information can be inherited from a parent node to a child node in a hierarchy if no other contradictory information is available.

There are a number of simple utility clauses provided to allow statements of the type.

```
if stream(class) is organic
```

to be evaluated. If the stream in question is pure component ethane, say, the interpreter will establish that alkane, the immediate parent class of ethane, is a compatible subclass of organic but not, say, of inorganic.

Certain aspects of background domain information can be represented, therefore, in a concise way using object hierarchies and default information. Caution must be exercised, however, in the use of default information in any theorem proving process as noted earlier in the description of the MECHO system.

3.3.4.9 Main Components of the Blackboard Data Structure

The blackboard data structure in Designer's Assistant consists, at the lowest level, of a collection of prolog structures recorded in the internal database of Prolog. A number of such structures taken together represent the "entities" or "objects" that the system can reason about. The two principle entries on the blackboard are referred to as the "system" and its "surroundings". These two entities are used to represent the overall context of any process modelling problem in terms of high level descriptions of any objects of interest. The description of the problem to be solved is stored elsewhere.

The notions of a system and surroundings to describe many types of process design problems is a common one and has the advantage that it can be applied recursively at

increasing levels of detail. Details of a system, or its corresponding surroundings, are recorded under the following five areas or slots:

1. description - a collection of predicates stating details of a problem situation not covered by points 2) to 5) below.
2. Component-data - component information if the model involves physical property information.
3. Stream-unit-network - a system is taken to consist of "units" interconnected by "streams". Details of their names and connectivity are given here.
4. Observations - a collection of module generated statements.
5. Surroundings - the name of the surroundings entity.

The five areas are simply used to partition predicates and control information into useful subsets. Details of streams and units specified in a system are recorded under another set of entries on the blackboard. In the case of streams representing process plant streams the slots used are "observations, components, composition, temperature, pressure, enthalpy and process models". The first and last slots allow arbitrary kinds of information to be stored regarding a particular instance of a stream.
Similarly, information on a process unit is recorded on the blackboard under the general headings "observations, process-model, real-model, control-scheme, mechanical-description, free-variable-analysis and cost-analysis". The difference between a process model and a real model is that the former refers, ultimately, to some set of equations describing an idealised behaviour whereas the real model is a set of predicates characterising the "real" behaviour of the unit.

The original intention was to create a blackboard structure that could be used in a variety of front-end process design problems. The generic concepts of a system, its surroundings, process streams and units should allow this to be achieved. For example, the heat exchanger selection system would make use of both process streams and units. The type of exchange selected could be recorded under "mechanical-description. Any factors deduced or assumed in the selection process would be stored under observations. The use of a "system" or "surroundings" would only be for initialisation purposes if advice was requested on a "one-off" basis. If a system description already existed, however, the selection of a heat exchanger type could make use of any relevant process information available.

3.3.5 Overall Strategy of the Physical Properties Module Interpreter

Having described two aspects of the representation style used in the physical properties module, namely the rule syntax and object hierarchy, the control strategy of the interpreter shall now be discussed. There is a third, and very important, part of the overall representation technique concerned with the definition of terms in a module that has not yet been discussed. This will be introduced in section 3.3.7 later

as it is first necessary to appreciate how the module interpreter interacts with the use of the blackboard system.

The description of the interpreter that follows assumes that the module has just been activated, i.e. the system scheduler has made a request to the physical properties module to perform some action. It is important to note that what is being referred to is a completely generic framework that could be used in a number of other modules if desired.

The top level call in the module interpreter consists of four simple goals.

1. Initialise the module
2. Plan any requests
3. Attempt to evaluate the sequence of planned requests
4. Prepare a report then FINISH

The various features of interest are most easily introduced by briefly summarising each of these goals in turn.

3.3.5.1 Initialisation of a Module

The software model used in Designer's Assistant is that the core system communicates with each module via a number of communication streams. Information is sent to and from a module via these streams. This means that modules can operate in a concurrent system if the streams are implemented, say, as communication sockets on a local area ethernet system.

The first part of initialisation is to connect the input and output streams of the module to those specified by the core or central system. Any subsequent input requests or generated output are then redirected as required without any special action on the part of the module. There are two output streams : one to ask questions on and the other to send a report to.

The second phase of initialisation is to check that the module has the minimum amount of information necessary to work with. This call, "check-set-up", is obviously domain dependent. A module can be activated in one of two ways in Designer's Assistant. If a user makes a specific request to a module, eg: for some advice on physical properties, then the physical properties module has full control over editing entries on the blackboard and is called the "active-host". If, however, a module is called on by another module, rather like a subroutine call, then it only has "read" access to the blackboard. In this case "check- set-up" may have to create entries on a "local" blackboard for its own use. The calling module is responsible for finally updating the "global" blackboard when the called module returns with its

answers.

This software model allows much stricter control over entries being made on the blackboard than a system like OPM.

The third part of initialisation is accessing any relevant domain dependent information. In the case of the physical properties module the compound class hierarchy and physical property data are loaded into the system. In a real application this data would probably be kept on an external data storage system.

Finally, the module records what the top level focus of interest is. This is specified in the calling argument list when the module is activated.

3.3.5.2 Planning Requests

The top level requests are given to a module when it is activated. Once the module has been initialised the interpreter then tries to formulate a local plan to perform these requests in an efficient manner. It should be noted that this plan is strictly local to the module and no attempt is made to formulate some global plan of execution. The literature suggests that current planning techniques can be used effectively on small to medium scale problems. The approach taken here, then, is to formulate a rough plan of action that is

1. local to the module and
2. can be used to guide the rule interpreter.

The "planner" in the physical properties modules is a simpler linear, depth first search in the style of the STRIPS system, Fikes and Nilsson (1971). It is known that this technique will fail in complex, interacting problems. This is not a general problem as different planners can be used in different modules and are wholly self contained. The top level clause is of the form.

```
plan-requests(ModuleName, ActiveorRemote, InputData,
                Requests, PlannedRequests, FocusofInterest) :- ...
```

The use of a planner local to a module seems a reasonable compromise between an opportunistic style, which is hard to control, and a global planner, currently infeasible in terms of computing resource. Indeed the synthetic or creative nature of process design requires that a certain degree of "opportunism" always be retained. A completely prescriptive approach to design is doomed to failure, at least in economic terms, and contradicts the basic principles of systems engineering outlined in Chapter 1.

3.3.5.3 Evaluating Planned Requests

Evaluating a list of planned requests simply consists of calling the rule interpreter for each goal in turn. Each planned request has an associated focus of interest that is

determined by the planner. The focus of interest is noted each time the rule interpreter is called. The only variation possible is that among the requests there may be strict "directives" to perform some action. If so, these are evaluated before any requests are interpreted. This mechanism is primarily intended to allow default information stored in the module to be overridden by problem specific information.

The interpreter is called with the next request and focus of interest as arguments. In the case of the physical properties module each request is classified as either "advise-only" or "advise and evaluate" depending on whether any calculations are needed.

In either case interpreter_pass 1" is called. The purpose of the first pass is to generate a set of partial models regarding the principle property or quantity referred to in the request. Any properties mentioned, e.g. vapour pressure, take the current focus of interest as the point of reference, stream12 say. The system first checks that it has not got stuck in a recursive loop by checking internally maintained stacks of control information.

A model is selected from the rule base by using the index part of the rule heads and the type of object referred to in the focus of interest part of the rule. The use of this model is noted and then investigated regarding suitability of use.

The source code calls are

```
get_a_model(IDEA, Property, Strictguard,
              FoiType, Antecedent, Consequent),
observe_use(IDEA),
investigate_model(IDEA, Property, Strictguard,
                  FoiName, Antecedent, Consequent)
```

The process of investigating a model follows directly from the rule syntax described previously. The main steps involved are:

1. Test the rule entrance guard. If this fails then the model is inapplicable.
2. Evaluate each of the conditions in the rule antecedent. Note any condition that fails.
3. If each condition in the antecedent is true then evaluate the rule consequent and RETURN.
4. If any condition failed then check whether a compromise can be made. If so re-evaluate the model.

A large set of utilities were developed to perform the work involved in step 2. For example, statements of the form-

```
X is Y, where X is the focus of interest
P of X is Y, where X is not the focus of interest, P is a property.
```

require utilities to access the blackboard data structure. Other utilities include

symbolic expression evaluation, units conversion, etc.

If the answer to a conditional statement is not known then one of two things happen:

1. The definition of the module language is interrogated to see if it can be found by solving some other sub-goal(s). (This has not yet been described but see section 3.3.7). If it can, try and solve these sub-goals.
2. A request is sent to the central system on the output communication stream to see if some other module can evaluate the request. This may, in turn, result in another request being placed with the original module. Hence the checks earlier for recursive loops. The module waits, meantime, for some response on the input communication stream.

The module can therefore search sub-goals in the usual backward chaining style or can ask for assistance from other modules in the environment.

The whole process continues until a model has been investigated. At this point the "process model" entries on the blackboard are updated. The items of information recorded are

- a fact expressing the suitability of the model,
- a value for its relative merit,
- the property being modelled,
- the name of the model,
- the source of the rule expertise,
- any conditions that failed, and
- any compromises that were made.

Noting the use of compromises is useful in case the user can supply extra information to re-evaluate the model.

When a model has been investigated the system selects another model and the whole process is repeated. This continues until all the relevant models have been investigated or a model has been found with an estimated accuracy level greater than the required accuracy. The latter value can be set as a percentage figure by the user. At this point the first pass of the interpreter is complete.

The second pass of the interpreter is called if the mode of request was "advise-and-evaluate" as opposed to "advise-only". All that happens in the second pass is that the models are ranked according to relative merit and the best one evaluated. An estimation method can be evaluated either in symbolic form, if a definition is available, or by some arbitrary procedure call. If the symbolic form is used the expression evaluator in the rule interpreter is used. This means that further internal requests and sub-goals can be generated if necessary in a recursive fashion.

3.3.5.4 The Input/Output Form Filling System

An important utility used in both passes of the module interpreter is a general form filling system written for terminal based interaction with Designer's Assistant. (N.B. This utility was written before the widespread use of graphics workstations. The general principle, however, is still relevant).

Whenever a statement is evaluated and it is found that some piece of information is missing, the first option that is considered is to ask the user to avoid any unnecessary search. All such input/output requests are made via the input/output manager provided for any module. Messages are written and received on the communication streams set up during the initialisation of the module.

The form filling facility also uses the same utilities and can be called when several items of information are needed at once. This is the most useful when a module is used as the "active host", i.e. the one directly communicating with the user. If the host is active the default action is to prompt the user with a number of forms to fill in that specify some standard problems. This is typical in industrial process design software. Forms can also be used at any other point in search process. Each form is defined in two parts. The first part is just a sequence of instructions to draw the form on the terminal. A number of generic routines are provided such as -

```
move(X, Y, Screen)
draw-frame(LX, LY, RX, RY, Screen)
write(Mode, Message, Screen), etc.,
```

where screen is the name of the terminal device to write to. The idea is to draw a boxed menu on the screen. There is a small library of alternative definitions for alternative screen types.

The second part of the form definition describes three things:

1. The coordinates of an item of information.
2. A specification of what the item of information represents.
3. An arbitrary procedure to run and test the validity of the typed input, e.g. a numerical range of values, an allowed choice of words, etc.

A "form manager" displays each form and maintains a stack of the forms displayed. At any choice point the user can move back to a previous choice point or a previous form and change the input values on that form. The user can jump from form to form by using a command preceded by a "#" symbol and followed by "p", for previous form, or "-np" for nth previous form, or "+np", for n forms forward.

A module can, therefore, display more than one menu on a terminal screen at a time by judicious use of coordinates.

3.3.5.5 Report Generation

The final top level clause in the module interpreter, "gather-report-terms", is concerned with collecting together a set of statements that represent the top level conclusions or answers generated in response to the initial requests. The terms collected are a subset of those generated during the investigation of each model and are stored on a separate part of the "local" blackboard. The last action of a module is to write these terms to the report output stream set up in initialisation.

The intention is that each module can generate a final report if requested as well as return its conclusions to the central system or other calling module. The report can be generated using the form definition facilities described for input requests. One such domain dependent report was set up for the physical properties module to describe responses to specific user queries, e.g. advice on selection of an estimation method for a transport property.

3.3.6 The Problem of Arbitrary Tokenism and Module Language Definition

The concept of having a number of modules communicating with each other in a blackboard system, e.g. Designer's Assistant, brings to the fore a fundamental problem in knowledge representation that shall be referred to as "arbitrary tokenism" in this thesis. It is remarkable that widespread discussion of this problem cannot be found in the literature. It is introduced here, therefore, by a trivial example.

Consider a system that has two sets of rules that cover, say, two different aspects of vessel design. The first set may be used for initial sizing of the vessel, e.g. representing flash calculations for a flash vessel, while the second set may refer to detailed mechanical design of the vessel. The two rule sets could be represented in two different modules in the system, or both together in one module in a structured or partitioned rule base, or perhaps all mixed up together in an unstructured rule base.

Now consider the concept of "PRESSURE". Mention of the word "PRESSURE" appeals in different initiative ways to different types of engineers, or for that matter readers of this text, and will bring in to mind different ideas depending on the context of the use of the word. Suppose that one of the flash vessel rules concludes that "pressure is high". The reason for this might be that the working pressure is relatively high compared to the critical pressure of the fluid being flashed and it is useful to know this when selecting physical property estimation techniques in other rules, eg: "if pressure is high and ... then ...". Assume that this fact has been entered on the blackboard and details of a mechanical design have been requested. Now when the set of rules concerning vessel design were written it is likely that the use of the fact

"pressure is high" would indicate something very different, e.g. in absolute terms where "high" might mean greater than 50 bar, or in terms relative to the vessel strength, hoop stress etc. The set of mechanical design rules might involve terms, therefore, written in the same way but representing very different intentions.

Any generic production system or interpreter will be hopelessly confused by this situation and will generate either contradictory information, or even worse, verbally incorrect conclusions.

Consider the following attempted solutions of this problem. Suppose that "high" in both rules is taken to mean an absolute measure. The problem has not been solved but merely shifted to finding a common definition for "high", e.g. >10 bar, >20 bar etc. If 20 bar is taken as the value how should a rule be applied if the system pressure is 19.5 bar? If an absolute scale of measurement is used then a number of words must be invented to write rules in different modules that work with different ranges of pressure, eg low (<3bar), "very low" (<1 bar), "very very low" etc. This is very unsatisfactory as "very low" to an engineer might be "very high" to an experimental chemist. Exactly the same problem results if alternative words are used for pressure, e.g. "relative pressure", "allowable pressure". If a fact involving such terms is asserted by one module how is it to be used in another module if no rules in the module refer to that term. Unrecognised statements could implicitly represent essential information for use in another module. The literature available is filled with examples of rules using "made up" words or arbitrary tokens to represent some aspect of a more general concept. The delusion of using tokens in rules that merely look like English words in the vain hope that they represent something more significant, appears to be a common one.

The process of representing aspects a complex concept, e.g. PRESSURE, and all that it entails by an arbitrary word, without further definition, is what I refer to as arbitrary tokenism. Its effects are evident not only between modules and different rule sets but also within a single rule set where information implicitly implied by one fact cannot affect implicitly dependent conclusions. This results whenever the set of tokens, axioms and inference rules are inadequate to explicitly derive these inter-dependencies.

The first attempted solution to this problem is now given. It was based on the requirement that every token referred to in a rule, fact or expression in a module be further defined in terms of a definition language for that module. The principle motivation was to enable statements to be interpreted in a consistent way in different

modules by making a module's language definition available to other modules if required. Each entry on the blackboard is recorded along with the name of the module that placed it there. If a module, A say, uses a hypothesis from another module, B say, that involves a token defined in different ways in both modules A and B then the definition from Module B, and any other tokens required, can be reinterpreted in module A. If the definition of a token is the same in each module then no further interpretation is required.

Every token used in a module is classified as one of four basic types. The types used are referred to as "reference", "descriptive", "evaluable" or "adjective" terms. A reference term is a word used to denote a type of identifiable object or entity on the blackboard. Some simple examples of reference terms might be "stream", "component", "unit", "system", etc. Note that this is different from the name of a specific instance of a reference term, e.g. stream1, since names of reference terms are never explicitly referred to by value in generic definitions. Reference terms can be used as the focus of interest in a rule, or referred to on the left hand side of descriptive statements in rules. A descriptive statement is one of the forms.

```
X is   Y,  X is not   Y
X has  Y,  X has not  Y
X does Y,  X does not Y
```

A reference term can also be used in place of X in the form

```
Property of X is Y
```

or one of the other equivalent descriptive forms. Each reference term can be described by a number of descriptive terms or evaluable terms. For example, "phase" is a descriptive term for the reference term "stream". Similarly, "class" describes a component or compound class in the class hierarchy structure, "name" describes or names a stream, etc. Descriptive terms are referred to as the attributes or slot names in object hierarchy terminology. They can be used in the left hand sides of descriptive statements in the "subject" position, as for reference terms, e.g.

```
Class is Y              <refers to focus of interest>
Component (class) is Y  <shorthand form>
Class of component is Y <preferred form>
```

Evaluable terms are essentially the same as descriptive terms but are used to define physical quantities which can be measured and estimated numerically. They are defined with an associated set of units. For example, pressure as used in the "pressure of a stream" is defined as an evaluable term, units of N/m^2, whereas type as used in the "type of a stream" is defined as a descriptive term. Evaluable terms are normally used in relational statements, such as $X < Y$, but they can also be used in descriptive statements, eg: pressure is high, if the term to the right hand side of the "is" is

appropriately defined.

The fourth type of tokens are "adjective terms". The word "adjective" is used in a loose sense and is simply taken to mean anything that is used as the "value" of a descriptive/evaluable term. Adjective type tokens can only appear on the right hand side of descriptive statements. For example, "pure, essentially-pure, polar, slightly -polar" are all adjective terms that can be used in the right hand side of "is" in "X is Y".

All the words used in a module are defined in the same way. The general structure is

```
<word>    vocabulary        [<type>, <module-name>]
          stored-as         [<access-instructions>]
          requirements      [<statements-to-evaluate>]
          units             [<units>]
          test              [<evaluation-test>]
          side-effect       [<side-effects>]
          computed-by       [<procedure-calls>]
          user-question     [<input-request>]
```

A set of these prolog predicates therefore defines a module's language. In the structure above <word> is replaced by the appropriate token, eg: "pressure", "high", etc. <type> is the type of term, ie. reference, descriptive, evaluable or adjective, and <module-name> is the defining module. The <access- instructions> in "stored-as" are used for accessing entries recorded in the blackboard.

The "requirements" term is a list of descriptive terms needed only in the definition of an adjective. For example, the adjective "pure" applied to a stream requires the "number of components" of the stream to be known if used in "checking" mode.

When the rule interpreter encounters a statement such as "if stream (class) is pure" then the list of requirements of "pure" are set up as sub-goals to be searched. This does not evaluate the truth conditions of the statement. It simply satisfies goals that must be known before the truth conditions can be evaluated. A requirement may, in fact, just be the name of an arbitrary procedure to be called.

The units entry is only used for evaluable terms and is simply a string defining the system units for the quantity.

The test entry is used to evaluate the truth conditions of an adjective term. Once the requirements have been satisfied, the values of the required descriptive/evaluable terms are substituted into the test expression. This expression is an AND/OR logical expression that must evaluate to true for the adjective to apply, eg: in the case of "pure" the test is "number of components = 1".

The side-effect entry is a control mechanism that can be used for a number of purposes. If the truth conditions for an adjective's test expression evaluate to true then a number of hypotheses that are directly implied could be entered into the blackboard. This technique can be also used for "context- layering" of rules. Alternatively, a number of procedure calls can be made at this point to perform some required calculation.

The completed-by entry is only used for evaluable terms. It is the sequence of procedure calls needed to evaluate the quantity. These calls are performed after the requirements are established and any substitution of values has taken place, as for a "test" entry.

Finally, the user-question entry is a number of input requests that can be asked to determine a value before any attempt is made to work it out. This may be a single question or a call to the form fitting manager described earlier.

Figure 3.8 below shows extracts from a number of source code entries for tokens used in the definition of the physical properties module language. The only extra feature used, and not described, is the use of common or pooled definitions and questions. This is simply a space saving feature. The use of the underscore symbol "_" in prolog means undefined value.

Figure 3.8: Sample Extracts of Word Definitions

```
pure vocabulary    [adjective,physprops]
     requirements [number of components]
     units        ["dimensionless"]
     test         [[],number of components=1]
     side_effect  [].

essentially_pure vocabulary    [adjective,physprops]
     requirements [composition]
     units        ["dimensionless"]
     test         [[],member(T,composition),T=[Name|Mfs],
                     member(Y,Mfs),Y =.. [Phase,Mf],Mf>0.95]
     side_effect
                  [current_foi(stream,S),
                   store_in_model(S,new_observation,
                   (component(essentially_pure) of S is Name),
                   "dimensionless")].

slightly_polar vocabulary    [adjective,physprops]
    requirements [polar_parameter(stockmayer)]
    units        ["dimensionless"]
    test         [[X=polar_parameter(stockmayer)],
                  X >=0.05,X =< 0.1]
    side_effect  [].

pooled_definition(component(class),C)
  vocabulary [adjective,physprops
    requirements [component(class)]
    units        ["dimensionless"]
```

```
    test          [[],no_conflict_term(component(class),C)]
    side_effect   [].

alkane    vocabulary [pooled_as(component(class),organic(alkane))] .
alkene    vocabulary [pooled_as(component(class),organic(alkene))] .
alkanol   vocabulary [pooled_as(component(class),orgainc(alkanol))].
```

Although the partial solution presented here does go some way to solving some simple aspects of the problem, later work, see chapter 5, will in fact conclude that providing an adequate set of tokens described by a set of axioms and inference rules for a non-trivial process design language is a misguided research goal. The motivation for the analysis in chapter 5 resulted from an appreciation of the problems that arose when the technique described above is applied to anything other than nontrivial problem domains.

3.3.7 Intermediate Conclusions on Blackboard System Developments

The development of the interpreter for the physical properties module in Designer's Assistant constitutes, in representation terms, a considerable improvement on the use of simple production systems.

The concept of a number of communicating modules operating upon a common blackboard structure allows a large, complex problem like process engineering design to be potentially decomposed in a structured way.

The general control strategy proposed here is that a degree of "opportunism" be retained in the planner or scheduler at the top most level of the system. Large scale planning on a detailed level is not yet feasible and probably too restrictive in nature for a process design environment. Within a module planning is performed to a limited extent before control is passed to an enhanced rule interpreter. This limits the initial amount of search to be done. The rules themselves are classified at two levels, the property being modelled and the type of object acting as the focus of interest, in order to further minimise the amount of search. A facility to act upon meta-information on every rule condition interpreted is provided by the use of a "side-effect" mechanism. Another useful feature of the interpreter is the ability to make compromises if faced with absence of data when recommending a hypotheses.

In order to allow meaningful interpretation of hypotheses by different modules the concept of a module language definition was introduced. Every token or word referred to in a module must be defined explicitly in terms of the type of token, indicating its syntactic use, and its dependencies on the use of other tokens in the module. The use of explicit dependencies allows the interpreter to further control or focus any search effort.

In order to better structure problem information a number of basic objects, streams, units, etc, are used to partition the blackboard structure. Within the physical properties module itself a domain dependent compound class hierarchy is used to represent data in a structured way. Simple inheritance of properties in the hierarchy is allowed. This allows default information to be easily expressed and referred to in rules.

The use of a blackboard system allows the emphasis to be shifted from a purely search problem to one of combined search and model building in a natural way. As search proceeds, the model represented by the objects on the blackboard is gradually filled with detail as new hypotheses are deduced.

It appears, therefore, that a blackboard system architecture is a good candidate for further development of representation techniques needed in a process design environment. Areas of expertise can be represented in different modules rather than one enormous rule set. This has important consequences as regards practical development and maintenance of a system. Cross-communication between modules only appears to be meaningful if suitably sophisticated token definition and interpretation facilities are provided. The language definition mechanisms described here also allows procedural information and descriptive information to be handled together in an easy way. These facilities, when taken together, constitute the very minimum in representation techniques that will be required in a process design environment.

The problem of "arbitrary tokenism" was introduced in simple terms. This, in fact, turns out to be a fundamental issue and is one that the remaining chapters of this thesis explore in much greater detail.

Finally, the overall framework and intended operation of Designers's Assistant has only been briefly mentioned. It is more fully described in chapter 6 in the context of other developments.

3.4 Development of Object Orientated Representation Techniques in CLAP

3.4.1 Preface

This section introduces research done on the development and use of object orientated representation and programming techniques in a process engineering context. This work, the earlier development of the production rule system and the physical properties module of Designer's Assistant, covers the period 1984- 1986/7 and just preceded, therefore, the widespread availability and use of commercial object orientated software packages. Many of the features described below are now taken for granted as standard programming techniques.

The work is still relevant, however, as the system developed here offers a number of unique features and combination of programming styles not currently available in commercial packages.

3.4.2 Background

The limited use of a simple object hierarchy in the physical properties module of Designer's Assistant indicated that the use of more sophisticated object orientated techniques could be very useful in representing certain aspects of knowledge in a problem domain in a highly structural way. It was decided, therefore, to investigate these techniques more fully.

At the time, a number of programming languages had been developed, each based on the common idea of an "object" but offering different techniques for their manipulation, eg ACTORS(Lieberman, 1981), FLAVORS(Weinren and Moon 1981), LOOPS(Bobrow and Stefik, 1983), and SMALLTALK(Goldberg and Robson, 1983). The development of this type of language can be seen as continued development of the trend from the use of record types in block structured languages to abstract data types in languages such as Modula-2 and Ada. Lisp based systems drew heavily from the existing use of property-lists in that language. The basic concepts of object orientated programming are now well documented in the literature, eg Stefik,Bobrow et al(1983), Horn(1988), and are not discussed in detail here. It should be noted, however, that the terminology used in the literature varies considerably from system to system.

The following sections highlight the principal features of a Combined Logic And Procedures interpreter, CLAP. Further details of any features of CLAP can be found in Struthers(1987).

3.4.3 Generic Objects and Instances in CLAP

A generic object in CLAP is a definition of a general type or class of object. It is defined in terms of a number of attributes or slots. Typical generic objects might include a "plate heat exchanger", "centrifugal pump", etc. A specific exemplar of a generic object is called an instance of the object class, eg pump p101 might be an instance of a generic centrifugal pump object.

The following example indicates how a generic object representing a tank might be defined.

```
object(tank) :-
   self      - Name,
   variables - [Dp,Sp,Rp,V,I1,I2,Coords,Scale],
   slots     - [ design_pressure - Dp,
                 source_pressure - Sp,
                 relief_pressure - Rp,
                 volume - V
               ],
   relations - [ connected_to - (items - [I1,I2]),
                 modelled_by - FuncName
               ],
   display   - [draw_tank(Name,Coords,Scale)].
```

All generic objects are defined using this general format. The order of the component parts, "slots", "relations" etc is immaterial. The intended use of each component part of the overall structure is:

- "self-Name". Name is the variable or place holder for the name of a specific instance of a generic object. It is the token used to identify an instance, e.g. tank_101.
- "variables - List". List is simply a list of the Prolog variables used in the rest of the generic definition. It is only used when creating or referring to specific instances.
- "slots - ListOfSlotPairs". This is a list of the names of slots or attributes used to define a generic object. Each slot name has a variable or value place holder associated with it. In the simplest case the variable is a Prolog variable, as in the example definition for a tank, that appears in the variables list. More complex forms can be used however, see below. Each term in the list of slot pairs is of the form

  <slot_name> - <value_place_holders>
- "relations - ListOfRelations". List Of Relations is a list of the frequent or common relationships in which instances of this generic type will be involved. Each relation will be used to describe a relationship between two different objects. Each term in the List Of Relations is of the form

  <relation-name> - <value_place_holder>

  The value place holder can be a simple Prolog variable or something more complex, see connected_to entry in the definition of tank above.
- "display - ListOfInstructions". This term is used to define the sequence of calls needed to display any instance of a generic type on a display surface, eg terminal, graphics workstation etc.

Once a generic object or class has been defined any number of specific instances can be created. Instances can be created dynamically, i.e. on request, or can be

predefined in a data file. The latter technique is more useful for saving instances of objects for later re-use.

The CLAP interpreter has a command language to manipulate objects. Suppose, for example, that an instance of a tank is required in a process model and is to be called "t101". The following command is all that is required to create this instance -

$create_object tank-t101

Similarly, to delete this instance the command

$delete_object t101

is used. In this case there is no need to provide the generic type of t101 since object names must be unique identifiers.

An important point to note is that generic objects, or types or classes, need not be restricted to objects representing physical objects. More abstract objects such as a "graph", "flowsheet", "simulations-history", "reaction- pathway", etc, can be symbolically defined in exactly the same way.

Creating generic objects and instances is common to any object orientated (0.0.) system. CLAP offers two useful features, however, not found in other systems -

1. Use of Cross-Instantiation constraints
2. Use of generic data structures in object definitions

CLAP objects are defined as Prolog structures. This means that any Prolog variable occurring more than once in an object definition is subject to the automatic instantiation/unification algorithm of Prolog. This can be used to impose inter-slot or relation constraints very easily by simply referring to a variable in more than one slot or relation value place holder. As soon as the value changes in one slot the variable's value is automatically updated throughout the rest of the object definition. Consider the following part definition

```
slots - [inlet_temperature - (temperature - Ti $of Inlet),
        ......],
relations - [inlet_stream - Inlet]
```

The constraint being expressed is that the inlet temperature slot value depends on the current temperature of the inlet_stream object, Inlet, attached to this object. As soon as Inlet is instantiated, presumably, but not necessarily through the relations of the object, then Inlet is constrained to the same value elsewhere in the instance. On the other hand, if Ti is set to some value, 200 say, then the constraint is that the "temperature" slot of Inlet should also have a value of 200, or whatever the current value of Ti is. This is just one of several ways of specifying constraints in CLAP, but it is a particularly concise way embodied in the definition of the generic object itself.

The second useful feature of CLAP, namely the use of prespecified forms of data structure in slot or relation definitions, was illustrated in the previous example. The slot inlet_temperature used a predicate form in infix notation, in its definition rather than just a simple variable, X say, as its value place holder. In fact, any arbitrarily complex structure can be used in a slot or relation definition. Typical examples might include a binary tree structure, an acyclic graph structure, etc. In the previous example it was a term that can be used both as part of a CLAP command and as a prolog predicate.

A competent Prolog programmer can use this feature to great advantage. Consider a trivial example of a binary tree structure consisting of a top node, a piece of information describing the node, a left subtree and a right subtree where the subtrees are recursively defined in the same way. A slot definition might be

```
slots - [...
        binary_decision_tree - tree(Node,Info,Ltree,Rtree)
          ...
        ]
```

An instance of this object initially has an uninstantiated version of such a tree. Suppose at some point partial information has been gathered regarding the structure of this tree, namely the left subtree is of at least depth two and the second level leftmost node is labelled "configuration 1" with an item of information "cost = 500". (This structure might be used in a synthesis problem such as distillation column sequencing, see Struthers 1984). This partial information can be stored directly as follows

```
   Level 2 = tree (configuration 1, cost = 500, Llower, Rlower),
   Level 1 = tree (_, _, Level 2, _),
   Top     = tree (_, _, Level 1, _),
where,
   underscore "_" indicates undefined,
   "=" indicates unification
```

and then unifying Top with the generic data structure in the slot's value place holder, see setting slot values below. Working with partial information in this way is extremely useful in any problem involving a degree of search or planning in which information is gradually built up during the solution process. By defining the generic form of the data structure in advance, the system developer can impose one form of control over the allowed slot values, i.e. the syntactic form of any slot value.

The feature is not so much a feature of CLAP but of the Prolog interpreter itself, the latter being a very convenient tool for manipulating symbolic data structures and performing unification.

3.4.4 Setting and Retrieving Slot Values

At a superficial level, setting the value of a slot in a particular instance of an object is very simple. The appropriate CLAP command is

$set slot - Object Name - Slot Name - Slot Value

e.g. $set slot - t101 - design_pressure - 10.

Similarly, to check or retrieve the value of a slot the corresponding "$check" form is used

$check slot - Object Name - Slot Name - Slot Value

As indicated above, the value of a slot can be set to some partially instantiated data structure or other complex form if desired.

A useful structure that CLAP provides is a "dotted sequence". It is a sequence of values, perhaps uninstantiated variables, written as XI..XN. A generic stream object, for example, might have a components slot defined as

components - CI..CN

and defined in a particular instance as

$set slot - stream1 - components - methane..ethane..nitrogen

Rather than explicitly provide a sequence of length three when retrieving the value of this slot elsewhere in a program, the pattern matching features of CLAP/Prolog can be used to perform arbitrary forms of unification. For example, to simply retrieve the components use

$check slot - stream1 - components - Value

or perhaps

$check slot - stream1 - components - CI..CN

If the order of elements in a sequence has significance then a particular form of sequence can be tested for, eg

$check slot - stream 1 - components - methane..Others

Pattern matching facilities are taken for granted, therefore, in CLAP commands rather than being a special add-on feature. In practice, setting slot and relation values usually involves use of several other sophistications, as explained below.

3.4.5 Classes and Inheritance

A simple use of a class hierarchy and inheritance of properties by a sub-class from its parent, and all the parent's ancestors, was presented in the development of the compound class hierarchy in the physical properties module of Designer's Assistant.

Class or object hierarchies are fundamental to an O.O. representation. Each class in a hierarchy, at one level, is seen as a specialisation of its parent class, at the level above. Each class inherits all the slots or attributes that define its parent class and do

not have to be redefined throughout the hierarchy. Each sub-class has attributes, however, that its parent class does not have and this is what distinguishes a sub-class from its parent.

An hierarchy of object types is simply defined in CLAP by including a slot of the form

is_a - Parent Type

in the sub-class or generic child object definition. Consider, for example, the classes organic, alkanes, alkene and an instance of an alkane, methane. The relevant links might be

```
object(organic):-
   slots - [is_a - compound_class]...

object(alkane):-
   slots - [is_a - organic]...

object(alkene):-
   slots - [is_a - organic]...

$create_object alkane - methane, ...
```

Inheritance in CLAP is strictly intended to provide only class - sub-class relationships in an object or class hierarchy. In particular multiple inheritance, i.e. one too many sub-class parent links, is not provided due to reasons described later.

It is possible, however, to view a "leaf" class in a hierarchy tree as belonging to different parent classes at different times in CLAP. This can be useful in a complex reasoning problem when the class type of an instance is not yet known but only that the instance has certain desired attributes, i.e. an intensional specification. In that case, an object type called "null_type", say, can be defined with a variable value for its is_a slot, eg

```
object (null_type):-
   self - Name
   slots - [is_a - Whatever | Desired Attributes]
   ...
```

(NB: The term [Head | Tail] represents a list of terms in Prolog). The is_a type, Whatever, can be set dynamically in the same way as any other slot value. A simple example of the use of this technique is in separation synthesis problems in which types of separator must be chosen to perform a specified separation, i.e. the type or class of separator is initially unknown but one of its eventual attribute values is, namely the components to be separated. Care must be taken, however, to ensure that any globally dependent hypothesis must be deleted or modified when the parent class changes.

3.4.6 Describing Slots at a Meta-Level, Default Values

If slots were simply data structures contained within an outer data structure, the "object", then there would be little to commend the use of objects over the use of record types in a language such as Pascal. The difference between a good O.O. system and a poor one is the description of slots at a meta-level and the way in which that information can be used. CLAP provides a number of such meta-features.

The two simplest examples involve specifying the allowed "type" of the slot value and the range of allowable values. For example, a slot "design-pressure" might be restricted to be a positive number, i.e. of type "number", and allowed values > = O. This meta-information is written in CLAP as

```
slot(design_pressure - tank, InstanceName - P):-
   valrange - (P> = O),
   valtype  - number.
```

This states that there is a slot, design_pressure, defined in the generic class "tank" which can be referred to by value as P in any object instance of a tank. The "valtype" indicates that any value for the slot must be a number. In fact any predicate name indicating the class or data type can be used, eg binary_tree, alkane, heat_exchanger etc. This is useful when the names of object instances are used as slot values and it is necessary to check the class of the object instance.

The "valrange" is any evaluable expression. It might be a simple numerical expression or a choice of words or a complex logical formula to be proved in Prolog. In the latter case the formula may simply involve predicates to be checked against those in working memory or it may involve clauses that require considerable search to conclude the proof. The form of checking slot values can, therefore, be very sophisticated in CLAP if necessary.

The next form of slot meta-information in CLAP is specification of default slot values. The notion of a default value is, as indicated earlier in the context of MECHO, a superficially simple idea : if a request is made for a value of an, as yet, undefined slot then use the default slot value provided. A simple example might be to assume ideal gas behaviour for thermodynamic models of alkanes in the absence of detailed problem conditions. The significant problem that arises is ensuring that the status of the resulting fact, i.e. that it is an assumption, and any other dependent hypotheses generated from it are properly maintained by any theorem prover, eg if it is subsequently found that the slot value contradicts the default slot value. A good introduction to reasoning with default logic and the problems that arise is Reiter (1978).

In CLAP, default information can be specified in one of two ways. The simplest way is to provide a specific value, number, word, data structure etc. This is specified as

```
slot(<slot> - <class>, <instance> - <value>:-
   valrange ...,
   valtype  ...,
   default_value - <default_value_to_use>
```

It is sometimes the case that there is not a single default value but rather a default function that returns a default value depending on the arguments used and the problem context. In CLAP a default "function" can be a simple or complex clause, eg a simple database look up, a procedural calculation, or a call to a theorem prover or a production system. In each case the default entry is written as

default_value - predicate_name(Arg1, Arg2,..., ArgN),

the value being returned in the last argument ArgN. It is worth noting that just as slots are inherited by sub-classes then so too is the valtype, valrange and default_value meta-information.

There are three more sophisticated forms of meta-information that can be specified in CLAP. These are:

1. The use of meta-view points.
2. Control Information
3. User specified information

and are now treated briefly in turn.

3.4.7 Meta-View Descriptions of Slots

A meta-view in CLAP is used to enable alternative view points of a slot value to be handled at the same time without creating multiple instances of an object to represent each view point. A trivial use of a view point is accessing a slot value representing a physical property in different units. Each set of units is considered as an alternative "view" on the slot value. A more complex use might be modifying a plant item cost estimate given an "optimistic" or "pessimistic" view point on the inflation forecast. There is no restriction on what constitutes a "point of view" but the feature should be applied carefully as it introduces similar problems to those resulting from using default values, ie maintaining the view point held in subsequent, dependent hypotheses.

The definition of a meta-view in CLAP is straightforward and is of the general form

```
meta_view(<view_point> - <slot_name> - <object_class>,
                         <instance_name> - <slot_value>) :-
   meta_value      - CurrentViewPoint,
   modifying_to    - NewViewPoint,
   default         - DefaultViewPoint,
   modifiers       - [ViewPoints1 - CommonCall1 | OtherModifiers]
```

For example, a view point of "units" for the slot "cost" in object class "plant item" might have the definition

```
meta_view(units - cost - plant_item, Item - Cost) :-
   meta_value       - Current,
   modify_to        - Required,
   default          - pounds_sterling,
   modifiers        - [[dollars, pound_sterling, dm, yen]
                        - convert_currency,
                       [Current, value_of_the_year(Current)]
                        - inflated_cost]
```

There are a number of points to note here

1. The view point applies to <slot_name> in any instance that is a sub-class of <object_class> in the definition. If <object_class> is left undefined then the view point can be used in any object class. If the <slot_name> is left undefined then the view point applies to any slot defined in the <object_class>.
2. Meta-view descriptions are inherited along with their slots.
3. A default view point can be specified either as a specific value or as a function to call.
4. The modifiers entry is a list of pairs, Views - ClauseName, in which ClauseName is a clause/procedure that can be called to convert the value of the slot from the Current view point to the Required view point. Both Current and Required must appear in the sub-list Views.

Working with meta-view points is a simple extension of the standard commands to set or check slot values. To set the value of a slot, temperature say, according to some view points, eg units of "Deg.C", the form

[$set slot - stream1 - temperature - 200 @ units - "Deg.C"]

is used. Meta-view points are specified after the @ symbol. Similarly, a meta-view point can be specified when retrieving a slot value.

$check slot - stream1 - temperature - T @ units - "Kelvin"

This view point may be different from the one used in setting the slot value. Alternatively, if the current meta-view point is not known it can be determined by using a variable instead of a specific value, eg

$check slot - stream1 - temperature - T @ units - UnitsUsed

In this case the variable UnitsUsed gets unified with the units currently in use, perhaps the default value for the units.

Finally, a slot value can have multiple view points associated with it at once, ie the overall meta-view point is a conjunction of individual view points. Individual values for a view point, eg the allowed types of units, are assumed to be disjunctive while view points themselves are assumed to be orthogonal in nature and non-contradictory. A simple example of multiple view points might be a slot representing the estimated cost of a plant item. One view point could be the monetary units used while another would be the rate of inflate assumed when estimating this value. This could be specified by

```
$set slot - column5 - estimated_cost - 2000
     @ units-dollars @ inflation-percent(10)
```

View points are also useful for eliminating "arbitrary" slots such as "estimated_cost", "real_cost", etc. A single slot "cost" can be used with a view point "status", say, which takes on the values "estimated", "real" etc.

Once a multiple view point has been specified it can be retrieved either in its entirety or from each constituent view point. In the previous example, for instance, both the units and rate of inflation can be obtained from

$check slot - column5 - estimated_cost - C @ MetasUsed

or the units, say, could be retrieved individually by

$check slot - column5 - estimated_cost - C @ units-U.

The principle use of meta-view points is to allow dependency information to be set and accessed in an easy way. The dependency information may itself be quite complicated, such as a list of assumed hypotheses upon which the fact, represented by the slot value, is dependent. The definition of these meta- dependencies is completely decoupled, however, from the definition of an object or simple restriction on slot values and type. Object definitions can therefore be gradually built up, extra sophistication being added when necessary.

3.4.8 Embedding Control Information and Procedural Attachment

An important part of the representation formalisms presented so far has been the means by which control information can be expressed and attached procedures can be evaluated to simplify and reduce the search effort required in any problem solution.

A standard way of expressing such information in O.O. programming is by the use of "demons" attached to a slot in an object definition. Demons are fired, or run, whenever the value in a slot changes. In CLAP there are two types of demon. A "before_demon" is used to perform some action just before the value in a slot gets changed. The action will typically involve checking the current values of other slots in some other object for consistency or informing other objects of the imminent change in slot value, eg message passing below. Conversely, an "after_demon" is used to perform some action immediately after the value of a slot has been changed. Once before or after demons have been declared there is no need to call them explicitly. They will automatically be called by the CLAP interpreter when required. Demons are declared in CLAP by adding entries of the form

```
before_demon - ProcedureNameB,
after_demon  - ProcedureNameA,
```

to the slot definition. The arguments of the procedures include object information,

the previous slot value, the new slot value and any meta-view points being used. Also, before and after demons are inherited in the same way as other meta-slot information.

It is very simple, therefore, to attach procedures to slots of objects using this mechanism. If changing a slot's value is to influence the control strategy of some search method causing the change then the demon must have access to the temporary data structures of that method in order to make any useful recommendations, eg other sub-goals to investigate etc. Since demon procedure definitions will be defined independently on any search algorithm that intends to make implicit use of them some global structure, eg a stock of current search methods, must be maintained to allow the demons to take appropriate action. This can be achieved in a particularly convenient way in CLAP. Multiple definitions of a demon can be provided if implemented as Prolog clauses, each definition being applicable to different situations. When the demon is fired CLAP will back-track through these definitions until the appropriate one is found and evaluated. Figure 3.9 shows an extract of source code used to implement a before_demon named "check_on_vessel_material". The demon is used to check the compatibility of a vessel's material of construction when the value of the vessel's fluids composition slot is changed, e.g. a trace component may be introduced which is highly corrosive and it is necessary to alert the design engineer to this problem.

Figure 3.9: Example of a High Level Demon in CLAP

3.9a: Typical Call to Deductive Database Type Search and Production System

```
check_on_vessel_material(Vessel_type, Vessel_name,
                         OldFluid, NewFluid) :-
   $check slot-Vessel_name-( material - CurrentMaterial),
   database_lookup(compatible_material, NewFluid,
                           CurrentMaterial ,Reply),

   $if Reply = okay $then return
     $else check_material_selection(NewFluid,
                                    CurrentMaterial,Vessel_name).
```

3.9b: Simple Implementation of a "Rule" in a Deductive Database

```
database_lookup(compatible_material,Fluid,Material,okay) :-
   not( incompatible_material(Fluid,Material) ).
```

3.9c: Typical Database Entry in Materials Database

```
incompatible_material(H2SO4(aqueous), carbon_steel).
incompatible_material(HNO3(aqueous), carbon_steel).
```

3.9d: Simple Interface to Production Rule System

```
check_material_selection(Fluid,Material,Vessel_name) :-
   set_up_prodtn_system(Fluid,corrosion),
   run_prodtn_system(material,Fluid, Options ...),
```

```
check_with_user(Fluid,Material,Options,Vessel_name).
```

The top level code in Fig 3.9.a first checks the current material of construction. It then performs a deductive proof in Prolog to confirm that the material is compatible with the new fluid composition. The search is performed over a materials database consisting of both facts and inference rules. This is called a "deductive database" in the literature. If this search fails or proves inconclusive then the top level code calls a rule based production system to suggest a new material of construction to use, see Figure 3.9.d. The production system may reside elsewhere, eg in another module in Designer's Assistant.

Sophisticated forms of procedural attachment, therefore, can be easily performed within CLAP using the demon mechanism.

3.4.9 Other forms of User Specified Meta-Information, Arbitrary Tokenism Revisited

The problem of arbitrary tokenism resulting from the use of if-then rule sets between communicating modules in Designer's Assistant was introduced in section 3.3.6. The same problem re-emerges in object orientated systems except the arbitrary tokens here are the names and values of slots in objects, eg one object in module A uses a slot "pressure" with value "high" which should affect the value of a slot called "working-pressure" in module B, or even in another object in module A, since hypotheses or facts are now implicitly defined in the object data structure. An object O with a slot S and value V might be re- expressed as

S of O is V

in another rule based module that has been called upon to make some recommendation regarding the value of V, e.g. the material of construction of a vessel in the previous example.

The solution described earlier was to require that each token be suitably defined in terms of other dependencies to help circumvent this interpretation problem. The idea of token definitions can be extended to slots in objects by simply extending the slot definition clauses, e.g. the "requirements - Dependencies" term can be added to the slot definition in exactly the same way as for valtype, valrange etc.

In order to make use of this information, however, the system developer must provide an extended definition of the built-in interpreter clause used to set or access slot values. This is most easily achieved by setting the appropriate user option in CLAP to indicate the name of the new clause to call. This alternative definition can simply embed the original call within the extra clauses needed to ensure self consistent use of tokens.

It is a simple matter then to experiment with different interpreters used to maintain and manipulate dependency information. The program developer is free to specify whatever additional information is necessary for a particular interpreter in the slot definition clause. There are no restrictions on the contents of this clause in CLAP. If the default interpreter is used this extra information will be ignored.

3.4.10 Multiple Contexts or Worlds in CLAP

An extremely useful programming technique in any sort of search or synthesis problem is the ability to create possible contexts or worlds in which alternative situations or hypotheses can be explored. CLAP provides such a facility by allowing multiple contexts to be created. An instance of an object class can have different slot values and relations in each context representing a different design alternative.

There are a number of simple commands to use contexts in CLAP:

1. $create_empty_context ContextName , creates a new context to work with.
2. $use ContextName , to jump into a Context performing any subsequent commands locally within that Context.
3. $reset ContextName , to clear out a context.
4. $create_copy_context NewContext , saves a copy of the state of the current context into a new context.
5. $copy_object Class - Instance $from Context1 $to Context2 , to copy an object instance from one context to another.

In addition, any other CLAP command can be run or evaluated in a particular context by using the general form

<clap command> $in Context

It is very simple to consider design alternatives by first creating a number of independent contexts and then evaluating each alternative in a different context. A top level clause can then compare instances of objects in each context and make some appropriate recommendation.

The difference between contexts and meta-view points is that the former is used to represent distinct alternatives at a coarse-grain level while the latter represents fine-grain variations on the slot/relation values of a single object instance.

3.4.11 Building Process Models from Relationships in CLAP

In the introduction to objects in CLAP it was indicated that an object can be involved in a number of relationships. Relationships in CLAP can be created at one of two possible levels:

1. simple symbolic relationships between related objects
2. mathematical constraints used to model objects

The first type of relation is only for use in simple situations. To use a relation of this type simply define a relation name and associated data structure in the generic object definition. The data structure is used to store information concerning a specific instance of a relation. A vessel object might have predefined relationships "inlet-streams" and "outlet-streams" amongst others:

```
object (vessel):-
   self       - S
   slots      - [....],
   relations  - [inlet_streams - InS1 .. InSN,
                 outlet_stream - OutS1 .. OutSN, ...]
```

The principal difference between this kind of a relation and a slot is that relations refer directly to other object instances by name. In this case the inlet and outlet streams would be the names of distinct objects, as opposed to data structures to hold slot values.

In a class hierarchy "static" relations of this type are inherited by sub- classes. The "values" of these relations are set and accessed in a similar way to slot values, ie

```
$set relation - vessel1 - inlet_streams - s1 .. s2 .. s3

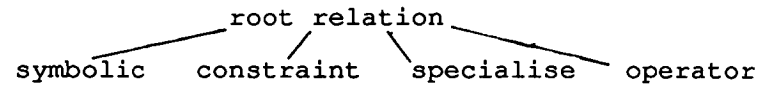
$check relation - vessel1 - outlet_streams - Outlets
```

The term "static relations" is used since the relations are automatically defined for each instance of an object irrespective of whether any relation "values" are known. This is useful for defining very common relationships that an object will usually be involved in, eg the connectivity relationships between a plant item and its neighbouring plant item. The relationships are said to be "symbolic" in nature in that they define some arbitrary relationship which does not directly generate mathematical constraints to help model an object. They may of course be used to generate constraints indirectly by other relationships that depend on their value. For example once the inlet and outlet streams are known there is no need to directly generate an overall mass balance (constraint) for the vessel unless there is a need for it. The mass balance relationship, however, cannot be formulated until the inlet and outlet streams are known.

The second type of relationship in CLAP is concerned directly with the specification of models of process items in terms of sets of different types of equation : algebraic, differential, etc.

Relationships of this type are defined in a relation class hierarchy just as object types are defined in an object class hierarchy. There is a root relation at the top of this hierarchy which has four basic sub-classes, namely symbolic, constraint, specialise and operator, see Figure 3.10

Figure 3.10 The Relation Hierarchy In CLAP



Each sub-class is an "is_a" class of the root_relation. It is possible to create other sub-classes of the root_relation if the default sub-classes are not sufficient. The default sub-classes are now described in turn.

3.4.12 Symbolic Relationships, Demons and Meta-View Points

A symbolic relationship in the relation hierarchy is simply a more rigorous definition of the simple static relationships described above. Relationships are defined independently of any object, ie they need not appear in the static relations list in an object definition but can be used at a later point to relate two appropriate objects.

The general form of a relation definition is:

```
relation (RelationName, DomainName - RangeName):-
   variable      - [ ... ],
   domain        - DomainClass,
   range         - RangeClass,
   default       - DefaultEntry
   before_demon  - Bdemon,
   after_demon   - Ademon,
   active_code   - Code,
   slots         - [is_a - symbolic, ...].
```

A binary relationship, i.e. one-to-one, is defined between an instance of a DomainClass object, DomainName, and an instance of a RangeClass object, RangeName. The class types are checked whenever a relationship is asserted between two object instances.

The default entry is only used when it is appropriate to generate some default object instance to use as the range of the relation. Default entries are either specific values or functions to call, as in slot definitions. This entry is optional and can be omitted.

The before and after demons work in exactly the same way as when setting slot values. The before demon is run immediately before a relation is asserted and the after demon immediately after it is asserted.

The active code entry is where any embedded control information and attached procedures should be defined. The code can be any sequence of CLAP commands or calls. This entry is primarily used by relations in the "operator" sub-class.

The slots entry defines any further information concerning the relation. This is where any user meta-information would be stored such as inter-relation dependencies etc.

The programmer is free to interpret relations using specialised inference techniques rather than the default method. This would be the case if the tokens used for relationships were further specified by a vocabulary definition such as in the physical properties module of Designer's Assistant.

Meta-view points of generic relationships can be defined in exactly the same way as meta-view points for slots. This is a more subtle concept in that the information being represented is particular view points of a relationship. Consider the relationship "distillation_column_5 was designed_by Fred". Useful meta-view points could be the status of the column design from a materials point of view, or a safety point of view, or a senior engineer's point of view, etc.

The general form of a meta-view point of a relation is defined as:

```
meta_view(MetaName-RelClass-DomainClass,DomainName-RangeName) :-
   meta_value    - CurrentViewPoint,
   modify_to     - RequiredViewPoint,
   default       - DefaultViewPoint or Function,
   modifiers     - ListOfPredicates.
```

The individual entries were explained in section 3.4.7 earlier. Meta-view points are inherited by relations in the same way as for objects, i.e. if relation R is a subclass of RelClass and object D is a subclass of DomainClass then the meta- view definitions for RelClass still apply when an instance of D is involved in a relationship R.

Finally, an important aspect of any relation is the definition of its transitivity in terms of other relations. For instance, the grandfather_of(G,C) relation is true if "parent_of(G,P) and parent_of(P,C)". The definition and use of transitivity and other properties of relations is left to the programmer. The slots entry in a relation definition can be used to define this information.

3.4.13 Establishing a Symbolic Relationship and its Inverse

The default method of establishing a relationship is simply to use the "$set" CLAP command, as in

$set relation - vessel1 - upstream_of - pump0

The domain object is vessel1, the range object is pump0 and the relationship is upstream_of. When this call is made a note of the relationship is stored in the relations list of object vessel1. The interpreter automatically handles any demons, active code, meta-view points etc. It then looks to see if there is a term of the form

[inverse - InverseRelation]

in the definition of the relation. In the example above the inverse relation might be "downstream_of". If this entry exists then the relationship -

$set relation - pumpO - downstream_of - vessel1,

is set. This in turn evaluates any demons etc for the downstream_of relation and makes a note of the relation in the relations list of object pump0. The process then stops as an infinite loop would otherwise occur. If an inverse entry cannot be found then a default inverse link is created. In this example the inverse relation would be called "upstream_of_inverse" and this would be noted in the relations list of pump0.

It is important to always maintain an inverse or two-way link between objects irrespective of whether the inverse relationship is defined or not. These two- way links are required to ensure that relationships can be consistently maintained when changes occur to either of the range or domain objects. For instance, if an object is deleted then all the instances of relationships which use this object as either the range or domain object must also be deleted, in one-to-one relationships, or modified, in many-to-one relationships. If the deleted object is the "one" in a many-to-one relationship then all of the many- to-one links must be deleted.

3.4.14 Constraint Relationships

From a model building point of view, constraint relationships are more important than symbolic relationships since it is the classes of constraint definitions that will define the equations used in a model of an object.

A constraint relationship is used to represent mathematical constraints of any type. It is recommended, but not obligatory, that the constraint class be sub- divided into equalities and inequalities and the equalities class be further divided into the types of equation, eg algebraic, differential, difference, etc.

The definition of an algebraic constraint is best introduced by a simple example. Consider the definition of an overall mass balance for a process unit. The unit will have a number of inlet streams, I, and outlet streams, O, defined perhaps as simple static relationships or symbolic relationships. There are two essential parts of a generic constraint definition:

1. The "return_form" - ie the generic form of the equation
2. The "bindings" - the specific variable bindings in an instance of the relationship used to create a specific form of the generic constraint

Here is the definition of an "overall_mass_balance" relationship -

```
relation(overall_mass_balance,Unit-Val) :-
   domain       - _,
   variables    - [Unit,Form,Bindings],
   bindings     - [ I = inlets $of Unit,
                    O = outlets $of Unit],
```

```
   active_code - [],
   return_form - (sum_of(mass_flowrate $of I, $over I)
                  = sum_of(mass_flowrate $of O, $over O)),
   return_type - equation,
   slots       - [is_a - constraint].
```

For this definition to work, the inlets and outlets must be defined as a dotted sequence or list of stream names, e.g. s1..s2..s3 . The return_form is simply the equality between the sum of the mass flowrates of the inlets and those of the outlets. In order to use the mass balance constraint the "specialise" class of relations must be described.

3.4.14.1 Specialise Relations

A "specialise" relation is required to generate a specific instance of a generic constraint such as the "overall_mass_balance" constraint. Consider a unit, u1, which has two inlet streams and one outlet stream. The following command creates an instance of the overall mass balance relations:

$set relation - u1 - has_overall_mass_balance - InstanceName

What happens is that a specific instance of the constraint is generated and given an InstanceName, say over_mass_balance2. This instance is considered to be the range of the "has_overall_mass_balance" relationship with unit u1 as the domain object. The return_form generated is -

F1 + F2 = F3

and the bindings generated are

```
F1 = mass_flowrate $of s1 -> Value1,
F2 = mass_flowrate $of s2 -> Value2,
F3 = mass_flowrate $of s3 -> Value3,
```

This information is stored in the instance of the constraint. The constraint instance is an object, just like any other object in CLAP. There are several points to note:

1. The constraint is not evaluated in any sense. The constraint instance only contains the specific form of the constraint given the current set of bindings.
2. A name for the constraint instance can be provided rather than using a CLAP generated name.
3. The relation "has_<Constraint>" is the default way of generating instances of constraints. This saves having to define different specialised relations for every generic constraint

In relation to point 3 it is perfectly allowable for a programmer to define their own specialised relations. Here is the outline definition for "has_overall_mass_balance" if explicitly defined:

```
relation(has_overall_mass_balance,Unit-Val) :-
   domain      - _,
   range       - overall_mass_balance,
   variables   - [Unit,_,_],
   slots       - [is_a - specialise],
   meta_slots  - [form_bindings -_,],
   active_code - [... do_your_own_thing(Unit,Val) ...].
```

The range of this relation indicates the class of constraint that is to be specialised. The active code entry would perform any special actions required. By the time this code gets called, CLAP will have created the default version of the constraint instance. This can then be modified at will by the active code. In most cases, however, the default behaviour is adequate.

In summary, when a specialised relation is asserted, using $set in the example command above, three things happen:

1. A specialised instance of a constraint is generated with appropriate form and bindings.
2. The specialised relation, eg has_overall_mass_balance, is noted in the relations list of the domain object, eg unit u1.
3. The inverse of the specialise relation is noted in the relations list of the range object, ie the constraint instance generated in 1).

It is most important to note that the implications of the constraint are not evaluated in any way, if trying to solve the mass balance and updating slot values in objects. This allows the existence and form of a relationship to be reasoned about without "believing" or "evaluating" any of the consequences of that relationship. This is a very useful feature of CLAP, termed here as "delayed evaluation of consequences". A typical use would be to allow the degrees of freedom of a set of constraints to be checked before any attempt was made to evaluate or solve the constraints to ensure that a model was not under or over specified.

3.4.14.2 Design Specifications

In any design problem certain specifications will be imposed by the engineer. Imposing values for "design variables" sets implicit equality constraints between the supplied value for a quantity and the defining expression(s) for that quantity. Consider the generic constraint "heat_content" which is defined as follows for a "stream" domain object

```
relation(heat_content,Stream-Val) :-
   domain      - stream,
   variables   - [Stream,Form,Bindings],
   bindings    - [],
   return_form - ( (temperature $of Stream)*
                   (mass_flowrate $of Stream)*
                   (specific_heat $of Stream)
                 ),
   return_type - expression,
   slots       - [is_a - constraint],
   active_code - [].
```

Suppose that in a design problem the heat_content of a particular stream, stream1, is specified to be 1000, in whatever units (the units could be given by a meta-view point). This is done by calling the following clause:

set_user_spec (heat_content $of stream1, 1000, Error g)

All this does is specialise the constraint as before, but the "form and bindings" in the constraint instance are set to -

```
Form     = Spec
Bindings = [Spec = heat_content $ of stream1 -> 1000]
```

This effectively overrides the definition of the generic expression for heat_content. There is still an implicit equality constraint, however, between the value 1000 and the specific form of the constraint, ie

```
Form     = T*M*Cp
Bindings = [T  = temperature $f stream1 -> Tvalue,
            M  = mass_flowrate $of stream1 -> Mvalue,
            Cp = specific_heat $of stream1 -> Cpvalue]
```

These implicit constraints are recorded by CLAP when the user supplies design specifications. They can be retrieved at a later point if necessary, as explained below, in the evaluation of constraints. This is useful when checking the consistency of user supplied design specifications.

3.4.14.3 Operator Relations, Evaluating Constraints

The final class of default relation types are referred to as "operator" relations. An operator relation is in fact a function that takes a specialised constraint instance and creates an evaluated form of that instance as the range of the relation. The specialised constraint and the evaluated constraint are two distinct objects. This is an important aspect of "delayed evaluation" in CLAP. A generic specialised form of a constraint for a given object need only be formulated once. Thereafter, different instances of the evaluated constraint can be created under different conditions and compared, if desired, by some inference technique.

This means that it is very easy to record different solutions for each step of an iteration loop, or single time steps in a dynamic simulation.

An operator relation can be defined in exactly the same way as a constraint or specialised relation. There is a default operator relation, however, to handle all specialised constraints just as there is a default way to specialise any generic constraint. It is invoked by the call

$set relation-Object-perform_<ConstraintName> - NewName $in World,

e.g.

$set relation-vessel1-perform_overall_mass_balance-Name1

where overall_mass_balance is a generic constraint that has been specialised for vessel1. A value will be generated for the Name of the evaluated instance.

All the default operator does is to establish the current bindings of a constraint instance, ie access the various slots or relations referred to in the Bindings part of an instance, and then call the predicate "manipulate_constraint_form" -

manipulate_constraint_form(DomainName,ConstraintType,Form,Bindings)
When the binding values are retrieved they are unified with the corresponding variable in the Form part of the instance, ie the expression or equation part expressing the constraint. It is this constraint Form that is passed as the third argument to manipulate_constraint_form.

The clause manipulate_constraint_form can be multiply defined, by the programmer, to handle the different classes of ConstraintType to be evaluated. This clause will typically interface to an equation solving package or problem specific solution methods. It is this clause that performs any evaluation of values.

The only complication to this process is how binding values are established when they refer to other constraint relations instead of slot values. Consider the following definitions of a heat_content expression and an overall heat balance relation.

```
relation(overall_heat_balance,Unit-_) :-
   domain         - _,
   variables      - [Unit,Form,Bindings],
   bindings       - [ I = inlets $of Unit,
                      O = outlets $of Unit],
   active_code    - [],
   return_form    - ( sum_of(heat_content $of I, $over I)
                       = sum_of(heat_content $of O, $over O)
                    ),
   return_type    - equation,
   slots          - [is_a - constraint].
```

The heat balance relation refers to the sum of the heat contents of the input and output streams. Consider a unit with two inlet streams and one outlet stream. The top level Form and Bindings are:

```
   Form           H1 + H2 = H3
   Bindings       H1 = heat_content $of stream1 -> H1Value
                  H2 = heat_content $of stream2 -> H2Value
                  H3 = heat_content $of stream3 -> H3value
```

When "perform_overall_heat_balance" is called with this constraint instance it checks to see if the Bindings are slot values or further constraints. If a binding expression is a reference to a slot then the current slot value is looked up and unified with the appropriate variable in Form.

If a binding expression is a reference to a constraint, eg H1, that has been set as a "user-spec" then the value of the design specification is unified with the appropriate variable in Form. For example, if the user had set the heat_content of stream1 to 1000 then H1 becomes 1000 in Form.

If a binding expression is a reference to a constraint that is not a "user_spec" then the expression for that constraint is retrieved and added to the Bindings list. For example, the expression for H2 is expanded to

```
H2        = heat_content $of stream1 -> H2Value
H2Value   = T2*M2*Cp2 -> ExpValue
T2        = temperature $of stream1 -> T2Value,
M2        = mass_flowrate $of stream1 -> M2Value,
Cp2       = specific_heat 4of stream 1 -> Cp2Value
```

If any of the expansions referred to further constraints then these too would be expanded in a recursive fashion.

When the default "perform_" relation is used all constraints are expanded or unwound recursively and a complete set of bindings is generated. Unwinding stops when a slot value or "user_spec" is encountered.

When a "user_spec" value for a constraint is provided there still remain implicit expansions that must eventually equate to the user provided value. CLAP does in fact generate these implicit expansions but stores the bindings in a local work area. These implicit expansions can be picked up later, eg by an equation solver, as shown later.

When an expression is expanded, eg heat_content, CLAP creates an object representing the specialised form of the constraint and notes this relation in the domain unit's relation list, eg the "has_heat_content' relation will be recorded along with the name of the constraint instance.

In complex situations the default "perform_" operator relation can be replaced by a user defined operator type relation. These relations are defined in the same way as specialise or symbolic relations except that the "active_code" entry must specify how to evaluate a constraint instance. The general format is

```
relation (RelationName, DomainName - EvaluatedInstanceName):-
   domain           - DomainType,
   range            - ConstraintType,
   variables        - [Domain,Form,Bindings,...],
   constraint_form  - ConstraintToUse,
   active_code      - SequenceOfClapCalls,
   reset_bindings   - Updates,
   slots            - Slots.
```

If the active code was simply a call to "manipulate_constraint_form" then the effect would be the same as using the default perform relation.

The entry "constraint_form" specifies the call to make to pick up the specialised relation that is to be evaluated. For instance, if the constraint type was "overall_mass_balance", the default constraint form to use is

has_overall_mass_balance @ (form_bindings - $all)

The meta-view point form_bindings is predefined for any specialised or operator relation and specifies the depth to which any embedded constraints should be

expanded, as described above. The value "$all" indicates that all constraints be unwound down to slot values of objects. If a value of "level(N)" is used, where N is some integer, then the process of unwinding constraints is only attempted N times. If an expression is fully unwound, i.e. down to slot level, before N levels occur then the system continues without error.

To evaluate a specialised relation then, the following steps occur

1. The specialised relation in the constraint_form entry is checked for in the Domain's relation list. The name of the specialised constraint instance is found and its Form and Bindings are retrieved. These values are unified with the Form and Bindings variables in the variables list of the operator relation once the Bindings have been expanded to the appropriate depth.
2. The sequence of calls in the active_code entry is then evaluated. The calls can be any CLAP command and will presumably involve Form and Bindings as arguments to these calls.
3. Any Bindings that have had their values changed as a result of running the active code should be updated, i.e. reset any slot values, relations etc.

A built in mechanism is provided to pick up any changes in variable bindings so that the Bindings in step 3) can be reset automatically. All that is done is to use the following general form for any active code:

```
active_code      - (save_new_bindings (Bindings, CheckLater),
                      ...other calls ...,
                    save_new_bindings (CheckLater, Updates),
reset_bindings   - Updates
```

The first call to save_new_bindings detects which Bindings could be reset. The last call checks which of these were changed in the active code. The binding expressions to be changed are left in the list Updates. When the active code has run, CLAP looks for a "reset_bindings" entry in the relation definition. If it has been defined as shown here then the Updates list will be unified with the list of bindings to be updated. CLAP then resets these entries automatically.

If no reset_bindings entry exists then no updating of bindings will take place unless they are performed in the active code itself.

If the reset_bindings entry is used in conjunction with save_new_bindings then no action is required by the programmer to complete step 3) above. All binding values will be automatically reset by CLAP when necessary.

Finally, it was stated earlier that any implicit binding expansions resulting from the use of "user_specs" are also generated by CLAP. To retrieve these bindings in the active_code first use the call meta_expand_stack(Key) Key is an index into an area of Prolog's internal database. The set of implicit bindings are recorded under this key in predicates of the form:

```
hidden_reference(Context, DomainName, ConstraintName,
                 ConstraintType, Level, IForm, I Bindings)
```

IForm and IBindings are the forms that would have been generated if ConstraintType had been expanded to a depth Level for the object DomainName. The user_spec versions of Form and Bindings are stored in the instance of the specialised relation just as for any other specialised form of a relation. Implicit expansions are only performed once for a particular object and constraint. Thereafter, CLAP checks the meta_expand_stack entries for any subsequent references to the same constraint.

When evaluating sets of equations it is always advisable to check the meta_expand_stack before any other calls in the active_code of an operator relation. It is then the responsibility of the active code to handle or check these implicit bindings. In simple applications they can be entirely ignored.

3.4.15 Summary of Development of Equation Based Process Models

Relations in CLAP provide a simple but powerful way of expressing a "model" of a process unit. Relational information primarily expresses constraints between different objects. In a design problem these objects will typically represent process units or streams. The relation class hierarchy can be basically split into those types of relations that do not have equations associated with them, called symbolic relations, and those that do. Equation type constraints can be one of three predefined classes:

1. Constraint - defines the generic form of a constraint, expression, equation, etc., for any valid domain object. For example, the generic definition of a mass balance for any process unit.
2. Specialise - These relations create a specific form of a generic constraint for a particular unit. All the variable bindings are left uninstantiated so that the same specialised form can be re-evaluated under different conditions, binding values etc. The specialised form or constraint instance is an object in its own right. Its name can be user specified or generated by CLAP. There is a default mechanism to specialise any constraint. This can be overridden by a user supplied definition. Design specifications cause a constraint instance to refer to user specified values rather than the defining symbolic equation.
3. Operator - Operator relations take a specialised form of a constraint and manipulate or evaluate that form in some user specified way. The constraint form is typically an equation or set of equations in symbolic form. A suitable interface to an equation solving package is needed to solve these equations.

A process "model" of a unit can be thought of as a combination of the unit's slot values, its slot descriptors and meta_view points, the symbolic relations its involved in and any equations that describe it that are stored as specialised constraints.

Process models can be gradually built up by simply asserting new relations whenever necessary. High level relations are easily defined since they can simply involve combinations of lower level relations. The "combination" might just be a set, as in a

set of equations, or it might be equation or inequality or an expression. The following definitions show how to define a "heat_and_mass_balance" relation as a set of two equations. These equations are further defined by other relations, ie the heat balance and the mass balance, which in the case of the heat balance is further defined in terms of a heat content relation.

```
relation(heat_and_mass_balance,Unit-Val) :-
    domain      - _,
    variables   - [Unit,Form,Bindings],
    bindings    - [],
    active_code - [],
    return_form - [overall_heat_balance $of Unit,
                   overall_mass_balance $of Unit],
    return_type - (list $of equations),
    slots       - [is_a - constraint].
```

It is very simple to build up useful high level relations that are defined in terms of other lower level relations.

CLAP has a built in facility to automatically generate all the constituent relationship bindings when evaluating or checking the consistency of a set of process model equations. In the case of "standard" sets of equations to describe a unit, a single high level relation could be asserted that implicitly generates all the specialised forms of the constituent equations. For example, the heat_and_mass_balance relation could be part of a higher level relation "standard_flash_model". The single assertion

$set relation - flash_vessel10 - has_standard_flash_model -

implicitly sets up all the "standard" equations to describe flash_vessel10. There is nothing to stop the engineer asserting additional relations that augment this set of equations. Alternatively, the engineer may incrementally build the flash model up one equation at a time. Model building, therefore, can be done in a very flexible way in CLAP.

A set of equations defined by some relation can be locally evaluated by simply calling the default "perform_<constraints>" mechanism, providing a suitable interface to an equation solver is available. Hutton(1990) describes one such interface to CLAP. Such a facility makes it straightforward to evaluate "partial" process models in isolation as a design proceeds.
When a constraint instance is evaluated, a separate object is created to represent the evaluated form of the constraint. This makes it simple to compare case studies in different worlds or contexts.

Finally, operator relations are not intended to be the principal means of solving problems. Very high level relations could be defined but this was not the original

intention. Section 3.4.20 introduces a more procedural/algorithmic framework in which individual relations can be asserted and analysed within general design procedures for well know problems, e.g. solving flash calculations, sizing heat exchangers etc.

3.4.16 Arbitrary Tokenism and Relations

The problem of arbitrary tokenism and the use of relations is analogous to the situation involving the names of slots of objects. In that case, meta_ and dependency information can be defined in the slot descriptor terms. User supplied inference techniques can be used to ensure this information is interpreted whenever a call to "$set" or "$check" is made.

The same technique is required for relations. Since relations are defined like objects, the meta_information can be stored in the slots term of a relation definition. This may describe the transitivity of a relation, the dependency of a symbolic relation on other relations etc. User supplied inference techniques can then be used to override or augment the default interpretation of relations.

In the case of constraints, inter_relation/slot dependencies are given by the bindings and return_form entries. There are two problem areas however:

1. The use of arbitrary tokens for slot names in expressions, eg vapour_pressure $of Component.
2. Different relations expressing the same constraint but in different algebraic forms

The first problem is inevitable. The second problem could be avoided by reducing all equations to a standard normal form, eg polynomial expressions. Any new relation definitions could then be checked against existing definitions and any redundancies either eliminated or noted. In numerical analysis algorithms it is often necessary to use a particular form of an equation for convergence to occur. These special forms could still be defined as separate relations but their equivalence to a base relation should be noted in the definition.

3.4.17 Specifying Low Level Design Procedures

The use of operator relations are primarily intended for the manipulation of algebraic constraints in an equation based environment. There are a number of well established design tasks or procedures that are more naturally expressed as a sequence of predefined steps or goals. Each step or goal, however, may well refer to the use of an operator relation to evaluate some set of constraints. The procedures may describe actual algorithms, e.g. the use of Aitken's method in the solution of a sequential

modular flowsheet. At a still higher level a procedure may describe the steps involved in detailed sizing of a heat exchanger. Process design in an industrial context involves the use of a great many such procedures.

CLAP uses special kinds of objects called "methods" to express procedural information. In their simplest form, methods merely take a number of objects as arguments and then evaluate a sequence of procedure calls relating to those arguments. The generic form of a method is

```
method (<method name>, <argument list>
   variable - <local> - <argument slot variable lists>
   type     - <program type>
   program  - <sequence of calls>
```

The "variables" term just names any local variables and the variable lists that specify the slot values of each object argument. The type of a method can be one of program, external or prolog. The latter two types are explained later. If the type of a method is program, then the program term is the body of the method, ie the sequence of calls to be evaluated. Each step in the sequence of calls can be one of

1. A CLAP command, eg manipulating objects, using worlds, etc
2. A Prolog clause to perform some proof
3. A call to an "external" procedure or subroutine written in some other language, eg C, Fortran, etc.

The different types of statement can be freely mixed in the definition of a CLAP method. Once a method has been defined it is called in one of the following forms:

```
$call Method
$call Method - Object Arguments
$call Method - Object Arguments $in Context
$call Method - $all InstancesOfClassType $in Context
```

To make the body of a method resemble languages such as C there are two useful constructs:

1. $if Test $then Action1 $else Action2, to express the usual if-then-else statements. There is also a $else_if form.
2. for(Variable = Range), ..., end for,
   to express iterative loops.

As a simple illustration, the following code indicates how a method to solve a flowsheet might be written. The method takes a "flowsheet" object as an argument. The slots of this object would hold the necessary model details.

```
method(solve_flowsheet, flowsheet-F):-
variable - ...
type     - program
program  - ($create_context World1,
            $if not($check slot - F - properly_specified - true)
            $then verify_specification (F),
            $call converge_flowsheet - F
            ).
```

The method simply creates a world, checks that the flowsheet is not over or under specified then calls another method to converge the flowsheet. All the steps happen to be CLAP commands except "verify_specification" which might be a Prolog clause used to prove that flowsheet F is correctly specified.

At first sight a method definition looks like a routine in a standard language, albeit in a high level language that has access to objects and Prolog clauses. Even at this stage, however, the expressive capabilities of a method should not be under-estimated. There is full access to all the CLAP commands. This includes all the features associated with objects, eg demons, default values, meta_view points etc, as well as full access to the Prolog interpreter to perform proofs nested within a method.

There are three further features of methods that greatly enhance their usefulness:

1. An interface to external routines written in other languages.
2. A built-in mechanism to call different inference techniques when choosing process models or making decisions.
3. The technique used to interpret the code in the body of a method

These features shall now be discussed briefly in turn. An extended example of how they could be used in a method for solution of sequential modular flowsheets shall then be given to clarify their use.

3.4.18 External Language Interface

Symbolic languages such as Prolog and Lisp are not that well suited to performing large amounts of numerical computation. In a practical design environment it will be necessary to have the well defined aspects of problem solution calculated as efficiently as possible. A typical example would be the use of Gaussian elimination to solve a set of linear equations represented as arrays of coefficients.

CLAP provides a two-way interface between itself and the procedural language C to allow numerical data to be created and modified at either level. This means that data structures, eg arrays, vectors, single variables etc, can be created by a CLAP command and then accessed or modified within a C function or vice versa.

There are a number of CLAP commands provided to manipulate arrays. These include:

1. $create_int_array, $create_double_array - create integer or floating point arrays as requested, eg $create_int_array(a(10)), a vector "a" of ten elements, $create_double_array(b(10,20)), a ten by twenty array.
2. $initialise_array (Name, Value)
3. $add_arrays (A1, A2, A3), solves A1 + A2 = A3
4. $multiply_arrays (A1, A2, A3)

5. $transpose_array (Array1, Transpose)
6. $gaussian_elimination (A, B, Error) - solves A.x = b leaving the result for x in B.

The Gaussian elimination command raises an important point concerning the use of "black box" algorithms and the problems they cause, see Chapter 1. All the Gaussian elimination command does is perform the elimination process in C to speed up calculation. This process is seen as an acceptably low level operation that can be performed without further reasoning. This would not be the case, for example, with Newton's method for the solution of a set of non- linear equations. This method involves performing Gaussian elimination within an iterative loop until values for the vector of unknowns converge. There are two important problems concerning the use of Newton's method:

1. Initial guesses for the unknowns - if poor guesses are provided then the method may not converge.
2. Infeasible values or estimates are generated for the unknowns during the solution process, usually resulting in failure to converge.

A lot of domain specific information could be used to eliminate these problems. For instance, if Newton's method is part of an equation based flowsheet program then good guesses for many unknowns will be available from the current context or state of the process design. This is particularly true if a top-down, hierarchical approach to flowsheet formulation has been used, e.g. Douglas(1988)

By writing Newton's method as an external "black box" a number of unnecessary problems are introduced since relevant information or knowledge is not being used in the problem solution. The following two sub-sections indicate how domain specific information can be used to guide a general algorithm such as Newton's method.

There are several other features of the external interface worth mentioning. The first is that there is a library of C functions, corresponding to the CLAP commands for arrays, that allow data structures to be created dynamically in a C function and then accessed later within the CLAP interpreter. These data structures can then be associated with high level CLAP objects or methods.

A special object type called a "table" is provided in CLAP. A table is a normal CLAP object, with slots, demons etc, except that it always has an array associated with it. The size of the array is specified by the slots "row-size" and "col-size". Individual elements of the array can be accessed or set, or the whole table array can be manipulated by using a set of commands identical to those used for manipulating arrays, e.g.

```
$create_object table(double) - newton_info1 $in world1,
```

```
$set slot - newton_info1 - row_size - 10 $in world1,
$set slot - newton_info1 - col_size - 10 $in world1,
$set slot - newton_info1 - store_var - coefficients $in world1,
$set el(3,4) - newton_info1 - 12.5 $in world1,
...
$check el(1,2 - newton_info1 - Value $in world1
```

(The value of slot "store_var" is the name of the array in C). The programmer can provide his own set of slot definitions when declaring objects of type "table" in order to represent problem specific information regarding the contents of the table array.

Finally, there are two ways of calling user provided C functions to perform arbitrary calculations. The first, and preferred, method is to define a method that has an external type:

```
method (user_callx, ObjectArguments) :-
   variables          - ...
   type               - external
   passargs           - ...
   argtypes           - ...
```

Suppose there was a single object argument of type column, ie

method(user1, column-C)

The "passargs" term is a list of the slots that are used as arguments when the external function is called, eg

passargs - [no_of_plates $of C, reflux_ratio $of C]

The "argtypes" is a list corresponding to passargs that indicates the "type" of each argument, ie integer, real or string and whether it is an input or output argument. All input arguments must have their slot values set. When the function is called and returns, the slot values corresponding to the output arguments will be automatically updated. This may result, for example, in demons being activated. It is extremely simple, therefore, to interface external functions to CLAP. They are called in the same way normal methods are called

$call user1 - column1

The second way of declaring external functions and data to CLAP is through the use of "procspec" and dataspec predicates. A function modelling a reactor, say, with 5 arguments is declared as

procspec(external, reactor, 5,[+double, +int, +int, -double, -'char*'])

The last argument is a list of argument types. A '+' indicates an input argument, a '-' indicates an output argument. The 'char' argument indicates a string in C.

Similarly to declare an external 100 by 100 array of real numbers to CLAP, the predicate

dataspec(external, 'double *', s, [100, 100]).

is used. The second last argument is the name of the array and the last argument is a

list of the dimension sizes. In this case, both external functions and external data can be treated like prolog predicates in the definitions of other methods:

```
method (toplevel, Arguments):-
   ...
   $call methodA - object1,
   reactor(s(1,100), 5, 3, s(2, 100) ErrorMessage),
   ...
```

The call to reactor looks like a call to any other Prolog clause. A feature of CLAP, however, is that it monitors the use of external data names. The array s, for instance, is an external array. When the call to reactor is made the first argument is replaced by the current value of s(1, 100). When the call returns the value of the output argument s(2, 100) is updated.

There are two differences between external CLAP methods and external functions declared using "procspec".

1. CLAP methods, whether external or normal, can use the message passing facilities and are themselves high level objects which can be reasoned about by other methods.
2. There is no control over the use of "procspec" external functions. CLAP has no access to the definition of the function or its intended purpose unless the programmer provides a set of describing predicates or rules and a specialised interpreter.

3.4.19 Using Inference Techniques in CLAP Methods

At the start of this section it was stated that methods in CLAP were primarily intended to represent "low level" design procedures as a sequence of steps to follow. The discussion so far has shown that the definition of a method can consist of calls to other CLAP methods, or calls to external functions using externally defined data structures, or calls to Prolog to prove some goal. Simple "$if - then" constructs can be used for known choices or alternatives while the "for(I = Range)" construct allows iteration to be performed. Any step in a method can also be a CLAP command. It is a trivial matter then, to dynamically create new instances of objects, create multiple contexts or worlds, set slot values, use meta_view points, invoke slot demons etc, at any point in a method.

The final feature of CLAP methods to be discussed here is the mechanism provided to call on other inference techniques, production rule systems etc. It is seldom the case that even for well defined, "low level" design procedures, the decisions to be taken can be expressed in a simple, problem independent way. The use of if-then rules describing the choice of heat exchangers, for example, is highly dependent on the problem context, eg is safety the prime consideration, or layout restrictions etc.

This problem dependent information greatly colours or influences the way in which if-then type statements should be interpreted.

Rather than try and express this information as complex "if then" statements in a method definition, as would typically happen in a simple C or Fortran subroutine, the following CLAP commands can be used to invoke more comprehensive inference techniques:

```
$model ObjectType - InstanceName $infer_with InferenceMethod
$model ObjectType - InstanceName
$decide_on ThingToChoose - ResultChoice $infer_with InferenceMethod
$decide_on ThingToChoose - ResultChoice
```

The first version of "$model" is used to invoke an InferenceMethod to select an appropriate model for an object instance, eg

$model column - c101 $infer_with column_modeller

The inference technique "column_modeller" might be a production rule system or another module in Designer's Assistant or a Prolog search method, etc. When a model has been selected a further method can be run to evaluate the model if this is appropriate. This latter information is specified in the predicates used to define the interface to the InferenceMethod. The form of the call is very flexible in that

1. Alternative inference methods can be used depending on the problem context, eg memory and time constraints, amount of detail available on the object being modelled, etc. The choice of inference method itself can therefore be the result of some meta_level decision procedure, as was the case in the MECHO system, section 3.2.5
2. The model chosen can reflect the level of detail at which the object is described and take into account other context dependent, problem specific information. Subsequent calls to "$model" could generate, therefore, different models to use, depending on the overall status of the problem solution.

If the second version of $model is used, ie no inference technique is specified, CLAP searches for the definition of a default method to use and runs that instead. It is up to the programmer to provide a default technique.

It is also possible to specify that an object be modelled by a particular method instead of always calling an inference technique. For instance, the call

$model column -c101 $by mcabe_thiele $in world1

specifies the use of a particular model. This is useful when a direct comparison of models in different contexts is to be made.

The calls to "$decide_on" are very similar but are concerned with making any sort of decision choice rather than selecting a unit model and running it. A simple example might be

$decide_on column_packing - Type $infer_with rule_system1.

The interface to rule_system1 might indicate that it exists in a separate module of Designer's Assistant. If so, the "$decide_on" command is a very simple way of establishing communication between modules in a blackboard system.

It is worth noting that the $model inference techniques can, of course, make full use of the relations facility in order to help decide which alternatives to choose.

3.4.20 Interpreting Methods in CLAP

The ability to make calls to CLAP commands, run methods, call Prolog clauses, use inference techniques, etc., makes the definition of well specified design procedures a straightforward task in CLAP. The inherent flexibility of a CLAP method is further enhanced by the strategy used to interpret its definition.

Evaluation of a CLAP method does not just involve calling each step in a sequential manner from start to finish. The body of a method, ie the sequence of steps, is viewed instead as a conjunctive formula to be proved in much the same way as a theorem prover evaluates a conjunction,

C1 $\wedge$ C2 $\wedge$ ... $\wedge$ CN. CLAP uses a left to right, backtracking search strategy to evaluate a method. This means that each step must implicitly evaluate to "true" before its successor can be interpreted. If a step fails for any reason the CLAP interpreter backtracks to the last choice point on the last step interpreted. If there are no further choice points then the interpreter backtracks to the second previous step and re-evaluates that if choice points exist. This backtracking/re-evaluation process continues in a depth first, recursive fashion until either the method is successfully interpreted or all choice points are exhausted and the method itself fails. This in turn can cause backtracking into a parent method that has called a child method if the child method fails.

In order to eliminate choice points, ie a method that can only be run once, the Prolog "cut" symbol, which is "!", can be used as a step in a method. Backtracking cannot take place anywhere between the first step of a method and the appearance of the cut symbol. If the cut symbol is the last step in a method then no backtracking will occur. If steps in a method appear after a cut symbol then partial backtracking can occur between the cut symbol and the last step in a method.

If a step in a CLAP method is a Prolog predicate then backtracking to this predicate will invoke the backtracking facility of Prolog to re-evaluate the predicate. This is extremely useful when defining "generate and test" procedures. The basic principle is

to generate a hypothesis and then test it. If the test fails then backtrack and generate another hypothesis. The whole process is repeated until a successful hypothesis is generated. If any subsequent hypotheses fail then backtracking can cause yet more hypotheses to be generated. The general format is:

```
method (select_thing, Class_Instance) :-
    ...
    generate_choice (Class, Instance, Choice),
    test_choice (Choice, ANs),
    ABs = true,
    dependent_goal1 (Choice, ...),
    ...
    dependent_goalN (Choice, ...),
```

3.4.20.1 Demonstration of CLAP Methods, A Simple Flowsheeting Tool

As a demonstration of some of the features, described above, in very simple flowsheeting framework will be described. The system is based on ESSPROS, Ponton(1985), a simple linear mass balancing system that is converged in a sequential modular fashion. By sequential modular it is meant that each process unit is represented as a function which given an input produces an output. A flowsheet is represented as a number of such functions that are called sequentially in a specified order. The calling sequence is repeatedly evaluated until convergence of all unknows is achieved.

The following example defines two generic object types, a process "stream" and a "mixer unit".

```
object(stream) :-
    self - S,
    variables - [N,P,T,C1..CN,CF1..CFN,F],
    slots - [name-N, pressure-P, temperature-T,
           component(names) - C1..CN,
           component(flowrates)- CF1..CFN,
           total_flowrate - F
          ].

object(mixer) :-
    self - S,
    variables - [S1,S2,S3],
    slots - [ stream - S1, stream - S2, stream - S3].
```

Given these generic types, instances of streams and mixer units can be created either using "$create_object" or using the following predicates.

```
dataspec(external,'double *',s,[100,100]).
dataspec(external,'double *',p,[100]).
dataspec(external,'double *',t,[100]).

stream-s1 :-
    variables - [1,0.0,0.0,
                 ch4..c2h6..c3h8..n2..h2..so2,
                 s(1,1)..s(1,6),
                 F
                ].
```

```
stream-s21 :-
    variables - [21,0.0,0.0,
                 ch4..c2h6..c3h8..n2..h2..so2,
                 s(21,1)..s(21,6),
                 F
                ].

mixer-m101 :-
    variables - [s1,s21,s121].
```

This creates two stream instances, s1 and s21, with undefined flowrates. The mixer m101 has two inlet streams, s1 and s21, and one outlet stream s121. The dataspec entries declare three external arrays that are used within ESSPROS. Notice that the component flowrates of streams s1 and s21 use parts of the external "s" array for storage, elements s(1,1) through to s(1,6) and elements s(21,1) through to s(21,6) respectively.

The following entry shows the two ways in which external functions can be interfaced in CLAP. The first part defines an external method "mix" which takes an instance of a mixer unit object as an argument. The arguments passed to the external function are defined as the values of the "name" slots of the stream objects In1, In2 and Out. Given the previous definition of mixer m101, and if m101 was the argument then In1 = s1, In2 = s21 and Out = s121.

```
method(mix,mixer-M) :-
    variables - $null - [In1,In2,Out],
    type - external,
    argtypes - [+int,+int,+int],
    passargs - [name $of In1,name $of In2,name $of Out].


procspec(external,reactor,5,[+int,+int,+int,+int,+double]).
```

The "procspec" entry indicates that there is an external function called "reactor" which has 5 arguments. The arguments are all input arguments as ESPROSS stores stream outputs in the external arrays s, p, and t. In ESSPROSS a stream is identified by a unique integer and ESSPROSS functions take these stream identifiers as arguments. The stream integer identifier is stored as the "name" slot in the definition of the stream object above, eg the "name" of s21 is 21, the name of s1 is 1, etc.

Suppose a "flowsheet" object is defined that has the calling sequence of functions defined as one of its slots as in

```
object(flowsheet):
   self - S,
   variables        - [Calls, Solver, Tests, Initial, Updates ...]
   slots            - [calling_sequence  - Calls,
                       convergence_technique      - Solver,
                       convergence_criteria       - Tests,
                       initialisation_calls       - Initial
                       update_calls               - Updates
                       ...
                       ].
```

A number of other useful slots can be defined such as how to initialise the flowsheet.

Given a flowsheet object it is extremely simple to write a generic method to evaluate the calls. The following template for one such method takes a flowsheet object as its argument. The variable lists are expanded in this example to avoid making calls to $check to find out slot values (the use of variable lists is explained in Struthers(1987) but should be obvious from the example).

```
method (sequential_modular(Package), flowsheet - F):-
   variables          - [] - [Calls,Solver,Tests,Initial,Updates,...],
   type               - program,
   program            - (initialise_package (Package),
                         set_solution_method (Package, Solver),
                         initialise_flowsheet (Initial),

                         current_context(C),
                         $display F $in C,

                         for(1, 1, IterationLimit),
                         $display all_cvars,
                         Calls,
                         $if Tests $then $exit
                             $else Updates
                         endfor,

                         save_output (Package)
                         ).
```

Before explaining the method it is worth listing the slot values of an instance of a flowsheet object, demo_flowsheet. The following structure is the value of the "initialisation_calls" slot.

```
[ reset,
  startiteration,
  setmethod('AITKEN5'),
  setnc(6),
  setstream(1,0.0,0.0,'100,0,0,0,0,0'),
  setstream(15,0.0,0.0,'0,0,0,0,0,0'),
  setstream(16,0.0,0.0,'0,0,0,80,0,0'),
  setstream(21,0.0,0.0,'0,0,0,0,0,0'),
  setstream(101,0.0,0.0,'0,150,0,0,0,0'),
  setstream(102,0.0,0.0,'-1,-1,1,0,1,0'),
  setstream(103,0.0,0.0,'1,0,0,1,1,1'),
  setstream(104,0.0,0.0,'0,-1,1,0,-1,1'),
  setstream(105,0.0,0.0,'1,0,0,0,1,1'),
  setstream(106,0.0,0.0,'1,0,0,0,0.999,0.001'),
  setstream(107,0.0,0.0,'0,0.8,0.003,0,0,0'),
  setstream(108,0.0,0.0,'0.001,0,0,0,0.999,0.01'),
  setstream(111,0.0,0.0,'0,0,0,0,0,0')
]
```

The slot value is a sequence of calls used to initialise a number of streams in ESSPROS. The setstream function is an external function declared by a "procspec". The value of the "calling_sequence" slot is the structure.

```
[ for(1,1,30)
  $display all_cvars,
  controlledmixer(15,101,7,6,0,150.0),
  split(7,2,8,0.7),
  $model mixer-m101 $by mix,    /* ... ->  mix(1,21,121), */
```

```
    mix(121,2,3),
    reactor(3,4,102,1,0.7),
    separator1(4,5,13,103),
    mix(5,8,9),
    mix(9,111,110),
    reactor(110,112,104,5,0.2),
    split(112,111,10,0.0),
    separator1(10,11,12,105),
    separator1(11,18,19,106),
    separator1(18,20,21,108),
    mix(12,13,14),
    separator(14,16,15,17,107),
    teststream(15),
    endfor
  ]
```

This shall be explained shortly. The slot value for the "convergence_criteria' is

```
unconverged(N) $and N=<0,
```

while the value of the "update_calls" slot is

```
$decide_on solution_technique - Choice,
set method (Choice)
```

The method is called by

$call sequential_modular('ESSPROS') - demo_flowsheet

The first two steps in the method, initialise_package and set_solution_method, simply initialise the data structures of any particular package. They are written here as Prolog clauses, allowing multiple definitions for each flowsheet package that can be used with this generic method. In the case of ESSPROS this initialisation is very simple:

```
initialise_package ('ESSPROS'):-
   reset,.              /* an external function */
   startiteration       /* an external function*/

set_solution_method ('ESSPROS', Method):-
   setmethod (Method).
```

The solution methods used in ESSPROS are AITKEN3, AITKEN5, and NEWTON. AITKEN3 uses Aitken's acceleration method on every third iteration, AITKEN5 is the same but every fifth iteration, and NEWTON is Newton's method.

The first important call is initialise_flowsheet. This takes the sequence of stream initialisation calls, stored as the value of the slot "initialisation_calls", and evaluates each of these calls. This is written as a simple recursive Prolog clause using the cut symbol, "!", to eliminate any backtracking:

```
initialise_flowsheet ((Call, OtherCalls)):-
   interpretcall (Call, true),
   initialise_flowsheet (OtherCalls).
initialise_flowsheet (LastCall):-
   interpretcall (LastCall). !.
```

The clause "interpretcall" allows any CLAP command to be interpreted in Prolog.

Once the flowsheet has been initialised it is displayed in the current world or context. Figure 3.11 below shows the display associated with this flowsheet object. (The "$display ObjectName" is a CLAP command to display any object). The main part of the method is a simple iteration loop. The first call

$display all_cvars

is a CLAP command to display any external variables that are to be traced during execution. In this case the component flowrates of recycle stream 15 are displayed on the screen, see Figure 3.11. The command simply looks for predicates such as

display_var(s(15,1),Current Value,[DisplayInstructions]).

This would display the array element s(15,1) somewhere on the screen.

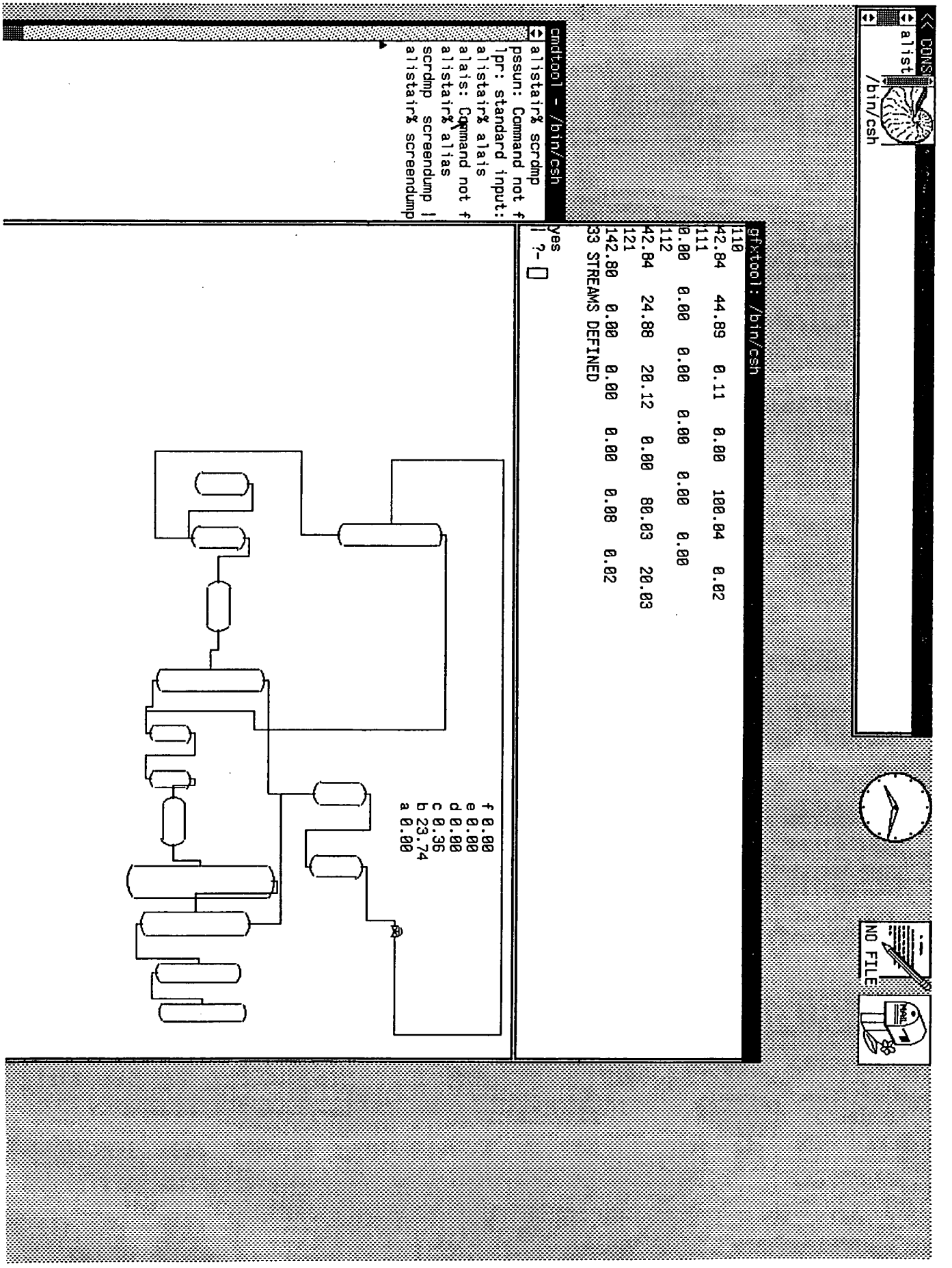
The second call of the "for loop" takes the sequence of calls stored in the "calling_sequence" slot and interprets them with the CLAP interpreter. There is no need for any "$call" commands since the value of the slot is a sequence of CLAP calls. The definition of the generic method "sequential_modular" is specialised to a specific form every time it is run by simply unifying the variable Calls in the second line of the loop with the same variable in "variables" list describing the argument object's slot values. In this example, the first ESSPROS function called is controlled mixer, then split, and so on.

The third step in Calls uses the "$model" command to select a particular type of mixer model. If the "$infer_with" variation had been used then this model could change on subsequent iteration loops. In the case of a reactor model for instance, it might be useful to use a rough model in the first few iterations and then use a more sophisticated model as the flowsheet nears convergence.

The backtracking strategy of CLAP also provides a simple means of recovery from infeasible estimates. If a subsequent call fails, perhaps due to use of too crude a model elsewhere, then the system can backtrack and re-evaluate another model using the "failure" information as part of the decision process.

Once the calling_sequence has been evaluated, the convergence_criteria are interpreted in the same way. In ESSPROS this simply means counting the number of unconverged streams. If no unconverged streams are found the flowsheet has converged. If the convergence_criteria evaluate to "true" then the system exits the iteration loop and saves any ESSPROS output. If the flowsheet has not converged then the "update_calls" are interpreted. In this example there is a call to "$decide_on" the solution technique. This is changed from AITKEN5 to AITKEN3 whenever the number of unconverged streams drops to less than four. In a more complex package,

Figure 3.11: Graphic Display of Flowsheet Recycle Variables



problem specific information could be used to guide sensitive numerical solution techniques, take care of memory management, etc.
The output of ESSPROSS is simply a listing of the "s" array. The values of this array are accessible, of course, through the "component(flowrates)" slot of the stream objects.

Despite the simplicity of the ESSPROS demonstration, it illustrates a number of important points.

1. It is straightforward to write well defined, generic solution procedures using methods in CLAP. Methods combine aspects of object orientated, procedural, and logic programming in a flexible way that is easy to use. Full use of CLAP relations and multiple inference techniques is also possible within methods. This variety of representation style is necessary if large, complex problems are to be easily represented in a single system or module of Designer's Assistant.
2. The use of appropriate high level objects, eg the flowsheet object, means that the "control and implicit reasoning" information used in black box programs can also be represented and reasoned about at a higher level. It is much simpler, therefore, to take account of problem specific information to guide the control of low level algorithms, especially optimisation and numerical analysis type algorithms, yet still retain the ability to write generic methods or frameworks to describe an algorithm.
3. It is straightforward to write generic algorithms that can be specialised to a specific form depending on the problem content. The code used to specialise the generic algorithm can be represented as data structures in slots of an appropriate high level object. In this demonstration, the initialisation calls, convergence tests and main calling sequence for a sequential modular mass balance method were all represented as slots in a flowsheet object. This means the formulation of the code, ie deciding which functions to call and in which order, can itself be the subject of an explicit high level reasoning strategy. This has important implications in an industrial context. The correct formulation of complex flowsheets for a particular package is by no means a trivial problem.
4. It is easy to decouple various aspects of a method or algorithm using CLAP methods, eg the display of graphics, initialisation requirement, control of convergence problems, etc.

The idea of treating code as data is an inherent part of languages such as Prolog or Lisp in which there is no distinction between programs or data. All programs, functions etc, are treated as data structures. This idea can be put to extremely good use as seen in the very concise but completely generic definition of the sequential modular solution method.

It is worth emphasising this idea by a typical problem that arises with Newton's method in flowsheet problems. Consider a design problem in which one unknown is the flowrate through a centrifugal pump. The equation used to model the pump is the "characteristic curve", relating output pressure to flowrate. The curve is of a "quadratic" shape but with physical limits on the upper and lower bounds of the flowrate, corresponding to zero pressure increase and the pump's shut-off head

respectively. Now a wrong initial guess for the flowrate will result in derivative information for this curve being used to estimate a new value for the flowrate. Depending on the shape of the curve this new estimate may be outside the physical upper and lower bounds. If so some interval bounding technique is needed to re-generate another estimate for the flowrate that takes into account any problem specific constraints so that Newton's method can proceed with sensible estimates for any unknowns. It would be extremely easy to make use of this problem specific information in a CLAP method using Newton's technique. Suppose each unknown, Xi, is represented as an object and that each Xi has a slot "newton_update_correction" that holds calls to modify an estimate of Xi to one within an allowable region or regions. Part of the loop used in the Newton method to update each unknown might be

```
for (x = X1 .. XN),
   generate_estimate (X, Xnew),
   $check slot-X-newton_update_correction
            - (Checkcode-Xnew-Xmodified)
   Checkcode,
   ...
endfor
```

When Checkcode is evaluated a new value for Xnew, called Xmodified is returned. Alternatively, Checkcode could change some slot values in ObjectX, fail, then let CLAP backtrack to generate_estimate to regenerate a new estimate for X taking into account the changed slot values. (The use of Xnew and Xmodified in the $check call is simply to establish variable bindings between these variables and the Checkcode).

The content of Checkcode can be changed within a single calculation as well. On the subsequent iteration the new code will be evaluated without any need for recompilation or linking of programs etc. This is true whether Checkcodes consist of calls to external functions or CLAP commands. This is made possible by representing code as data structures within high level objects. The result is that code can be reasoned about and modified dynamically to take into account the problem context. This allows much more rigorous, problem specific versions of generic techniques, eg Newton's method, to be used with no extra computing effort.

3.4.21 Message Passing in CLAP

A principal feature of object orientated programming that has not been mentioned so far is the concept of message passing between objects. The notion of an object in many O.O. systems is an entity that combines both the properties of procedure and data since they can perform computations as well as save local data states. Some O.O. systems insist on a uniform use of objects to perform all actions and change of state. In such systems all actions result from a process of message sending between

objects.

Message sending is a form of indirect procedure call. In order to achieve some intended action a message is sent to an object. Upon receipt of a message an object will respond by invoking an appropriate "method" or "behaviour". The method selected will depend on the form and content of the message. This is in direct contrast to naming or calling a procedure to perform some operation on an object.
The behaviour methods of an object are defined locally, each object thought of as an autonomous agent with a number of predefined behaviours. A set of related messages that can be handled by an object is defined by the generic class or type of the object, and is called a "message protocol". A given object class may support several message protocols. Several different classes may also support shared protocols but use different behaviours or methods for their implementation, ie the same type of message can be sent between different kinds of object which respond in different ways.

CLAP supports an extended for of message passing between objects that can be freely used within the other representation styles. Several features of message passing in CLAP are not supported in other O.O. systems and are worth discussing briefly.

3.4.22 Standard Message Passing and Method Inheritance

CLAP supports the common forms of message passing found in systems such as FLAVORS(Weinreb and Moon 1981) and LOOPS(Bobrow and Stefik 1981).

To send a message to an object simply use a command of the form

$tell Object - MessageName

A message can also take arguments and be sent to a particular world or context:

$tell Object - MessageName - Arglist $in Context

When a message is received an appropriate method or behaviour is invoked. These behaviours are defined for each object class. An instance of an object class inherits the set of behaviours from its defining class. Consider a message "draw_yourself" which is to be defined for the object type vessel. This is simply defined as:

```
object_method (column, InstanceName, draw_yourself, Arglist):-
   code - <sequence of CLAP calls>
```

The defining code can be any CLAP command or call, Prolog clause, call to an inference technique etc. This code could be invoked by the call:

$tell column101 - draw_yourself - [screenA] $in worldA Other behaviours for different messages are defined in the same way.

CLAP also allows instances of a class to inherit behaviours, not only from its parent class but from all of its ancestors. Behaviours can be successively specialised at lower levels in a class hierarchy.

Consider the draw_yourself behaviour for a column. Suppose a column is a subclass of plant_item and that the draw_yourself behaviour is also defined for the plant_item class. The behaviour at the column level can either inherit this code and then specialise it or it can overwrite the behaviour locally. In the latter case the draw_yourself definition would be exactly as before.

In order to inherit the plant_item code and specialise it the definition must be modified. CLAP uses the concept of "wrapper code" to implement "method inheritance". As its name suggests, wrapper code is additional code that wraps around, ie specialises, the code inherited from a parent class. If there are multiple levels in a hierarchy then the behaviour at any level consists of the top most definition successfully wrapped up in each layer of wrapper code down to the level at which an object is defined. There is no need to specialise a behaviour at every level. One common behaviour can be defined for all sub-classes in a hierarchy if desired.
The following example shows the general form of behaviour definition for the plant_item and column classes:

```
object_method (plant_item, Name, draw_yourself, Arglist) :-
   code - <Sequence of core calls>.

object_method (column, Name, draw_yourself, Arglist):-
   inner_args        - $all,
   before            - <Sequence of before calls>
   code              - $inherit
   after             - <Sequence of after calls>.
```

The inner_args term indicates which arguments should be passed to the higher level inherited behaviour. The "before" term is the before code and the "after" term is the after code. Both terms are optional and can be a sequence of any CLAP calls. The "$inherit" entry in "code" indicates that the code is to be inherited from the parent class.

3.4.22.1 Other Forms of Message Passing in CLAP

CLAP allows arbitrary messages to be sent of objects other than those defined in a message protocol. The command is of the form :

$tell Object - AnyOldMessage $in Context.

Where AnyOldMessge can be any Prolog predicate, expression, data structure etc. These messages can be checked later using the form

$check told - Object - Message $in Context

. It is usual that Message will be a partially instantiated term that can be pattern matched against known messages for a given Object.

Messages of this form are stored internally along with an object's slot values. The default behaviour is to do nothing when a message is received and use pattern matching to retrieve or delete messages at a later point. The programmer can provide his/her own clauses to interpret these types of message if necessary. This would be necessary if the problem of arbitrary tokenism is to be avoided when passing messages between different types of object.

Another useful form of message passing in CLAP is the notion of sending messages to CLAP methods. CLAP methods were described in section 3.4.20 and are not the same as the methods or behaviours used to handle messages. Sending messages to a method is useful for a number of reasons:

1. System control information, eg available memory, allowable run time, etc, can be sent to a method prior to evaluation.
2. If a "library" method can take any one of a number of unrelated object types as an argument then it will not be possible to specify a specific object type in the method definition. The method can receive the name of the object to operate on through a message rather than through an argument list. The same applies to any input/output data that is not associated with an object.

In order to send a message to a method the "$attach" command is used instead of "$tell". The command is of the form:

$attach <what> - Value $to Methodname,

eg $attach cpulimit - 10.0 $to solver.

There is a special MethodName called "self". This is used for a method to attach messages to itself during execution. This is useful for output or debugging purposes.

The "$attached" command is used to query what has been attached to a method. This means one method can query what has been attached to another method, or a method can query itself, eg

```
$if $attached input_data-D $to solver $then $exit.
$if not ($attached input_data-D $to self) $then $stop.
```

Since a CLAP method is in fact an object then instances of methods are created every time a method is run. Messages attached to a method are therefore stored locally in each method and can be recovered at some later point after evaluation of the method.

3.4.22.2 When to Use Message Passing and When to Use Procedures

It may seem at first sight that it is not necessary to have both procedures and message passing in the same system. Indeed many O.O. systems insist that all actions be

performed by message passing.

It is the author's opinion that both techniques are equally useful and necessary when tackling complex problems. A CLAP method is extremely useful in that it is an explicit type of object that expresses the control strategy of problem solution that would otherwise be implicit in the implementation of message protocols. In many process design problems it is very important to be able to closely monitor and reason about the solution control strategy. This is done much more satisfactorily by the use of appropriate high level objects rather than relying on "run-time" interactions in a message passing system. In addition, the information implicit in the formulation of a procedure is often important "knowledge" that should be explicitly represented in a declarative form. For example, the order of steps used in the sizing of a heat exchanger, and their relative importance, is an important part of expertise in heat exchanger design. This strategy should be explicitly available to be reasoned about and improved.
On the other hand, it is undoubtedly the case that method inheritance is an extremely concise way of defining certain types of operation local to an object. A good example is the definition of graphics code used to display an object on a workstation screen. An object can be redisplayed at any instant by simply sending an appropriate message. Another good example is sending a message to an object to make it re-initialise its slot values.

Another obvious application is in the area of discrete event simulation. Any changes of state in an object can be transmitted to its nearest neighbours by sending an appropriate message. These neighbouring objects can then propagate the effect by sending messages on to their nearest neighbours, and so on.

Even in simulations, however, knowledge of the global control strategy is better expressed in a method framework, eg how to account for objects that are modelled by sets of "stiff" equations compared to other objects which have much larger time constraints. This is important numerical information that must be properly used if accurate simulations are to be performed.

Since message sending is implemented as CLAP commands it is possible to use this technique within the definition of a CLAP method like any other feature, eg use of inference techniques, relations etc.

3.4.23 Display Options in CLAP

An important feature of CLAP is the ability to directly display objects and monitor any changes in the state of an object, ie any changes in slot values or use of relationships. Any such changes can then be displayed in some user defined way as described below.

Object displays can either be simple ASCII-terminal like displays, ie text and numerical information, or graphics based displays, or a mixture of both. CLAP has a complete interface to the core graphics interface package (CGI) for use on Sun workstations. This means that object displays can make use of multiple windows, mouse driven input, menus etc.

An explicit request to display an object or one of its slots or relations can be made by calls of the form:

```
$display ObjectName {$in World}
$display SlotOrRelation $of ObjectName {$in World}
```

This requires that an entry of the form

display - SequenceOfCalls

be present in an object, slot or relation definition. The calls can involve any CLAP command but will typically just be a sequence of graphics calls or text instructions.

In order to monitor an object's slots or relations CLAP provides a display switch mechanism. Display switches can be set for individual slots or relations, or collections of them or for all slots and relations. The following commands allow display switches to be set on or off:

```
$set display_switch(slot-ObjectName-SlotName-on) {$in World}
$set display_switch(slot-ObjectName-SlotName-off) {$in World}

$set display_switch(relation-ObjectName-SlotName-OnOrOff)

$set display_switch(all - ObjectName - OnOrOff)
```

Once a display switch has been set no further action is required. CLAP automatically redisplays any slot or relation that is modified if its display switch is set to on. The last version of "$set display_switch" above switches on all slots and relations for which there are display definitions.

Switching a display switch between on and off is extremely useful when observing a certain part of an object during a calculation. A graphics workstation display is easily customised to focus the user's attention on the slots and relations of objects relevant to the problem being solved.

It is also possible to monitor the values of externally declared variable, such as arrays etc. in C. All that is required is a predicate of the form

display_var (Variable, Value, FormatInstructions).

Variable is the name of the variable to be displayed, Value is an undefined variable that gets assigned to Variables current value, and FormatInstructions is a list of CLAP calls to display the value. For instance, in the ESSPROS example earlier a monitor was placed on the component flowrates of a recycle stream. These values were stored in an external array "s" in s(15,1) to s(15,6). An example predicate is

display_var(s(15,1),Value,[$text([X,Y],a),$ftextf([X + 15,Y],Value,'%4.2f')]).

The display output in Figure 3.11 shows that the component flowrate, written as "a" followed by its Value, is correctly displayed at point (X,Y). This technique was used to monitor the convergence of the component flowrates in the recycle stream.

Such a facility would be useful in any CLAP method that involved interfaces to external software.

3.4.24 Specifying Design Methodologies in High Level Frameworks

Methods in CLAP are intended to be used for specifying well defined design procedures or solution algorithms. Each method is seen as a predefined statement of the individual steps to be followed for solution of "standard problems". The methods, of course, can still be very flexible in that calls to inference techniques, use of CLAP relations, message passing etc, can all be used within a single method. Higher level methods can be built up in a structured way by making calls to lower level methods in much the same way as any procedural language.

The statement or expression of an overall design methodology, however, requires a less rigid framework. The whole concept of design based on an AIP approach, see chapter 1, requires that the design strategy be constantly modified to take account of the responses of different engineers at each stage in the design. A mechanism must be provided to enable a number of parallel activities to be active at any one time and then have the relevant parties agree to some common commitment in order for the design to proceed in an incremental, stage by stage fashion. A variety of design options should be maintained until enough information is available to make reasoned decisions about which options should be pursued further.

The final representation style available in CLAP that is discussed here is precisely intended to allow high level design strategies to be expressed in a flexible way. The

representation style, called "extended methods", was written with AIP design approaches in mind and has a "parallel" call facility to enable multiple processes or modules to be run.

An extended method is a generic object that is predefined in CLAP. High level design strategies are expressed as instances of this generic definition.

The generic definition is simply:

```
object(extended_method) :-
    self -Name,
    variables - [Step1..StepN,G1..GN,Glist,A1..An,R1..Rn,Object,
                 Predicate,Status,Surface,Rubbish,D],
    slots - [call_sequence - Step1..StepN,
             guards - G1..GN,
             guard_macros - Glist,
             generate_assertions - A1..An,
             repeat_points-R1..Rn,
             status_update - Predicate,
             status - Status,
             dustbin - Rubbish],
    relations - [object_argument-Object,
                 top_level_surface - Surface],
    display - [D].
```

The individual entries are used to express the following information:

1. Calling Sequence - The calling sequence is a list of "areas" or activities, as opposed to specific methods, that should be investigated. A number of activities can be investigated simultaneously by using a "pcall" construct, pcall stands for parallel call.
2. Call Guards - A "guard" can be placed on each activity in the calling sequence. Guards are simply the requirements that must be satisfied before an activity is worth investigating. They will typically be expressed as high level relationships.
3. Guard Macros - If a guard is a very complex expression then it can be simplified by using dummy macro guards that make the original expression easier to read. The full expression parts can be stored in this entry instead. This is simply a programming aid and has no further significance.
4. Call Assertions - when an activity has been investigated it is useful to assert relations that describe the status of the design and any special assumptions that were made. Each entry allows general relations of this sort to be associated with each activity.
5. Loopback Points - If any activity is "stuck for information" that should have been determined earlier in the design, the engineer can be directed back to the appropriate point earlier in the design. The Loopback points entry simply indicates which extended method should be reactivated if an activity "fails" for any reason.
6. Object Argument - The object of interest is a high level object that represents the focus of attention of an extended method. It will typically be the object representing the process being designed or an object representing the design process itself. By "design process" it is meant the design activities, their state of completion, the engineers involved etc.
7. Other Slots - any other slots needed to store information locally within an extended method. Extended methods can set their own slots just as low level methods can attach data to themselves.

The following entry shows a simplified example of an instance of an extended method called collect_input_information. This activity, ie method, is called right at the start of any design to establish some basic information, eg the name of the process site, etc. Each of the terms in the dotted sequences correspond to each other, so the first term in the calling sequence corresponds to the first term in the guards entry, etc.

```
extended_method - collect_input_information :-
    variables - [

/* CALLING SEQUENCE */
/* ================ *
                      pcall(input_information,
                            technology_details..
                            synthetic_path_spec..
                            input_site_details)..
                  $null,

/* CALL GUARDS */
/* =========== */
                 (known_object(technology-T)..
                 ($guard - know_about_reaction)..
                 ((known_object(site-_))
                      $or (create_default_site(Tech))))..
                  $null,

/* GUARD MACROS */
/* ============ */
     [know_about_reaction - (known_object(process_chemistry-Path)
                       $or (create_default_reaction(Tech)))],

/* CALL ASSERTIONS */
/* =============== */
               ($null..$null..$null)..$null,

/* LOOPBACK POINTS */
/* =============== */
               (top_call..top_call..top_call)..top_call,

/* OBJECT OF INTEREST */
/* ================== */
                      Tech,

/* OTHER METHOD SLOTS */
/* ================== */
            $null,Status,Surface,Rubbish,Display].
```

In this example the calling sequence only consists of one call. This call is a "pcall" which means the activities in brackets can be investigated simultaneously. The current version of CLAP will automatically create a number of windows on a workstation display to allow interaction with each activity. This behaviour could be specialised, for example, to cause another module on another workstation to be activated instead.

The guards for this example are simple and merely check that the basic process and plant site details are known. If the process site is undefined then a new "site" object is created. Both green field and brown field site designs are therefore catered for.

There are no assertions associated with the method and all the Loopback points refer back to the very top level of the design strategy.

The focus of interest is an instance of a "technology" object. A "technology" is an industrial term for a class of similar processes, eg ammonia technology. An instance of a technology is therefore a specific plant, and in this example the plant being designed.

The other method slots are not used in this extended method.

There are three important aspects of the calling sequence of an extended method that should be noted:

1. Each entry is either another extended method or a pcall to allow a number of activities to be run simultaneously. Parallel activities must all be resolved to a certain extent before the next activity can be investigated. This is the basic mechanism by which AIP design strategies can be expressed and executed.
2. CLAP relations are a powerful means of expressing complex constraints in a process design. Extended methods can make full use of them by checking them in guards or asserting them in the "assertions". The specialise class of relations can be used to automatically generate constraints specific to the plant being designed. The relations can still be used when only incomplete information or partial data can be provided since their use is based on the notion of "delayed evaluation". This is extremely important in "real-life" design problems where adequate information is seldom available. Indeed, an essential part of the design strategy should be to identify the "data" that is needed for an adequate design. This is indicated in the systems engineering framework in Figure 1.18.
3. The most important point about the calling sequence is that, despite its name, it is not to be thought as a strict procedure or sequence of steps to be followed. Rather, what they represent are "nodes" in the general framework, in a systems engineering context, of the design methodology that is being expressed.

The last point requires further clarification. The discussion of a systems engineering approach to process design introduced two basic ideas:

1. the system concept, i.e. the entity being modelled as a system,
2. the systems engineering procedural model. The procedural model consists of a problem solving strategy that can be successively reapplied to every "life stage" of the system. Figure 1.17 depicts this in graphical form.

This is exactly what can be expressed by an extended method. The system concept is represented by the object of interest or any subpart of it when analysis is recursively made at increasing levels of detail.

The "nodes" in the calling sequence, ie the activities or areas of interest, represent a particular problem solving methodology. The CLAP interpreter can move up and down between these nodes in a similar way to the vertical movement indicated in Figures 1.17 and 1.18. The "path" followed is not predetermined but depends on a particular problem context. The combined effect of the guards, assertions and

loopback points will determine any particular path that is taken.

It is worth considering the more detailed procedural model in Figure 1.18 and how it relates to extended methods in CLAP.

The procedural principle involves task definition, selection of search techniques, selection of solutions,etc, applied over each life stage from the Abstract to the Concrete. It was indicated that three levels of abstraction could be used in this process:

1. The Functional Level - to characterise the functional structure of the system. This must be reasoned about in the task formulation phase.
2. The Physical Level - translation of the functional level into physical and chemical effects, typically represented in the models of potential unit operations.
3. The Construction Level - development of the details needed to design actual pieces of equipment

In order to perform analyses at these levels an information storage capacity is required which represents, amongst other things, methods, data and alternative solutions. At each stage of analysis information input can be identified, eg required information, scope of the problem etc, and information output generated, eg data requirements, intermediate results, etc. Most aspects of this general model can be directly represented by extended methods that make use of the other representation styles in CLAP.

The procedural principle is embodied in the extended method itself. Each extended method represents a framework of activities, any of which can be activated simultaneously. This framework is maintained implicitly by CLAP and allows constrained movement between each activity. In complex situations, such as process design, it is preferable to explicitly represent the design process itself, ie the status of activity completion, etc, as a high level object. It is then easy to relate the overall design activity to the identifiable life stages in terms of suitable CLAP relations.

Each of the representation levels, functional, physical and constructional, can be represented by either relations or slots for the object of interest. Reasoning about the functionality of a system can be achieved by identifying the intended purpose of generic units by a suitable "descriptive relation", eg to exchange heat between streams, or to separate components. This "descriptive relation" can be applied to any unit or system that satisfies the domain and active code requirements of a CLAP relation. To check whether the functionality can be achieved, e.g. can two particular components be separated, requires the definition of another relation stating the constraints to be satisfied. Checking these constraints may involve calling low level

methods, using message passing etc.

The representation and storage of methods, data, alternative problem solutions etc, can all be directly expressed in CLAP using the various representation mechanisms, eg objects, slots, meta_view points, contexts, methods, specialise and operator relations, etc. These are not directly referred to in the activity framework of an extended method but directly or indirectly in the guards or assertions of the extended method. Checking or asserting either relations or slot values can activate code to run methods or check other relations. The specialise class of relations and use of Prolog instantiation in CLAP methods means that problem specific constraint forms and "procedural type" code can be automatically generated.

The information input and output requirements at any stage can be directly expressed in the guards and assertions associated with any activity in an extended method. The problem specific output requirements generated by one activity can be picked up and used as input requirements to be satisfied by subsequent activities.

The concept of an AIP approach is not explicitly represented on Fig 1.18 but see Fig 1.21. The "pcall" construct in an extended method provides the basic mechanism to implement such a strategy. In a blackboard system such as Designer's Assistant each parallel activity could be handled by independent modules covering different aspects of the design.

3.4.25 Summary: Extended Methods

Extended methods in CLAP, when combined with the other representation mechanisms, provide the basic functionality required to express complex, high level design methodologies. A methodology expressed as an extended method is not a strict set of procedural steps to follow but a framework of activities that can be applied to any particular system of interest. The linear ordering of activities simply represents the implicit hierarchy of constraint requirements to be satisfied by each activity.

The interpretation need not be "linear", i.e. sequential, at all. CLAP provides the ability to move up and down between activities depending on unforeseen problems that may arise. This movement is guided by loopback points which indicate a likely activity, or group of activities in the case of reference to a "pcall" entry, to be resurrected.

The basic purpose of extended methods is to allow -

1. a design methodology to be expressed in a suitable framework
2. the status of a design to be explicitly maintained along with any assumptions,

design decisions taken and at what point, etc

The sequence of steps taken in each activity is determined in a dynamic, problem specific manner. For example. a top level activity might be to establish a block flow diagram for the process. This in turn may require an initial mass and energy balance as an internal requirement. The generation and solution of the mass and energy balances can vary in every problem both in terms of the model detail and the solution techniques. The most suitable techniques and models can be selected for a particular problem by using the "$decide_on" and "$model" commands. Problem specific code can then be evaluated by generic procedures, as shown in the ESSPROS example in a straightforward way. This is exactly what was alluded to in section 1.8 where it was stated that a systems engineering approach should provide a flexible framework that can be used in an appropriate and creative manner.

Since an extended method is represented as an object and the calling sequence of activities is represented as a slot value naming other extended methods, there is nothing to prevent inference techniques being written to reason about the activities that should be involved. It is questionable, however, whether reasoning about task formulation at such a high level of abstraction would be of much use in process design. In industrial designs, the identifiable life stages or intermediate design goals are very clearly defined and the number of distinct activities that must be completed to achieve these goals are usually the same, eg, development of process flowsheets, design specification sheets, piping and instrumentation diagrams, technical drawings, etc.

Reasoning about steps within a single design procedure, as described by a low level method for example, does not require the use of an extended method.

3.4.26 Discussion and Summary of the Overall Structure of CLAP

. As indicated in the preface to this section the development of CLAP either predated or coincided with the widespread availability of comparable commercial software on non-specialist hardware, eg KEE(1985), Knowledge Craft(1988) and ART(1986). These systems are usually called hybrid systems in that they provide a number of programming techniques, many of which they share in common. For example, they all support description of objects in some form and make use of multiple worlds or contexts.

KEE is basically an object orientated programming tool written in Lisp. KEE does not use explicit meta-slots but equivalent information can be expressed in so called "slot facets" which further describe a slot (they can best be thought of as slots of a

slot). KEE supports a "strict" form of message passing which relies on the use of slot demons. Other than objects, KEE has a production rule interpreter, a truth maintenance system and a limited pattern matching facility.

The rule interpreter is a simple agenda based, rule activation system. The programmer can modify the control strategy for processing the agenda by settling various global variables. The truth maintenance system can be used in conjunction with the rule interpreter. Dependencies are expressed by so called "deduction rules" of the form

```
while   <conditions>
believe <statements>
```

These rules are maintained by the rule interpreter and are commonly used to "poison" or rule out possible worlds representing different states of objects.

KEE has a TELL_AND_ASK language which allows simple forms of predicates or logical expressions to be asserted or queried.

KEE also has extensive graphics facilities and in particular uses "active images" to activate demon like functions associated with an object. This allows extremely interactive and visual user interfaces to be developed.

Knowledge Craft is very much a set of library utilities, written in Lisp, that can be used to build a specialised application. The most notable feature of Knowledge Craft is its use of relations between objects. In the context of CLAP the relations are of the simple symbolic type. Knowledge Craft also has an extended version of OPS5 that takes account of Knowledge Craft's objects and allows sets of production rules to be constructed in the usual way.

ART is interesting in that everything is written as rules. Great care has been taken, however, in providing an interface that does not require the programmer to write object descriptions, say, in an unnatural rule based format.

In terms of representation style there is little to choose between any of these systems but KEE is generally regarded to be the most comprehensive system.

Despite a number of resemblances to these systems at a superficial level, CLAP has several distinct features that makes it a useful development tool.

1. Perhaps the most important point is that CLAP was purpose built to allow procedural and declarative knowledge to be represented in the same system in a natural way, hence the name CLAP - Combined Logic and Procedures. Low level procedures, ie CLAP methods, are evaluated or interpreted as proofs within a backtracking control strategy. The ability to treat "programs" or code

as data structures when combined with Prolog's unification mechanism, allows extremely clear and precise generic methods to be defined.
2. External software can be easily interfaced or incorporated into methods. This includes using or referring to external data structures as values in objects.
3. Relations in CLAP provide a way to automatically generate sets of equations that can be used in building process models. Operator relations can be used to create distinct solutions of specialised constraint forms under different sets of bindings.
4. Meta_view points provide a powerful way of viewing slots or relations from a particular point of view.
5. The use of multiple inference techniques is supported by explicit CLAP commands, eg $model, $infer_with, rather than relying on the programmer to provide LISP code to modify the behaviour of the system.
6. It is easy to replace or augment CLAP's default interpreter techniques when more sophisticated methods are required, eg the use of a meta-vocabulary for consistent interpretation of slots. This is done by setting a number of allowed user-options.

In comparison to other systems CLAP has a rich variety of representation styles that are completely compatible with one another. The emphasis in CLAP is to use the most appropriate representation technique for different aspects of the same problem. Certain information is best expressed as rules, while other knowledge is better represented as a class hierarchy or as an explicit procedure or as sets of relations and constraints. Local consistency checks are naturally expressed as demon functions while global information can be conveniently transmitted by message passing facilities.

It seems unnatural, and totally unnecessary, to try and force information to be represented in a particular style. This is especially true in a complex problem domain such as process engineering design in which a variety of problem types must be catered for.

All of these considerations are, of course, completely general and can be made regarding any knowledge representation system. The fundamental difference between CLAP and other commercial systems is that it provides three specific techniques that are essential in the development and use of a process engineering design environment:

1. A hierarchy of explicity defined relationships or constraints that can be automatically specialised in a particular problem context. This can be used to generate sets of equations at a low, model building level or describe inter-object constraints at a much higher level in common engineering terminology.
2. An ability to express well defined design procedures or solution algorithms in a generic way that can be specialised later to take account of specific details of a problem. Low level CLAP methods provide such a facility.
3. An ability to express a design methodology in a high level framework that allows adequate flexibility in the way it is applied to any one particular problem. Extended methods can be used for this purpose. In addition, they provide a basic mechanism to allow a number of parallel activities to be coordinated within the framework. This allows for the development of AIP

based design strategies.

A short summary of the overall relevance of different representation techniques to aspects of process design is given directly below.

3.5 Relevance of Representation of Styles to Process Design

This section provides a brief summary of the different types of representation techniques investigated and developed in the initial research work of this thesis. In particular their potential for immediate application in the development of a process design environment is considered.

The comments given below are made with considerable hindsight and represent a pragmatic view of the use of current representation techniques as high level programming languages or styles. The discussion is solely concerned with the practical adequacy of these formalisms for immediate software developments that will improve upon the current software situation as described in Chapter 1. The remaining chapters will argue, however, that the formalisms are in fact totally inadequate, from a theoretical point of view, to address the basic problem of integrity of knowledge in a process design environment.
The comments are still useful, if only for comparison with related research by other workers in this field, Section 3.6 .

3.5.1 Rule Based Production Systems

The literature suggests that rule based production systems have been successfully used in tackling certain types of problem in a wide variety of areas. In particular, problems that are of a relatively narrow scope and can be characterised in situation-response terms are amenable to solution.

In the context of process design, rule based systems can be applied, often in a trivial way, to standard design procedures based on decision trees. Typical examples include the selection of most items of process equipment, eg valves, heat exchangers, pumps, types of distillation column internals, etc. Since decision trees can be processed without the use of a rule based system a reasonable question to ask is why use rule based systems at all for this purpose?

A trivial answer would be simple programming convenience. It is much easier to write decision trees in a natural symbolic form rather than invent some contrived numerical coding for its implementation. The symbolic form should be easier to maintain, since the programmer's intention is easier to follow, and hence, extend in the future. If programming convenience is the only reason and a rule based system does not provide a clear, easy to understand syntax then there seems no reason for its use whatsoever.

In the context of the main theme of this thesis, the answer, of course, lies in the basic

commitment to the explicit representation and maintenance of knowledge in a process design environment. The selection of an item of equipment is not an isolated problem but made in the wider context of the process design. Any intermediate design decisions, assumptions, etc should be available to influence the equipment selection procedure. Similarly, any intermediate assumptions, hypotheses, etc used in the decision procedure should be made explicit to ensure that they are not subsequently contradicted at a later point in the design. Any terms or words referred to in a node of a decision tree, eg the vapour pressure, that are represented by a single numeric value lose their wider significance, eg the status of the value - was it guessed, estimated, calculated, the relation of the word to other words, etc.

Most importantly, it should be realised that equipment design decision trees are usually gross simplifications and must be used in conjunction with other decision aids, eg design calculations, safety considerations etc. It is important to be able to reason explicitly about the nature and importance of the heuristics expressed in a decision tree and compare this with other sources of information when making a selection choice. For instance if one technique suggests the use of a graphite block heat exchanger it is important to know if this was even considered in an alternative equipment selection decision tree. It is this knowledge of what is considered in a decision tree and how it relates to other problem information known at the time that is so important. The use of rule based systems is just a first step in this direction but nevertheless a useful one.

Another immediate application of rule based production systems or theorem provers is in heuristic graph searching algorithms. The first section of this chapter explained at some length heuristic search algorithms in terms of searching graphs with the use of heuristic evaluation functions to estimate the cost of further search. Classic optimisation strategies, eg branch and bound, dynamic programming etc, can all be expressed in these terms. It was then shown that logic theorem provers operate on exactly the same kind of graph structures, and that many production systems can in turn be re-expressed as theorem proving systems.

The problem with many previous attempts at writing heuristic synthesis of flowsheet structures, was that high level design information was being inadequately expressed in low level representations. It was impossible, therefore, to make full use of the implications of high level statements or general background knowledge to guide the search.

Higher level design information could be reasoned about in a production system/theorem prover to help guide the lower level search strategy. In particular, it is not necessary to try and bring all the information down to the level of the

search/optimisation algorithm, ie in the form of numerical/ mathematical constraints. There is little point, and considerable difficulty, in trying to express and make use of general, static information in the form of some highly contrived cost function. Even if this could be done it would be most unlikely that the information expressed could be used in other parts of the design process.

A more sensible approach is to use appropriate representations for different levels of the search/optimisation problem. At the lowest level it is perfectly acceptable to perform rapid enumeration of, say, distillation column sequences with simple cost functions. Enumeration of sequence configurations is, however, only a small part of the design of a separation section of a process. Guiding the search process and controlling the depth to which it is evaluated within this wider context is much more easily done by means of a high level evaluation process. It also means that the intermediate reasoning performed in the search can be explicitly represented and used again later. Since both production systems and heuristic search techniques can be described in very similar ways it is easy to embed the control of the latter within the operation of the former.

There is little point, however, in taking a standard optimisation algorithm and merely re-expressing it in a rule-based form in the hope that a "symbolic" search will somehow be more effective than an equivalent "numerical" optimisation. Niida et al(1985) is a typical example involving separation synthesis heuristics. In the context of a process design environment a symbolic "black box" algorithm is just as troublesome as a numerical "black box" algorithm.

One of the main reasons for using a high level representation system at all is to allow more adequate formulations of problems to be made by placing them in a wider problem description and making better use of different sources of information in their solution.

3.5.2 Frame Based Systems and Object Orientated Programming

It is difficult to make any generalisations about frame based systems, and object orientated programming styles, as there are so many variations between systems. Nevertheless, a few main points can be identified regarding their practical use.

The first and most obvious point is that it straightforward to construct class or object hierarchies to store general descriptions in a relatively structured way. Useful examples include the classification of equipment types, classes of chemical compounds, etc. Flexible systems, such as CLAP, allow other types of entity to be cast into hierarchies. CLAP uses a hierarchy of relation types to classify different

types of constraints, equations, etc.
From a programming point of view it is undoubtedly the structured storage aspect of objects that is most useful. A complex problem can be characterised by first identifying the different types of entity necessary to describe the problem. This is just an example of good programming practice involving the use of abstract data types. Languages that support abstract data types, eg Modula-2, Ada, recommend that all manipulation of data types, eg setting or checking values, be done through a defined set of messages, ie a protocol. The implementation of a data type and its protocols is then completely insulated from the rest of a program. This results in significant benefits when writing and developing large, complex programs.

High level objects retain all these benefits with the addition that individual slots or attributes of objects can usually be further described by meta- information. This information might simply describe the allowed value types for a slot or it may express complex dependency information between a slot and other slots, possibly in different objects. The use of this type of meta-information is a first step towards the ability to maintain the integrity of data across sets of objects.

The use of slot demon functions for control or side-effect purposes should be kept to a minimum since the "knowledge" being used is implicit or "hidden" from any global reasoning strategies. If demons, either of the before or after kind, are restricted to checking explicit relationships, inter-dependencies, etc then they too can help maintain simple forms of data integrity within an environment.

An emphasis on the use of objects rather than procedures to characterise a problem also allows external procedures or programs to be interfaced to an environment in a much more logical way. Specific functions can be "attached" to individual slots or objects as a whole and organised around the object data hierarchy rather than vice-versa. Functions or programs perform local operations of well defined data structures. There is no need to maintain complex links or execution paths between each function. This style of code organisation is becoming widely appreciated especially in the context of large programming developments, eg see Weiner and Pinson(1988).

It should be noted that all of the benefits concerning the use of schema to describe abstract entities and structure data within a process model, as outlined in chapter 1 regarding database management systems, can be equally applied to many 0.0. systems. The programming styles and techniques available are, of course, very different between current 0.0. systems and languages used in earlier DBMS's.

3.5.3 Logic Based Representations

The most widely available logic programming tool is the Prolog interpreter. The interpreter is not a sophisticated theorem prover itself, being based on Lush resolution on Horn clauses, but can be used as the core language in which to implement other more sophisticated systems. The development of CLAP is a typical example that makes full use of Prolog's built in unification and backtracking search facilities.

The most obvious use of first order logic representation systems in process design is in expressing and reasoning about complex relationships between entities. Few, if any, of the rule-based or frame based systems have the capability to express full first order logic formulae or expressions. In particular, existential and universal qualifiers are poorly handled, as is the treatment of negation.

It is generally assumed in the literature that some form of logic representation is necessary, either implicitly or explicitly, in sophisticated knowledge representation systems. Further discussion is delayed, however, until chapter 5. Since no comprehensive, logic based systems are available commercially the best short-term solution appears to be the use of Prolog in conjunction with other representation styles in a hybrid system, eg CLAP.

3.5.4 Procedural Methods as Declarative Data Structures

There is nothing new about the use of procedures and algorithms in process engineering software. What is new in the development of methods in CLAP is the emphasis placed on treating "procedural information" in a declarative way that combines the features of both logic and procedural programming.

A CLAP method is treated as an explicit object or item of data, the contents of which can be reasoned about by other methods or inference techniques. This makes it easy to specialise the "code" or "body" of a method in a problem dependent way. Most importantly, it also means that the solution technique, assumptions made, etc, can be made readily accessible to other decision making methods.

The evaluation or interpretation of a procedural-like method is seen more as an exercise in theorem proving or satisfaction of constraints rather than a blind execution of sequential steps. This approach is greatly enhanced by the use of CLAP relations and Prolog unification within a method definition.

It is undoubtedly true that valuable aspects of design expertise are most naturally expressed in a procedural manner. It seems essential therefore that a flexible high level language be used to express these ideas in a way that can be explicitly reasoned

about by other inference mechanisms.

The continued use of large, self-contained algorithmic programs is of little long-term value. The development of more sophisticated systems requires that external code be gradually reduced to small, very well defined operations that are free from global side effects, eg multiplication of arrays, etc. Even at this level the implicit functionality should be represented elsewhere in an explicit form, e.g. the definition of the Gaussian elimination procedure, or any variation thereof, Newton's method, etc. Many problems can be formulated in an apparently correct low level form only to find that convergence does not occur because of numerical approximation errors or instabilities. A lengthy tracing or debugging session by hand is usually necessary to locate and resolve the problem. This situation could be handled much more satisfactorily in a system like CLAP by reasoning about non-convergence in an explicit way as part of another method.

Extended methods are unique to CLAP and are directly concerned with expressing high level design strategies in a systems engineering framework. Their relevance to immediate use in a process design environment is therefore obvious.

3.5.5 Blackboard Systems

The immediate benefits to be gained from use of blackboard systems are not due to any representation style, since any representation technique can in theory be used in each knowledge source (KS), but due to the structuring of knowledge, procedures and data into related "chunks" or modules. In many ways this is exactly the same argument as was made for structuring information within objects.

The basic difference between KS's in a blackboard system and objects in an 0.0. system is that messages are not made on a direct one-to-one correspondence. All messages are sent to the blackboard and the activation of any particular KS to process the message further is at the discretion of the agenda scheduler. The scheduler is typically opportunistic in nature but there is no reason why an alternative planning strategy, eg the use of non-linear planners, could not be used to control the activation of KS's.

The opinion taken here is that a blackboard architecture is well suited to partitioning large, complex, ill-defined problems at a rough grain size but are less appropriate for smaller, more well defined tasks. The latter type of problem is more appropriately represented within a single KS where tighter control can be kept on any search or problem solving strategies.

3.6 An Overview of AI applications in Chemical Engineering

The time of writing this review on other related AI research in process engineering considerably post-dates both the work described so far and the remainder of the research work in this thesis. It is with considerable hindsight therefore that the following comments are made.

The main aim of this thesis is to investigate the use and suitability of knowledge representation techniques for use within process engineering design. Since the period 1984-1986, which covers the research of the previous chapters, a large number of papers have been published pertaining to the use of "expert systems" in process engineering. As it turns out many of these papers are of a preliminary, investigative nature and have little to contribute to the study of knowledge representation.

Nevertheless, many of the papers are still interesting if only to demonstrate the potential range of applicability of these ideas. Examples of such papers include Bingzhen et al (1988) - an expert system for heat exchanger network synthesis, Earl and Williamson (1988) - control system synthesis, Niida et al (1988) - process fault diagnosis, Wetherhill and Cameron (1988) - preliminary hazop studies. These and other papers are not reviewed here for the reason stated above. References, however, to other reviews which do go into more detail are given later.

What is of more benefit to this thesis is a general overview of the representation techniques that have been investigated by other workers to date. References are given for papers indicative of each approach.

The basic areas of research that can be identified in the literature are:

1. Rule Based Production systems
2. Blackboard systems
3. Frame/Object Oriented systems

It is fair to say that work in each area has concentrated on the use of a representation style for a particular, isolated problem. It is not until very recently that the wider implications of representation methods have been considered, see Henning et al (1989) and Piela (1989) below. It is, of course, this wider context that is of prime concern in the development of an integrated process design environment.

The development of production rule systems and blackboard systems are well illustrated by a group of related papers. Banares-Alcantara and Westerberg (1985) describe a rule based system to aid the the selection of appropriate vapour liquid equilibrium methods. The rules are represented in an inference network in the

CONPHYDE system, which is a close derivative of PROSPECTOR (see section 2.5.3). This work is very similar in nature to that described earlier in section 3.1 on heat exchanger selection.

A more sophisticated development is DECADE, see Banares-Alcantara (1986) This system is based on a blackboard architecture and is concerned with the selection of catalysts for a specified single step reaction. The overall model is based on opportunistic planning and solution of goals that are posted on the blackboard. Some of the knowledge sources involved include thermodynamic feasibility testing, reaction specification (user interactive) and reaction classification. The implementation is interesting in that it uses a hybrid approach. The OPS5 system is used for rule based pattern matching while the knowledge sources themselves are organised around frames in an inheritance hierarchy using the SRL language, see Wright and Fox (1983). The overall system is driven within a LISP environment. The representation techniques within DECADE are, however, somewhat limited. A more general discussion of the use of both goal driven and data driven architectures within an opportunistic blackboard system is given in Weke, Lien and Westerberg (1987). This describes an outline of the use the AKORN-G and AKORN-D systems in the context of design of separation systems. A somewhat different rule based system is that of Kirkwood, Locke and Douglas (1987). The PIP system is concerned with the heuristic synthesis and evaluation of process flowsheet alternatives. The system structure is one of a hierarchical decision tree, searched in a depth first manner, that is closely coupled to a library of utility calculation procedures based on short cut modelling techniques. As an example of an extended heuristic graph searching program it is of some interest but is of little value from an explicit knowledge representation point of view.

In the use of object oriented environments the work of Stephanopoulos et al (1987) is typical of many subsequent papers. That paper describes the ideas involved in DESIGN-KIT, an application implemented using the commercially available KEE toolkit, see KEE (1986). It is shown how objects and slots in KEE can be used to represent typical plant items and sets of modelling equations and how KEE methods can be attached to slots to perform various types of calculations. Most of this functionality is directly available in CLAP, see section 3.4.

An extended discussion and criticism of frame based systems is given in chapter 4 so further comment is not provided here. Another notable application is that of Vancoille and Bogaerts (1989), also Vancoille, Bogaerts and Perdieus (1989). The system is called PRIME and is implemented on top of KEE. It deals with a large range of material selection and corrosion problems classified under different areas of

process technology, eg oil refining, agrochemicals, etc. It is interesting since it is one of the few process engineering systems that has been implemented on a large scale and is operational.

Finally, there are two recent works that are of direct relevance to this thesis in that they propose the use of general modelling or representation languages. The first work is that of Piela (1989) and the ASCEND system. In this work Piela describes a modelling language based on objects within an inheritance hierarchy and a set of tools for manipulating the resultant structures.

The modelling language is in fact very simple compared to other AI toolkits. This is because it was written directly for the purpose of generating process models in an equation oriented fashion and their subsequent solution. The basic construct is labelled a MODEL and can be thought of as a simple kind of object. The main features of the language are:-

1. REFINES - one model refines another more generic one. This provides model inheritance.
2. IS A - a term within a MODEL can have a type associated with it to provide access to parts of other model constructions.
3. ARE ALIKE, ARE SAME - mechanisms used to provide partial and total merging of variables in models that refer to other model structures.
4. IS REFINED TO - this provides a deferred, or late/run time binding mechanism for instantiating terms in a model.
5. UNIVERSAL - provides a mechanism to declare global variable values.

The following code fragment shows a model of a tank draining problem that is to be solved by the Runge-Kutta method.

```
MODEL tank-draining-problem
   ms              IS A    equal multistep;
   /* ie an integration method*/
   ms.h            := 0.2 (mins)
   ms.n steps      :=   8;

   ms.init state   IS-REFINED TO tank model;
   ms.steps [1]    IS REFINED To Runge Kutta 4:

   ms.init state.y[1] := 10 (ft^3)
   ms.init state.y[1].fixed := true;
END tank draining problem;

MODEL TANK model REFINES FUNCTION evaluation
...
   y[1], vol        ARE THE SAME;
   y prime[1], dvol dt ARE THE SAME;
...
   dvol dt + =  F out:
   F out = params. k* 1.0 (gpm) * (ht/(1.0 (ft))) ^ (1/2);
   ht = vol/params.a;
END tank model;
```

The code is sufficient to show how models can be constructed in a hierarchical way and instantiated at run time. Piela discusses the way in which equation solvers are

interfaced to ASCEND to solve such problems. In many ways ASCEND is better considered as a 4GL modelling language. In comparison with the work here it is clear that the relations facility in CLAP, both for specialising generic forms of relation, ie equations, and evaluating specific relationships, can be used to directly emulate the modelling capabilities of ASCEND. The work of Hutton (1990) has demonstrated this in practice. The advantage of CLAP is that it retains a number of other representation and programming mechanisms that can be used for different sorts of problems.

The second, and final work to be described is Henning, Leone and Stephanopoulos (1989 a, b) and represents a development of DESIGN-KIT. The work involves the development of a modelling language called MODEL.LA. The language is concerned with modelling two areas:

1. Modelling objects and their describing relationships in design states.
2. Modelling procedural design tasks.

In each case a restricted set of terms are provided to let more complex objects or procedures be constructed in a systematic way. Those for the object descriptions are listed below:

1. is a : e.g. PLANT is a Generic Unit
2. is a member of : e.g. R101 is a member of CSTR
3. is composed of : e.g. Heat Exchanger is composed of (TUBES, BUNDLE, SHELL)
4. is part of : inverse of composed above.
5. is attached to : e.g. Feed F 1 (port) is attached to S 10 (stream)
6. is connected by : inverse of attached above
7. is described by : e.g. F 1 (unit) is described by (LUMPED MASS BALANCE, LUMPED ENERGY BALANCE)
8. is describing - inverse of described by above.
9. is disaggregated in : e.g. TO STRUCTURE is disaggregated in (REACTION SECTION, SEPARATION-SECTION)
10. is abstracting : opposite of above

Similarly there are ten task items used to formulate procedures, eg SELECT (some alternative), IDENTIFY (information required), EXECUTE (some procedure) etc. The language represents an improvement implementation details of over the DESIGN-KIT in that each term has an explicitly defined semantics. This should help prevent ad hoc interpretations of objects and their slots in an inheritance hierarchy (see chapter 4). The implementation of object and procedural tasks in MODEL.LA is very close in spirit to the approach taken in CLAP. The main difference between the two, apart from CLAP's special equation generating facilities, is that CLAP does not insist on the use of a predefined set of operators to describe objects and procedures. Using the operators available in MODEL.LA it remains to be seen how much of a

procedure can be represented in explicit terms and how much will remain hidden as side effects within LISP code. The main contribution of the work is that it attempts to bring some order and formal analysis to the otherwise ad hoc and haphazard use of object oriented systems.

Finally there are a number of other reviews and collections of papers describing applications in chemical engineering and other engineering disciplines. Two early reviews are Sriram (1984 a, b). A later review is Hutton, Ponton and Waters (1990) and a useful collection of related papers is Reklaivis and Rippon (1988). Some useful discussion on the general applicability of AI techniques to process engineering can be found in Lien, Suzuki and Westerberg (1986), Stephanopoulos (1989), and Sargent (1989). Venkatasubramanian and Dhurjati (1987) is useful for its criticism of rule based representations. In a different vein both Huang and Far (1988) and Motard (1988) provide useful comparisons between relational database systems and object oriented databases.

Chapter 4
Towards an Intelligent Process Engineering Design Tool

The initial research described earlier in this thesis focussed on the use and development of state of the art symbolic programming techniques and their potential application in a process engineering design environment. Three main developments were presented:

1. The use of a rule based production system in a typical design selection problem, i.e. the initial choice of the type of heat exchanger to select for a given duty. A generic framework more suited to engineering design problems was outlined. The use of the rule interpreter is embedded within "context setting" clauses which can be used to perform "before" and "after" calculation. This avoids any unnecessary forward and backward chaining/search. Much of this information is more readily expressed in procedural form and does not need to be represented in a cumbersome rule format using the given framework.
2. The use of a model based rule interpreter within the wider context of a module operating in a blackboard system called Designer's Assistant. The rule interpreter, concerned with the selection of physical property estimation techniques, was significantly more sophisticated than the earlier version used for heat exchanger selection. In particular, compromises on the application of rules to select estimation techniques and the availability of data can be easily expressed.
   The interpreter makes use of a taxonomic classification of chemical compounds that allows generic properties to be defined at any level and supports simple inheritance of these properties by sub-classes. This allows generic domain dependent data to be defined in a structured and concise way. A "process model" is built up during problem analysis and represented in partitioned areas of a common blackboard corresponding to a system, its streams, units, etc. The interpreter incorporates a simple localised planner, to sequence the order in which requested goals should be searched, and a two pass advice and evaluation search strategy. At any point in a search the interpreter can make a request either to other modules in the system or the end-user using a standard module interface. This means that large complex problems can be broken down and solved by a number of smaller, better defined modules that can interact with each other. This potentially allows alternative, specialised representation and inference techniques to be used in different modules where appropriate.
   A first attempt was made to solve the fundamental problem of arbitrary tokenism that occurs both in intra-module and inter-module interpretation of information, e.g. rule conditions, process model assumptions etc. This involved the definition of a module language that describes every token referred to in terms of simple dependencies on other tokens, attached procedures for evaluation, its representation in a process model, etc. This information is used to help guide the interpreter throughout a search process and could be used by other module interpreters in an attempt to maintain consistency in the use of tokens in different modules.
3. A very flexible hybrid programming environment called CLAP (Combined Logic and Procedures) was developed. The system combines many features of object orientated, procedural and logic programming styles in a high level way. The use of message passing, multiple worlds, procedural methods, interfaces to "external" code, theorem proving techniques etc, can be freely intra-mixed depending on the most appropriate way to solve a particular problem.
   CLAP has a unique relational facility that allows generic process constraints, equations, etc, to be easily defined and then automatically specialised and set up

for solution for a specific problem. CLAP can maintain multiple solutions of these constraints for later use if necessary. This makes the process of constructing alternative unit operation models a very simple one.
Another interesting feature of CLAP is the use of meta-view points to allow multiple points of view on the "value" of a slot or relation of an object to be represented. A formal mechanism to define both meta-slot and meta-relation information is provided. This meta-information can be used to reason about complex dependencies in a similar way to the use of the module language token definitions in the blackboard interpreter. It is left to the system programmer to augment CLAP's default interpretation techniques where necessary.
Although CLAP is a generic programming tool it has two additional features specifically intended for use in process design applications. The first of these is the use of procedural like methods to express low level, well defined design procedures or algorithms. They are interpreted, however, in a theorem proving like way using backtracking search, unification etc. This allows methods to be expressed in a very concise but generic form that can be later specialised in the solution of a particular problem. The second feature is the use of extended methods to represent high level design methodologies. These provide the necessary framework to allow different design strategies to be formulated and described from a systems engineering point of view. This includes the use of AIP based design strategies as introduced in chapter 1.

It is clear that given these initial developments any number of more detailed demonstrations concerned with different aspects of process design could have been developed to further illustrate the programming techniques involved. This is the approach that has been taken by other researchers in this area, see section 3.6.
Alternatively, the concept of an AIP design methodology could be developed much further using the features of CLAP embedded within a module of the blackboard framework of Designer's Assistant. The blackboard entries could be easily modified to a set of CLAP objects that could be used to pass information from module to module. Each module could use CLAP as its base interpreter since other inference techniques, e.g. rule based interpreters, are easily embedded within CLAP. Existing software could also be directly interfaced with CLAP objects using external type methods if necessary.

Neither of these options were in fact pursued further. The purpose of the rest of this chapter is to highlight some of the deeper issues involved and provide an initial explanation of why further research on representation techniques was deemed necessary.

1 Reassessment of the Requirements for an Integrated Process Design Environment

The concepts of integrity of data, knowledge and integrated process design were defined in chapter 1. These definitions are repeated here in abbreviated form for convenience:

- Data Integrity - the consistent use of data in design activities in terms of the

availability, uniqueness of value and explicit functional constraints imposed on the data .... the control of data should explicitly take into account the way in which that data is used in related process design activities.
- Knowledge Integrity - An explicit representation of the assumptions and principles upon which the data is based as well as an explicit knowledge of how that data can be used and manipulated in terms of other data dependencies, both of which must be consistently maintained throughout the design process. This can be thought of as being consistent about the reasons behind data constraints which in turn requires awareness of the background knowledge, theory and decisions responsible for the creation of the data.
- Integrated Process Design - An explicit representation of the principles of overall management of a design project in terms of the design activities and associated methodology. This methodology must incorporate the notion of knowledge integrity into its formulation.

Current database management systems were considered and it was concluded that they were far from adequate in terms of providing a programming environment to ever achieve these goals. Three basic levels of representation were identified, see section 1.7, that are needed in an integrated process design environment:

1. A means to both represent and interpret the basis and intended effect of operations which manipulate data. This involves maintaining known assumptions and functional dependencies.
2. A means to represent, reason about and control an overall design methodology whether at a project management level or a low level problem solving level.
3. A means to represent the fundamental principles upon which "data" is based.

The types of data and knowledge representation needed for each of these levels were discussed throughout chapters 2 and 3. This included:

1. Descriptions at an object, attribute, relationship level.
2. Description of problem solving techniques in regard to the use and consistency of the information expressed in 1).
3. Explicit representation of an overall design methodology and its relation to particular decision making events.

This effort was in direct response to the research goals stated in section 1.9.

The initial developments of a rule based production system, Designer's Assistant - a blackboard based system, and CLAP - a hybrid object orientated, procedural and logic programming tool, were attempts to investigate the use of state of the art programming techniques in AI to express this sort of information, design expertise, etc. In order to appreciate more fully the processes involved, several of the programming methods were redescribed in standard logic based, theorem proving formulations. This allows the different styles to be understood in terms of common, clear and precise semantics.

After careful consideration, the main conclusion to be drawn from this early work is that the basic structure or framework of Designer's Assistant is a potentially

appropriate one for use in a process design environment if it is embellished with additional features such as CLAP relations, low level methods and extended methods. What is clear, however, is that the individual interpretation techniques developed so far are totally inadequate to satisfy the basic requirements of an integrated design environment as stated above. In order to explain why this is so, a number of general observations concerning the work so far, and that of other researchers in AI, must be given. This will start with a statement of the key remaining problems still to be solved, section 4.1 below. Sections 4.2 through 4.4 briefly highlight the main flaws associated with the direct use of a system like CLAP or the physical properties rule based interpreter in an integrated process design environment.

4.1 Fundamental Research Objectives

It is easy to see how simple systems like Designer's Assistant and the CLAP interpreter could form the basis of a structural, modular process design environment. It is the author's belief, however, that the problems of producing a truly intelligent computer aid or environment for solution of a complex task involving a large and diverse domain of knowledge with complex but systematic interrelations, as found in process design, go far beyond the scope of any current "expert" or "knowledge based" system.

It appears that two fundamental objectives have still to be achieved:

1. A unified and systematic representation of fundamental, heuristic and procedural knowledge and a consistent framework for its interpretation must be devised to cope with the domain of process engineering. The breadth of this domain and complexity of interactions within it precludes the use of the simplistic representations of current database management systems as a practical option. The depth and complexity of decision making involved would appear to make the interpretation methods of simple rule, frame and hybrid based systems seem wholly inappropriate and inadequate. A unified representation and interpretation is clearly essential if the problem of arbitrary tokenism is to be avoided as knowledge is shared between many diverse tasks and engineers in a design.
2. Intelligent engineers dealing with complex phenomena, such as chemical plants, appear to derive their understanding from a localised world picture or "mental model". (This is suggested by recent research in cognitive science and psychology as explained later). This model or understanding of the situation will be updated and refined by knowledge from various sources as a design proceeds. The mechanisms for constructing, updating and reasoning about this "fundamental model" are crucial to the development of any truly intelligent designer's aid. (The term functional model is defined later as part of an approximate representation of a mental model).

It is not believed that current representation techniques offer a satisfactory long-term solution to these problems. Section 4.1.1 below illustrates this by indicating some of

the basic features or capabilities required by any intelligent, problem solving agent.

4.1.1 Intelligent/Expert Systems

The use of the term "expert system" has been consciously avoided in the discussions so far. It was originally used to describe systems such as MYCIN and PROSPECTOR but now appears to be used to cover almost any sort of program involving symbolic programming techniques or search algorithms, both in academic literature and popular press. There is no single definition of the term but the following quote is typical: "An expert system is one that -

a) handles real-world, complex problems requiring an expert's interpretation, and

b) solves these problems using a computer model of expert human reasoning, reaching the same conclusions that the human expert would reach if faced with a comparable problem", Weiss and Kulikowski(1983).

This definition is particularly unhelpful on a number of counts:

- "real world, complex problems" and "an expert's interpretation" could be taken to mean just about anything. It gives no indication of the type of problems that can be solved, what aspects of expertise are required, etc.
- Vague references to "computer models of human reasoning" are purely wishful thinking. The techniques currently used are simple symbolic search and theorem proving techniques as described in chapter 2.
- To say a program reaches the "same conclusions" that an expert would is particularly naive. A program may conclude some statement represented in a symbolic form that can be treated or interpreted at a superficial level to be equivalent to some conclusion given by a human. It is certainly not the "same" conclusion since this would require taking into consideration the whole context in which the expert's conclusion was made, e.g. any supporting arguments, assumptions made etc. This contextual information is rarely, if ever, enunciated in a detailed form. Design manuals, for instance, do not indicate the many forms of reasoning or arguments that were used to make a particular decision.

What can be said about current "expert systems" is that they are programs which can perform useful tasks based on certain problem solving heuristics, usually expressed in symbolic form. The knowledge, heuristics, data, or whatever, that is used is only a small part of an engineer's expertise. Without appropriate forms of reasoning to interpret this knowledge in a variety of ways, then any claims regarding the intelligence or expert ability of a system are totally misleading.

Consider for the moment an intelligent agent, whether it be an undergraduate, junior process engineer, consultant, etc, that is to assist in a process plant design. The following features will be fundamental for its successful use:

1. Understanding - a system must have some understanding of its chosen domain, ie at least an appreciation of the basic principles of a subject, how they can be applied to a particular problem both in terms of problem formulation and

solution, how they relate to other domains of interest, etc. This is particularly true in front-end engineering design where the ability to apply appropriate principles to a problem is an essential part of the problem solving process.

2. Learning - any expert or intelligent agent must have the ability to learn about and expand its knowledge on the particular domain of interest. Engineers constantly learn of, and take advantage of new research ideas, unit operation types, reactor catalysts etc. Their depth of understanding is continuously modified with the experience gained in solving different types of problems. Given the breadth and complexity of a domain such as process engineering it seems inconceivable that any computer system could ever be programmed in a "final form". Indeed, the more "expert" a system becomes the more frequent the need to learn about new information and how it relates to previous ideas and results. What is of particular importance is that this learning is done in a self-consistent and incremental manner.
3. Explanation - Any intelligent agent must be capable of explaining its reasoning regarding conclusions it has made if it is to be of any practical use. The form of explanation required goes far beyond that available in most current systems. For example, a simple traceback of rules that fired in some search process is of very limited use. A novice engineer who does not understand the ideas on which these rules are based will gain little, if anything, from such an "explanation". Attempts by others, e.g. Clancey(1983), to use meta-information to provide more coherent explanations represents a move in the right direction but is still far from adequate.
4. Cooperation - the sheer size of most engineering design problems and the number of considerations that have to be taken into account inevitably means that engineers must cooperate and exchange information, ideas, etc, in the solution of a problem. This is an interactive process in which design goals, requirements, etc, must be clearly stated and agreed upon by the parties involved. Any truly useful intelligent design aid must be capable of participating in this type of activity.

Features 1-4 represent formidable research goals if they are ever to be implemented in a design environment. Even if they are not implemented they are still essential considerations in achieving the simpler objectives stated on page 269. The implication is that constructing an intelligent process engineering design and will be very difficult. Consider, for instance, the variety of tasks involved, the range of knowledge required, and the conceptual skills required even to formulate artificial undergraduate tutorial problems, let alone to construct their solutions (see also Figure 1.7).

What appears to be required then is a system possessing basic conceptual skills that can help the design engineer cope with variety and complexity of problems that are encountered in design projects. These skills will be required to meet the two research objectives stated earlier: namely the ability to construct, reason about and update a functional process model within a consistent framework for its interpretation.

It is worth noting that the problem of arbitrary tokenism, defined in section 3.3.6, represents in many ways the tip of the iceberg as far as the problems that items 1-4

cover. The problems of in-depth understanding of a subject, an ability to learn, clear expression and explanation of reasoning, and the ability to cooperate interactively are all inextricably linked. Their solution will require a common, sophisticated internal representation of language "tokens" that facilitate in some way the operation of a number of very different but internally consistent interpretation mechanisms.

4.2 Deficiencies of Rule and Frame Based Systems

Frame and rule based systems have been extensively discussed, both in chapter 2 and elsewhere [e.g. Jackson(1986), Johnson and Keravnon(1985), Barr and Feigenbaum(1981)]. The following discussion only summarises the features of both types of system that are directly relevant to their unsuitability of use in the development of more sophisticated process engineering design and analysis tools.

4.2.1 Rule Based Systems

A good deal of both industrial and academic research is still based on the use of "if-then" rule based production systems. It is quite clear from the research here and that of others, however, that the variety of types of engineering knowledge precludes representation in any single form. In particular, production rules are much too limited or inappropriate to cope with much of this knowledge. Algorithms or design procedures, for example, are not well represented in rule format. Despite this fact much of the published research has involved such a knowledge formalism in one form or another and has consequently been of a highly constrained and simplified nature, e.g. early attempts at heuristic synthesis, Lu and Motard(1985), synthesis of operating procedures, Fusillo and Powers(1988).

There are two fundamental flaws with this type of system. The first is that without a token language definition/dependency facility much of the domain knowledge remains implicitly rather than explicitly encoded in the rules. This is typically seen in ordering rule sets in specific ways, ordering rule antecedents, use of Lisp/Prolog code in rules and other implicit forms of procedural attachment. Many of these "tricks" rely on knowing in advance how the rule interpreter will work in detail. The system is never aware of this implicit reasoning, which is vital for the successful use/combination of the rules, nor of the principles or concepts that are represented by the symbols in the rules. As a consequence it is very difficult to ascertain functional dependencies between symbols or tokens and how they should affect the systems inference mechanisms. Relying on a particular interpreter's search strategy and use of arbitrary tokenism has serious consequences for any design tool trying to tackle the problem of consistent usage of data and the assumptions inherent in that data. Consider the following questions:

1. What happens if a new search strategy, or even worse, a new rule interpreter is required ? It is very unlikely that there will be detailed information available regarding the reasoning behind why a rule set was written and ordered in the way it was. Manual updating of rule sets becomes very difficult and automated learning virtually impossible.
2. How are new rules added to previously "ordered" rule sets if the reasoning

behind the original ordering is not explicit?
3. How can the consistency of a set of rules be tested? Are exhaustive theorem proving techniques required, and if so, how practical is this for large rule sets?

These problems are serious indeed. The second flaw results in a problem in many ways even more important, and has to do with the inference capabilities, and hence problem solving capabilities of production systems. Following the second objective stated on page 270, it is believed that the solution of many non-trivial design problems generally proceed according to the following mechanism:

1. A problem must first be recognised and formulated from the wider context in which it is (implicitly) stated. This is a fundamental analytic/conceptual skill that an engineer must acquire. Problems are rarely stated in simple terms. Rather the engineer works with general design goals in mind and must formulate for himself problems to be solved that will help in satisfying these goals.
2. Partial models must then be set up and solution techniques to solve particular problems selected. Partial models are continuously expanded and updated as intermediate conclusions are made.
3. The model must be continuously verified to check for consistency. This includes an analysis of any initial or intermediate assumptions that are made in the solution of a problem and how they affect other assertions represented in the model.
4. As a result of solutions to intermediate goals, improvements or suggestions must be acted upon to help achieve the long term goals in the most satisfactory way. This means that any intended plan of action must be periodically re-evaluated.

A number of complex conceptual skills are required to carry out these steps. Many rely on some insight into the principles of the subject, an understanding of the problem at hand and an ability to apply the former to the latter. The notion that this approach to process design or analysis can be replaced by a "shallow", surface level rule based representation combined with simplistic pattern matching and search/inference techniques is untenable in all but the most trivial examples of decision making in process engineering.
Simple chaining inference techniques, with or without the use of conflict resolution strategies such as those found in OPS5[Brownston et al(1985)] or context layering techniques, are both inadequate and inappropriate in their current form.

These observations, in conjunction with the experience gained in the development of CLAP and Designer's Assistant, lead to the conclusion that rule based production systems cannot be considered as the primary form of interpreter in an advanced design environment.

4.2.2 FRAME BASED SYSTEMS

In an attempt to express more subtle forms of knowledge such as prototypical or default information, or exploit the natural hierarchical structuring of knowledge in many problem domains, several systems emerged that either combined or replaced rule based representations with frame based or other object orientated styles, e.g. CLAP (chapter 2), KEE(1985), LOOPS (Bobrow and Stefik 1983). The common theme in these systems is that both data and procedures can be organised into structures related by some form of inheritance mechanism. Stephanopoulous(1987), for example, has suggested that these types of system be used as the basis for "knowledge based" engineering software tools.

It is the author's opinion, however, that frame based systems are flawed in much the same way as production systems as regards achieving the basic research objectives set out earlier. Indeed, most of the previous criticisms are directly applicable to hybrid environments such as CLAP, KEE, etc. There are a number of other deficiencies, however, that derive directly from the use of inheritance techniques.

The following discussion is a brief summary of some of the theoretical problems presented in the AI literature. This draws heavily on the seminal papers by Hayes(1979(a)) and Brachman(1983, 1985) but also on the insight gained from the development of CLAP (it is worth noting that CLAP supports most of the features described below in modified forms plus several of its own which, to the author's knowledge, do not directly exist in other systems, e.g. CLAP's form of relations, extended methods, unification with object/procedure definitions, extended message passing, etc).

4.2.3 The Basics of Frames

The term "frame" was introduced by Minsky(1975) to bring together a loose collection of ideas on knowledge representation. However, it was not clear then what theoretical status frames had and has since become even less clear due to the diversity of treatment in particular systems. Nevertheless, following Brachman(1983a) the basics of most frame languages appear to be:

- Objects are represented by frames which are taken to be non-atomic descriptions of some complexity.
- frames are defined as specialisations of more general frames
- individuals are represented by instantiations of generic frames, and
- the resulting connections between frames form taxonomies, i.e. class_subclass hierarchies. The link between a parent and child class is usually labelled an "IS_A" link.

The commonly recurring theme of the use of a taxonomy of structural descriptions in

a representation language is an intuitively appealing one. As Brachman (1983a) points out, the resulting data structures must have an interpretation imposed on them, i.e. they must "mean" something, if they are to be of any practical use. In practice, many simple systems bypass this problem by leaving the definition of the semantics to the programmer, or user. This is a totally unacceptable situation and has resulted in much confusion in the literature. Brachman(1983b) is a good introduction to the large number of possible ways in which the IS_A links between nodes in a hierarchy could be interpreted (or misinterpreted).

The discussion below concentrates on the two main approaches to interpreting frames and the more serious problems that arise. The first approach, the "assertion point of view", interprets frame descriptions as being assertions or statements about the items in the world being represented, e.g. the link p101 IS_A pump appears to be an assertion regarding item p101. Conversely, the second approach treats frames purely as descriptions with no direct assertional import, i.e. no assertions can be assumed merely due to the presence of links between nodes. Both approaches will be briefly considered in turn.

It is important to emphasise that terms such as "frame", "inheritance", etc, are being used in a representative way and are based on descriptions of typical, well known systems, e.g. FRL (Roberts and Goldstein, 1977), KRL (Bobrow and Winograd, 1977), NETL (Fahlman, 1979), KEE(1985). A basic understanding of the concepts involved is assumed below.

4.2.4 Frames - An Assertional Viewpoint

An important contribution to the study and use of frames was Hayes(1979). In this work Hayes attempts to describe frame based systems in terms of first order logic assertions in order to establish a clear understanding of the semantics of such systems. This analysis is briefly summarised below. Given this insight, and that of Brachman(1983) later, a number of serious problems will be identified that severely compromise the usefulness of frame based systems.

Hayes used the KRL system (Bobrow and Winograd, 1977) as the basis for study. KRL was one of the most ambitious early frame based systems and is notable for two reasons:

1. It was one of the first systems to state precisely how and why to use procedural attachment and meta_descriptions.
2. It was inspired by research in cognitive science on the use of prototypical concepts, i.e. prototypes, in human reasoning.

The use of prototypical information was based mainly on research by Rosch and Mervis(1975) and introduces important ideas that are discussed in much more detail in the following chapter. Details of KRL are therefore provided where necessary but see Bobrow and Winograd(1977) for full details.

4.2.4.1 Semantics and Representation Languages

There is much confusion in the literature as to the point or purpose of "representation languages" and the definition, or lack of definition, of the associated semantics. Indeed the word "semantic" is greatly overused, often mis-used.

Hayes characterises a representation language as "one which has (or can be given) a semantic theory". This in turn means "an account... of how expressions of the language relate to the individuals or relationships or actions or configurations, etc, compromising the world, or worlds about which the language claims to express knowledge". In order to give such an account or semantic theory a number of metaphysical assumptions, i.e. assumptions concerning the basic types of entities, must be made. For example, the ontological commitment needed to understand predicate calculus requires the assumption that the world consists of individual entities and relationships that hold between them.

A semantic theory "defines the meanings of expressions of the language", ie "expressions in the language carry meaning". Bundy(1983) emphasises the importance of a well defined semantics for without it becomes difficult or impossible to decide how to represent knowledge in a formalism, or decide whether described procedures are reasonable, or interpret what a given process does.

In chapter 2, see also Appendix B, the "semantics" of first order logic is defined to be the set of interpretation laws for formulae and are denotational in nature. The purpose of a semantics is therefore to assign meaning, truth values here, to well formed sentences. The important point to make is that any semantics should provide clear, unambiguous interpretations.

Given this understanding, Hayes identified three ways in which to consider frames:

1. As a means to design a formal representation language
2. From a metaphysical viewpoint - that is, to make an assumption that certain "kinds" of knowledge are to be represented by frames. An example of this is the use of frame-like structures, called scripts, to represent stereotypical events or situations such as ordering a meal in a restaurant, see Schank(1975), Charniak (1977). The emphasis here is on what kinds of things a system should reason about rather than how they can or should be represented.
3. From an implementation viewpoint - that is, that frames are simply a programming technique for storing representations and organising the retrieval and inference processes that manipulate those representations.

The metaphysical viewpoint, although interesting, was not treated further by Hayes

since it is not concerned with the actual representation of ideas. The sequence of events in a script, for instance, could be expressed in any reasonable formal language, e.g. predicate calculus.

"Similarly, implementation issues are not central to the discussion. Any representation language can be implemented in a number of ways. For example, assertions in predicate calculus might be implemented as a linked list structure (as in Lisp or Prolog), or as a string, or as a tree or graph structure, or as part of an array, etc. As Hayes points out, if representation is confused with implementation then Lisp or Prolog could be taken as universal representation languages ending all further discussion.

4.2.4.2 Interpretation of Frame Expressions

Frames are typically data structures, or forms of expressions, that represent stereotypical objects or situations. They contain named slots that can be filled with other expressions, "fillers", which may themselves be frames or identifiers associated with other frames. Particular objects are represented by "instances" of the generic frame and require that the slots be filled with appropriate filler values, e.g. identifying the value for the 'number_of_tubes' slot in a 'heat_exchanger' frame. Slot values can be left unspecified to represent the expression of doubt or can be associated with a number of further constraints or refinements, e.g. to represent justifications of the value.

It appears, therefore, that in many systems frame instances simply denote "individuals" and that each slot denotes a relationship that may be defined between an individual and some other. This can be simply expressed by an assertion in predicate calculus. For example, a slot "upstream" in a frame instance of a pump, called P1 say, with slot_filler value of R1, R1 being an instance of a reactor, can be represented by the assertion:

upstream_of(P1,R1)

If this is the case then Hayes points out that frames are essentially grouped collections of properties that can be re-expressed in terms of first order logic. This allows the meaning of frame expressions to be clearly defined. The definitions below are used by Hayes to characterise the KRL system, Bobrow and Winograd(1977).

Frames in KRL are called "units" and have one or more named slots. Each contains a number of "descriptions" which in turn consists of a list of "descriptors". Each descriptor has an associated set of "features". The values of each feature will vary from instance to instance. The set of descriptions in any instance of a slot is taken to

be the slot-filler "value". There are a limited number of descriptor types that attempt to provide a fairly natural way of specifying information. They are not a minimal set in that there is some overlap between the mechanisms. The definitions of these descriptor types are now given along with Hayes' definitions in predicate logic.

In the following, KRL units are enclosed in square brackets, each slot is enclosed in angled brackets, <...>, individual descriptors are enclosed in parenthesis, (...), and braces are used to combine multiple descriptors into a single description. The general format is:

```
[unit_name UNIT unit_category
   <slot_name1 { (descriptor1 ...)
                 (descriptorN ...) } >
   ...
   <slot_nameN {....} >
]
```

1. Direct Pointer Descriptors - these are simply unit names, numbers, strings or pieces of Lisp data used to provide unique identifiers or point directly to data, e.g. "pump101".
2. Perspective Descriptors - the simplest use of perspective descriptors is to assign units to membership in a category (a sort of IS_A link). Perspectives were, in fact, based on a much subtler idea of viewing unit in terms of comparison with some prototype, many such comparisons being allowed for any instance. This is discussed later. The format of a perspective descriptor is:

   ```
   (A prototype with feature1 = filler1 ... featureN = fillerN)
   ```

   For example,

   ```
   (A pump with shutoff_head = 100 flowrate = 5)
   ```

   Perspective descriptors are used to describe the special SELF slot. This is used to describe the unit itself, eg

   ```
   [p101 UNIT Individual
   <SELF {(A pump with flowrate = 5)
          (A safety_hazard with ...) }>
   ...]
   ```

   This sort of information can be expressed by a lambda expression in predicate logic, eg

   $$\lambda x.\text{pump}(x) \wedge \text{flowrate}(x) = 5$$

3. Specification Descriptors - these are used to state the role of a unit in a complex object or event. The format is:

   ```
   (The slot From prototype perspective_to_use).
   ```

   For example

   ```
   (The flowrate From pump (A process_unit with ...)),
   ```

   or the special form

```
(The flowrate From pump ThisOne)
```

when referring to slots in the same unit. This can be expressed by a 1-expression as follows

```
1x. flowrate (P101) = x ∧ pump(P101)
```

where the 1x quantifier is similar to ∃ but indicates the x that exists.

4. Predication Descriptors - these are used to describe relationships in terms of a relation name, arguments and attached procedures. The format is:

```
(Which predicateName. predicateArguments)
```

The arguments can be defined in terms of specifications, eg

```
(Which is_upstream_of (A process_unit))
```

Predication descriptors correspond directly to lambda expressions and relations, eg

```
λx,y. is_upstream_of(x,y) ∧ process_unit(y)
```

5. Logical Boolean Descriptors - used to express a complex logical formula using the connectives OR, XOR, and NOT. The AND connective is implicitly used to combine descriptors. The format is:

```
(OR formula) (XOR formula) (NOT formula)]
```

This, of course, can be directly translated into a non-atomic logical formula in predicate calculus.

6. Restriction Descriptors - A restriction descriptor provides a description that is sufficient to refer to a unique object in a given context. The format is:

```
(TheOne description)
```

For example,

```
(TheOne {(An exchanger)
         (Which functions_as (A feed_pre-heater))})
```

As for specification, this can be expressed by a simple 1-expression -

```
1x.exchanger(x) ∧ ∃y.feed_preheater(y) ∧ functions_as(x,y)
```

7. Selection Descriptors - These are the equivalent of simple CASE or SELECT statements in procedural languages. The format is -

```
(Using Descriptor
       SelectFrom predicate1 ~ selection1
                  predicateN ~ selectionN
       Otherwise   default_selection
```

As for restrictions, this is simply a 1-expression with each case statement being expressed as an OR expression

```
1x.(exp1 ∨ exp2 ∨ ... ∨ expN)
```

8. Contingency Descriptors - these combine descriptors into time or situation dependent descriptions. The format is -

```
(During specification Then contingent_description)
```

For example,

```
(During case_study1 The capital_cost From (A plant with...))
```

This is expressed as 1-expression just as for restriction descriptors except the body must mention a bound state variable to identify the situation, eg

```
1x.∃y plant(y,case_study1) ∧ capital_cost(x,y) ∧ ...]
```

The analysis above has shown that most of the frame "structures" or "expressions" in KRL can be represented, in a superficial form at least, in standard predicate logic terms. The syntax of KRL is, in fact, more expressive than most frame based systems. This indicates that a similar analysis could be made on a number of other systems with more or less the same results. This is an important point as the purpose of Hayes' work was to provide a clearer understanding of the semantics commonly found in frame based systems.

Having studied the content of frame expressions it is now possible to consider the types of inferences that are allowed in frame based systems. This involves considering how frames are to be used. In the case of KRL, the intended use of frames is much more complex than normally found. The designers of KRL were concerned with developing an experimental system that could be used to model complex cognitive processes in a natural language understanding system. As a consequence, few of the mechanisms described below can be found in current commercial systems, e.g. KEE(1985). They are worth discussing however since the ideas involved are of direct relevance to developments later in this thesis.

4.2.4.3 Instantiation, Criteriality and Matching

The most basic type of inference common to frame based systems is that of instantiation. An instance of a generic frame is simply generated by filling in slots with appropriate values. Slot values can be left unspecified and the use of default values may be permitted. In many systems this is the only form of inference available other than the sub-process of inheriting slot values from parent classes in the instantiation process. In terms of predicate calculus this type of inference is simply a unification process involving a collection of properties describing an individual.

A more subtle form of inference is that of "criteriality inference". The concept of criteriality involves the idea that possession of certain attributes is a sufficient and necessary condition for an entity to be regarded as an instance of a given concept. For example, if filler values for all the appropriate slots of a frame can be found then an instance of a concept can be inferred to exist. This sort of inference is useful when trying to classify some abstract entity with given properties, as might arise in

appropriate formulations of process synthesis problems. Rather than enumerate a set of known types, a synthesis search could first establish the desired properties of, say, a separator unit and then identify the resulting type. If no type exists then the properties could be used to investigate whether a new type or class of separator could be invented.

This type of inference can be mapped directly into first order logic. Consider a frame representing concept C with slot relationships R_1 to R_n. The clause:

$$\begin{aligned} & \forall x.C(x) \supset R_1(x, f_1(x)) \\ \wedge\ & \forall x.C(x) \supset R_2(x, f_2(x)) \\ \wedge\ & \ldots \\ \wedge & (\forall x y_i \; R_1(x, y) \wedge R_2(x, y_2) \wedge \ldots R_n(x, y_n)) \supset C(x) \end{aligned}$$

captures this type of inference. The criteriality assumption is given by the last part of the formula. Such formulas make it clear whether criteriality is being assumed or not and whether it has been used in generating further assertions. This is usually not true of frame based systems where such assumptions are implicit and "embedded" in the implementation of the frame interpreter. Implicit assumptions of this sort will cause severe problems for a system trying to maintain consistency of information.

A third form of inference is "matching". The inference involves more than just syntactic unification and relies on the assumptions about the domain which the frames are representing. The example that Hayes gives is of a Man frame with a "pet" slot that is to be matched against a DogOwner frame. A sufficient condition for this match might be that the pet slot is filled with an instance that is known to be canine. KRL provides a mechanism to perform the transfer of the slot_fillers from one frame to another. This simple type of match can be expressed by a sequence of implications. For example, the condition for having a dog might be:

$$is_dog(x) \wedge pet_of(x,y) \supset dog_of\ (x,y).$$

Given the assertions :

$$name(man1, \text{"Alistair"}) \wedge pet(man1,ben) \wedge is_dog(ben)$$

and assuming the criteriality of DogOwner only involves a name slot and a dog slot, it follows directly that

$$dog_of(man1, ben) \text{ and } DogOwner(man1)$$

Many examples of matching involve simple forms of theorem proving of this type. The match algorithm in KRL allows for different proof strategies, see Bobrow and Winograd(1977). The match can be restricted to explicitly defined properties or can use deduced properties as indicated above. There is a strong heuristic aspect of the search strategy involving the use of attached procedures, evaluation of partial matches, best "fits", resource limitations, etc.

Hayes points out that some of these features may not be directly expressible in pure first order logic, creating new relations for instance by abstraction during a match process. Most examples of matching inferences discussed in the literature can, however, be expressed in a standard theorem proving framework.

4.2.4.4 Making Comparisons and Caricatures

The use of multiple perspective descriptors in KRL to describe the SELF slot of a Unit is an example of the ability to view one unit as though it were another that is found in some frame based systems. In KRL a Unit can be specified by a number of comparisons with known prototypes, i.e. by perspective descriptors.

Hayes suggests that there are three basic ways in which these comparisons can be interpreted. The first and simplest view or interpretation is that each comparison is simply asserting more and more restrictive properties of an entity. For example, to say an instance of a heat exchanger is a process unit allows certain properties to be inferred while stating it is also a "plate_heat_exchanger" allows more restrictive properties to be inferred. In logical terms this simply consists of extending a conjunctive expression describing the properties of an entity. This in fact corresponds in many ways to the common notion of multiple inheritance, i.e. inheriting properties from more than one parent class. Multiple inheritance is not provided in CLAP since similar effects can be achieved more rigorously through the use of explicitly defined relations.

The second interpretation of comparison with a frame is that the frame being used for comparison is a "correct way of looking at an entity" with the proviso that neither frame is a further specification of the other. For example a heat exchanger can also be a safety hazard. Each frame has slots not referred to by the other and each can have their own independent criteriality. There is no problem so far as this can simply be expressed as a conjunction of predicates. A problem does arise, however, when an entity seen from a different point of view has apparently contradictory properties. For example an exchanger might be seen as "hazardous" from a mechanical point of view, e.g. pressure considerations, but "non-hazardous" from a chemical point of view, e.g. possible reactions if leakage occurs. The contradiction arises when the property is referred to by the same name and number of arguments in each viewpoint. Hayes suggests three reasons for the contradictions arising:

1. Different properties are really being asserted in each frame, e.g. "mechanical" hazards and "chemical" hazards are just different notions and the difference should be made clear.
2. An extra argument, or arguments, is being implicitly encoded that should be made explicit. A common example is an implicit reference to time or place, e.g. a contradiction from a person being described as unfriendly or friendly if he

really is unfriendly at work, say, but friendly at home.

3. The frames really do contradict one another. In this case a translation into assertions must also explicitly represent the contradiction. Some other inference mechanism must be used to resolve this situation.

The first two points are simple illustrations of the problem of arbitrary tokenism in frames. Properties in different frames can be implicitly taken to imply different relationships, e.g. the use of the token "hazardous" or "pressure" as the name of a property, in just the same way as different uses of tokens in production rules. In the KRL system it is intended that a token representing a property name can be defined in alternative ways in different Units. Unless a translation mechanism was provided between these definitions then any consistency or "matching" inferences would not be reliable, contradictions of the sort described above being inevitable.

The third way to interpret comparisons is to treat them as forms of analogy. Analogical reasoning is important in a number of areas including solving novel problems, by making analogies with solution techniques used in vaguely similar problems, and language understanding, e.g. making sense of phrases like "John is a pig" or "a pig of a problem". In this case describing an object by comparison with a known prototype requires that the similarities and points of difference be noted in a restricted form of matching. The usual intention of a caricature or analogy is that some but not all of the properties be "transferred" to the caricaturee, e.g. the non-linear characteristics of a function or the eating habits of someone compared to a pig. The essential point is that criteriality is not assumed when making an analogy. Only the relevant properties needed to make the analogy need be transferred. The analogy holds, or is at least plausible, when the transferred properties do describe the caricature. Very few, if any, frame based systems could cope with both analogies or caricatures and factual assertions since the assumption of criteriality is not handled in an explicit form. What is required to make a caricature is a non-criterial frame and an ability to map qualities, i.e. relational symbols, between one entity and another. A simplified form of mapping can be expressed quite simply in logical terms. If a defining concept for, say, pig_likeness as opposed to pig_hood, can be given by:

$$\lambda x.\psi(x), \text{ where } \text{Pig_like}(x) \equiv \psi(x)$$

(where ψ will generally be some complicated assertion) and ϕ is some syntactic mapping of the analogy, e.g. sty -> house, then Pig_like(John) is true just when $\phi(\psi)$ holds for John. This is the basis used by Kling(1971) to express analogies as part of a matching inference mechanism for use in algebra problems.

Another well known system that attempted to use analogies was MERLIN, Moore and Newell(1973). MERLIN is flawed, however, in the same respect as KRL as

regards the implicit assumption of criteriality and the inability to distinguish between assertions associated with caricatures and those that are strictly provable.

4.2.4.5 Default Reasoning

Most frame based or object oriented systems allow default values to be defined for slots. The default value for a slot, which may also be inherited, will be used when there is no explicit evidence to the contrary, ie when there is no explicit value provided for the slot. A simple example to consider is the value of a "status" slot in a frame representing a pump, say, in a plant model. The possible values of the slot might be {okay, needs_attention, broken} with a default value of "okay", ie unless it is known otherwise assume that the status of the pump is okay. If it is subsequently observed or concluded that the pump is not okay then the assumption made given the earlier state of knowledge must be recognised as no longer valid in the light of the current state of knowledge. This is not the same as a contradiction since the state of knowledge or belief has altered: new beliefs have been added while previous ones must be removed.

This form of default reasoning can be directly expressed in logic if it is recognised that the default assumptions involve an implicit reference to the whole state of knowledge of a system, eg the global database of a production system, at the time the assumption was made. Any event or exception that arises that changes the truth conditions of default assumptions must be acted upon. This means that the propogation of inferences from assumptions must be explicitly controlled. This requires that some form of reference to the knowledge state which produced the assumption be available, as well as an ability to check the possible inferences that can be deduced from the addition of new assumptions to assertions that already exist in a knowledge state. The latter functionality can be simply provided by use of the connection graph method, see Section 2.3 - Kowalski(1975), ie to check whether the addition of proposition p can affect proposition q.

Since the analysis of Hayes a number of developments have occurred. A number of logics have been developed which are directly concerned with revisable forms of reasoning, of which default reasoning is an important example. The default logic of Reiter(1980) is most commonly cited. A number of techniques have also been developed to control the use of assumptions in making inferences and are usually referred to as Truth Maintenance Systems. The most important examples are Doyle (1979) and de Kleer(1986). Brief details of these techniques are given later.

Despite these developments the basic conclusion of Hayes is still valid. Default expressions require the use of a concept of "observations", ie the addition of new

assertions into a knowledge based on observations. Deriving the consequences of assumptions, ie making inferences, requires that observations be noted and the state of the system's knowledge be made explicit. As Hayes points out, "this requires not a new logic, but an unusual ontology and some new primitive relations". This will involve the system making assumptions about itself, when making inferences. This is exactly the case in default logic where a modal operator M is used to define default assumptions and is understood to mean "it is consistent to believe". The general form of a default rule is:

$$\frac{\alpha : M\ \beta}{\gamma}$$

This reads as - if α is believed and if it is consistent to believe β then γ can also be believed. The previous example of the status of a pump would be expressed as

$$\frac{\mathtt{Pump(x)} : \mathtt{M\ status(x,okay)}}{\mathtt{status(x,okay)}}$$

4.2.4.6 Reflexive Reasoning

The use of default values and making caricatures or comparisons are examples of more sophisticated inference techniques that are important in systems that attempt to investigate or model human reasoning, eg language understanding in KRL. As Hayes suggests, a good deal of common sense reasoning involves the ability to reason about oneself and one's own abilities as well as about the world or domain being described. This is necessary for handling both defaults and caricatures/comparisons, eg referring to states of knowledge of the system. The awareness of, or ability to reason about, one's own internal deduction processes is very important when solving even trivial problems. A familiar example in process engineering is knowing how and when to model processes in a simplified way, eg the use of ideal properties. This model may have been the only option at the time, eg due to lack of data, but could be changed as soon as more data becomes available. The use of an idealised model is a form of caricature/comparison and the availability of data shows the interaction needed between observations and inference processes. Some form of reflexive/revisable reasoning is needed to handle this common situation adequately.

Apart from the use of default logics and truth maintenance systems mentioned above, a number of other developments in the area of logics to express revisable forms of reasoning are worth noting:

1. The use of modal predicate logics to express "uncertainty" or "possibility". For instance, alethic logic introduces the operators "it is possible that" and "it is necessary that", epistemic logic uses the operators "knows" and "believes", etc. Thayse(1988) is a good introduction, Smets et al(1988) is a more sophisticated discussion.

2. McDermott(1980, 1982) describes a non-monotonic logic that allows revisable reasoning by evaluating maximal sets of consistent assertions that can be inferred from a set of premises. A non-monotonic logic is simply one in which the usual monotonic consequence relation, $A \models p$, does not hold, ie that any model of A is also a model of p (see Appendix B). For example, to infer p from a set of premises A and then to withdraw p when some new assertion q is added means that there is a model of $A \cup \{q\}$, and hence a model of A, that does not verify p. Put more simply this just says that if you believe p at some point in model A to be true then you can't rely on it being true when new information q is added. This means that any intermediate reasoning based on p being true is probably now invalid.
3. Moore(1984, 1985, 1988) describes an autoepistemic logic concerned with reasoning about one's own beliefs and knowledge. The logic allows states of belief to be altered. The work is in fact a reconstruction of McDermott's non-monotonic logic.

There is not space to describe these logics in any sort of detail here. The logics are all relatively straightforward extensions of the classical axiomatic systems discussed in Chapter 2. An appreciation of their definition will be useful in fully understanding Chapter 5. The reader is referred to the original references for full details, but see Turner(1984) for a general overview. Where absolutely necessary basic details have been provided, ie Section 4.3.1.6 on modal logics.

The important point to make is that certain forms of reflexive reasoning can be expressed in predicate logic terms for which a precise semantics can be defined. This type of reasoning, when attempted has usually been implemented in an ad-hoc or ill defined way in many frame based systems.

4.2.4.7 Summary

The value of Hayes' analysis was to show how many aspects of frames and slots could be translated into sets of unary and binary predicates in standard first order logic. The meaning of frame expressions can, therefore, be stated in more precise terms. Hayes suggests that the real value of frames has little to do with representation issues but rather with implementation issues, eg assertions can be stored in "nameable bundles", ie frames, in non-clausal form. Frame based systems have also been useful in that they have stimulated research on more sophisticated forms of logic to handle defaults, beliefs, etc.

The author's only objections to Hayes' account is that it fails to take into account the "spirit" in which many of the early frame based systems were developed. KRL, for example, was based on ideas taken from cognitive science research and was seen by Bobrow and Winograd as a toolkit or means of investigating/modelling human reasoning processes. There is no report of KRL ever being applied in this way but if it had it is the author's opinion that the reflexive forms of reasoning needed would be difficult to express in the logics currently available. Another point is that Bobrow and

Winograd make it clear that the intended use of comparisons was precisely to avoid the concept of fixed or static definitions of units. The fact that their implementation turned out to be very much like fixed definitions is another matter. (It shall be argued later, Chapter 4, that natural kind terms (tree, apple, etc.) cannot in fact be defined in analytic terms). What the designers of KRL failed to do was provide an explicit means of stating whether, or when, criteriality assumptions could be used.

Hayes' analysis is certainly true, however, of many simpler frame based or object oriented systems. With this analysis in mind it is now possible to highlight some critical problems that arise with many implementations of frame based systems. These problems will be seen to directly undermine the primary research objective of providing a unified representation with a consistent means of interpretation, as stated in Section 4.1.

4.2.5 Problems Concerning The Use of Frames

The problems considered here are those that arise from default-style frame representations in conjunction with inheritance hierarchies that allow slot values to be arbitrarily overridden. This sort of representation is the rule rather than the exception in frame based systems. The main problem will be seen to arise from the temptation to try and use default based frame notations for all aspects of knowledge representation, eg using frames to represent both arbitrary concepts and their interrelationships. There is strong evidence that this is the case in publications on related research in process design, eg the Design-Kit System (Stephanopoulos et al, 1987), Kramer(1987), Venkatasubramanian and Rich(1988), with little or no indication of an appreciation of the representation problems that exist with frames. The problems are therefore well worth discussing here. Many of the following points stem from Brachman(1985).

It is clear that in many frame based/object oriented systems the use of an IS-A inheritance link corresponds to the implicit, rather than explicit, use of universally quantified "if-then" statements: "if" a child frame has an IS-A link to a parent frame "then" it inherits the parent properties - hence the implicit universal quantification. It is also clear that in most systems frames are intended to have constant definitions , ie a predefined number of slots with names that are interpreted as necessary conditions. This has been noted by a number of authors, Charniak(1981), Hayes(1979), Fikes and Kehler(1985). In the previous discussion, for example, Hayes represents a concept C with slot relationships $R_1, R_2, ..., R_n$ as

$$\forall x\ C(x) \supset R_1(x,f(x_1)) \wedge ... \wedge R_n(x,f(x_n))$$

Once it is known that an entity E is a subclass S of some parent class P then all the P-like properties of E can be found by inheritance. Now most frame systems allow an instance to override or cancel the properties that it would normally inherit. For example, a generic reactor frame might have a default value for a "flow-regime" slot as "well-mixed". A particular instance of a reactor might override this value and set it to "plug-flow".

If a frame system allows properties to be cancelled in this way, as is usual, then the suggestion that there is an implicit universal quantification is not quite right. The correct reading must be that the sense of "every" in $\forall x$ be replaced by "typically" and take this to mean that the properties of P are only default properties of S. Indeed this is exactly how prototypes are intended to be used in KRL, ie they are used to represent stereotypical situations. The common interpretation of "typical" is equivalent to "in the absence of any information to the contrary, assume ...", Reiter(1978), ie only use the default property from P if no local value is provided for S.

The important point is that any system that allows exceptions to be defined by overriding properties that would normally be inherited is using frames in the "typical" rather than "every" sense. If this is the case then a number of, somewhat drastic, consequences follow that cannot be ignored in the design of an environment that attempts to maintain any sort of data/knowledge consistency. These are now described, the discussion assuming that the system allows inheritance of both properties and their values to be cancelled - again this is commonplace.

The first thing to consider is what it is that is being cancelled. Consider a cellular design unit which "typically" has a feed pump, expressed as a property, with a default type of pump to use, eg centrifugal. If a particular instance of a design unit simply changes the type of pump then what is being cancelled is the value. If, however, an exception arises in that it has all the design unit properties except it does not have a feed pump then what is being cancelled is the attribute or property itself. If there is no mechanism to state explicitly which properties are necessary, ie essential, for the IS-A link to be made then the somewhat absurd situation can arise in which most or all of the properties are cancelled, eg a pump is a heat exchanger except that it has no baffles, no tube plate, no tubes, etc. This might be done just to inherit a heat loss equation if this was defined as part of the heat exchanger frame. The heat exchanger frame no longer really represents a heat exchanger and inheritance is being used merely for programming convenience. Checking the consistency of new assertions becomes extremely problematical in such a system where the interpretation of IS-A

links becomes unclear.
The situation is further confused where the cancellation of a link is intended to mean something, eg the feed pump is removed for repair.

A more important consequence of interpreting frames in this way is that the default-exception approach cannot be used, without significant alteration, to make unequivocal universally quantified statements since any other frame linked by an IS-A link could be an exception that cancelled the relationship. This is extremely important as it applies to both contingent universal truths, ie those that are satisfiable but not always valid (see Appendix B) and necessary truths, ie exceptionless universal statements, such as those found in algebra systems.

To illustrate the point Brachman considers the simple example of observing that all the vehicles in a car park are in fact cars, ie no bikes, lorries, etc (replace car park with cellular design unit and car type vehicles with plate heat exchangers for a more relevant example). It would be convenient to represent this contingent universal fact with a single statement. Predicate calculus allows a non-committal assertion, ie regarding the number, make, etc of the cars, to be made very simply, eg

$$\forall x \; car_park(p) \wedge vehicle(x,p) \wedge car(x)$$

Since nothing is guaranteed in the frame system the only possible way to check statements of this form are true, eg in reply to the question "are all the vehicles in the car park cars?", is to examine all the instances of a frame to see if the description holds. Even then, the information in the system will need to be complete and operate under the "closed world assumption". This assumption assumes perfect knowledge about the domain being modelled in that if it is not known consider it false, see Reiter(1978). This could be very expensive to perform and the process would have to be repeated every time the assertion was queried or used in an inference.

In the case of necessary truths, ie definitional conditions that should always be true, this situation introduces another problem. There are many situations in which properties do indeed hold for all instances, eg all triangles have three sides, all batchelors are not married, all heat exchangers have at least one hot stream and one cold stream. These are necessary conditions. It should now be clear the they cannot be directly represented by frame slots if cancelling is allowed. Without this definitional capability Brachman illustrates at some length, that even the simplest composite descriptions cannot be formed in a reliable manner, eg the definition of a "one pass heat exchanger". Both necessity and sufficiency are explicitly required to make composite descriptions of this sort. If exceptions are always allowed then an

instance of a "one pass heat exchanger" might turn out to have more than one pass. Any usable inference mechanism must be able to deduce without fail that any "one pass heat exchanger" has only a one pass fluid flow arrangement.

An ability to express and reason with necessary conditions and facts will be undoubtedly necessary in a representation system concerned with knowledge integrity.

(An interesting phenomenon that occurs is that in the desire to try and endow frames with some compositional meaning, complex hyphenated names are used to label the frames in a form of wishful thinking, eg one-pass-heat-exchanger - see McDermott(1981) in "Artificial Intelligence meets Natural Stupidity" for further discussion).

There are a number of other problems concerning the use of frames. Many could be avoided by careful reimplementation but few systems properly address the following problems:

1. There is often an ill-thought out approach towards the interpretation of frames in different ways at different times. The principal ambiguity results from a lack of differentiation between definitional, eg generic, and factual, ie observation based, interpretations and the way these are used in inference procedures. That is, how does one discriminate or combine these two types of information and how is this distinction maintained for use in later inferences? For example, a property P may be deduced in one frame based on a typical definition in a frame F and some observed facts O. If any of the observed facts O are modified or the default properties in F are overridden, how is the status of P to be handled?
   This is another example of the need to be able to explicitly reason about states of knowledge and unravel the consequences of assertions based on assumptions in a rigorous or complete way.
2. The "meaning" of most frame representation languages is specified only in terms of the data structures, typically inheritance networks, used to implement it. There are many ways in which the links between the structures could be interpreted, see Woods(1975) and Brachman et al(1983). If the system designers do not provide strict guidelines on how to interpret frames, which is always the case, then the user is free to interpret frame structures in whichever way he/she wishes, as in point 1. These interpretations may well be inconsistent. This is particularly important when frames are taken to be assertions about the way things are in the world. If an explicit mechanism is not provided to clearly distinguish between the use of essential or criterial properties and accidental or non-criterial properties, as discussed earlier, then wholly unwarranted inferences can be made.
3. The ability to override inherited properties in an unrestrained fashion means that composite concepts cannot be constructed out of simpler conceptual units in a reliable way. This follows directly from the discussion above on the inability to state definitional conditions in an unequivocal way. This is an important point as it is the basis on which process units are defined in the DESIGN-KIT environment, Stephanopoulos et al(1987), through multiple inheritance. Stephanopoulos describes a "desired-reactor" object which illustrates the confusion indicated in point 2 above. The reactor has an IS-A link to "reactors"

which in turn is IS-A linked to "generic-process-unit". This seems reasonable enough except for the use of the plural form to name a typical reactor. However, IS-A links are then interpreted in a different way to pull in necessary characteristics needed to describe the "desired-reactor", eg IS-A links to "adiabatic", "perfect-mixing", "single-step" frames. Two things are clear: frames such as "adiabatic", which is IS-A linked to "thermal-characteristics", certainly do not represent typical entities. Rather, they define single or sets of necessary relations. Secondly, multiple inheritance is being used to try and form a composite object by an implicit or intended use of an IS-A link to mean conjunction. The programmers have, therefore, implicitly interpreted the IS-A link in different ways, ie class membership and some sort of conjunction, which is not supported by the underlying system, KEE(1985). For example, if the "adiabatic" properties are cancelled or overridden then what is the status of the IS-A link to the "adiabatic" frame supposed to be? Are inferences concerning the "adiabaticness" of the reactor still valid because the IS-A link still remains even though the property value might indicate the reactor is non-adiabatic.
Most of the other drawbacks identified above with frames can be applied to the representation style adopted in this description of DESIGN-KIT. It is important to note that this is just not logical nit-picking. They are fundamental problems that undermine the theoretical status, whatever that may be, and practical usefulness of such systems.

4. Simple expressions of doubt or incomplete knowledge are often difficult or impossible to express during the instantiation process, ie slot filling, since statements in predicate logic typically cannot be handled directly in assertional frame based systems. For example one cannot say that the faulty pump is either p101 or p102. Typically this would have to be represented as a Lisp list of alternative values and some inference technique provided by the programmer would be needed to handle this disjunction. The underlying system usually has no way of knowing that the slot value represents a disjunction and, hence, can make no explicit use of this information. It is extremely unlikely that user supplied inference techniques will establish or maintain the consistency of the entire knowledge base. The more likely situation is that small pieces of code will be written to only check for local consistency for the frame in question.

A number of other criticisms can be found in Jackson(1986, 1987).

The problem of arbitrary tokenism in the context of frames, as regards the use of arbitrary slot and class names, has already been mentioned. In the light of the criticisms above concerning the inconsistent interpretation of frame structures, this problem could become even more acute in frame based systems. This is especially true in hybrid toolkits where the onus is on the programmer to build or construct his own inference algorithms from the basic tools provided.

4.2.6 Improved Implementations of Frame Based Systems

One of the basic causes of the numerous problems that arise from using frames is the lack of a clear definition or understanding of what frames in a particular system are supposed to represent and how they are to be interpreted. This is mainly due to the different sources of ideas that originally inspired their use, eg experimental studies in cognitive science (Rosch 1975), use of "semantic networks" in language understanding (Quillian 1966) etc. As it turns out, the notion of a "concept" in human

reasoning raises extremely deep philosophical and linguistic issues. It is no surprise that the use of frames became "confused" as they appealed in different intuitive ways to different programmers. The whole issue of what a "concept" might be is presented in Chapter 5.

Before leaving the topic of frame based systems a number of specific systems and studies will be mentioned that directly address some of the problems described above. Full details are not given here, the reader being referred to the original papers, since development of frame based systems in a currently recognisable form is not pursued further in the remainder of the thesis. The reasons why are explained in Chapter 5.

4.2.6.1 CLAP - Conditional Definitions and Contingent Universals

A partial solution to the provision of both conditional definitions and contingent universal facts can be very easily implemented in CLAP. First consider the problem of defining property values that should not be cancelled in classes linked by IS-A links. Since the instantiation of objects in CLAP is achieved with Prolog's unification algorithm, the trivial solution to providing a property value that cannot be overridden is simply to define a ground term, ie variable free, as the slot value in the definition of the generic object. For example, to state that a triangle has three sides simply use -

```
object (triangle):-
  slots - [number of sides - 3,
           ...],
```

in the generic definition. This is referred to as a static slot in CLAP. There is no means of overwriting static slot values in CLAP.

In order to define more complicated necessary relations it is better to use "static relations". For example, the generic relation hierarchy can be used to define the general form of the relationship, either symbolic or a mathematical constraint. An instance of the generic relation can be referred to by "name" in the generic object definition using a static relation, eg

```
object (whatever):-
  slots - [...],
  relations - [has constraint_x - x_001,
               ...],
```

where x is the constraint and x_001 is the name of the specific form being used. As with static slots, there is no means to overwrite an inherited static relation.

As regards contingent universal facts there is no problem. CLAP has its own way of asserting facts, in predicate logic form, or the user can make use of the underlying Prolog interpreter. Once a generic type has been declared with static slots then assertions involving the $\forall$ quantifier can safely make use of the name of the generic

type. There is no need to gather up all the current instances and check out particular slot values as would have to happen in many frame based systems. try and use multiple inheritance to express sets of relational constraints, as was the case in the reactor example earlier. Inheritance in CLAP is only intended to support specialisation of classes through refinement of properties via implicit conjunctions down the hierarchy.

4.2.6.2 KRYPTON

KRYPTON, Brachman, Fikes and Levesque(1983), was an important attempt that tried to overcome the problems resulting from users having free access to the internal implementation data structures of frame based systems and then using them in ad hoc ways to compensate for the assertional limitations of the system. Krypton is a rare example of a system that attempts to allow the definition of complex terms as well as provide a powerful assertional capability. It is also interesting in that the service provided by the knowledge representation system is defined functionally in terms of what it can be asked or told about the domain.

Krypton uses two internal representation languages: a strict frame based language to form descriptive terms, the TBox language, and a first order logic based language to make assertions, the ABox language. These shall now be treated briefly in turn.

The TBox language allows taxonomies of structural terms to be established and questions about analytical relationships, ie necessary constraints, between these terms to be answered. Expressions in the TBox language are strictly used as structured descriptions and have no direct assertional import. That is, unlike the systems discussed so far no deductions can be directly drawn from the presence or absence of a Concept/Frame/Link etc in an inheritance network. Instead, some other mechanism is needed to use a structured description to state facts. This corresponds to the "descriptional" view of frames, as opposed to the assertional view, referred to in Section 4.2.4. The TBox language is in fact based closely on the KL-ONE system, Brachman(1979) which interprets frames in a strictly structural way. (KL-ONE is discussed elsewhere, section 4.2.6, but not in the context of the current discussion).

In particular TBox supports Concept expressions, which roughly correspond to frames, and Role expressions, which correspond to slots (they correspond to KL-ONE Concepts and Roles). Structural descriptions are formed by combining or restricting other Concepts and Roles using a number of predefined operators. There are nine such operators, five for Concept expressions and four for Role expressions, which are carefully defined in a compositional framework.
For example the "ConjGeneric" operator in a concept expression simply forms a

concept corresponding to the conjunction of the other concepts used as arguments, C_1 to C_n

$$(\text{ConjGeneric } C_1 \ldots C_n).$$

The VRGeneric concept expression operator, i.e. Value-Restricted Generic, written as

$$(\text{VRGeneric } C_1 \text{ r } C_2),$$

yields a concept meaning "a C_1 any r of which is a C_2" where C_1 and C_2 are concepts and r is a Role. This corresponds to restricting the type of a Role filler, i.e. the "data type" of a slot value in simple terms. A more interesting concept operator expression is the PrimGeneric operator. In many domains it is useful to be able to define necessary but not sufficient conditions for a concept. The PrimGeneric operator provides this functionality and forms primitive specialisations of a concept. A primitive Concept is one that is subsumed by its parent Concept but where no sufficient conditions exist to determine if something is described by it. (NB a type A subsumes a type B if the meaning postulates for A and B logically imply

$$\forall x[B(x) \supset A(x)]$$

A typical example of a Role expression operator is the VRDiffRole, Value Restricted Differentiation, that takes a Role r, a Concept c and derives a derivative role meaning "an r that is a c", eg. (VRDiffRole brother person). For more details and examples on these operators see Brachman, Fikes and Levesque(1983).

The ABox language, used to make assertions, also uses sentence forming operators to construct sentences in a compositional way. In this case the operators are the usual sentential connectives and quantifiers from first order logic, eg not, or, there-exists, etc. These are necessary as the TBox is "purged of any assertional ability". In a standard first order logic language the non-logical symbols, ie the predicate and function symbols, are taken to be "independent, primitive, domain-dependent terms". In KRYPTON, the TBox provides a way of specifying domain dependent terms. The main difference between ABox language and a standard first order language one is that the non-logical symbols are taken to be terms of the TBox language. The resulting predicates can therefore be definitionally related to each other, ie the reasoner in the ABox has full access to the definitions in the TBox. ABox predicates are not simply unconnected primitives as they are in predicate calculus. This division of representation into a terminological one and an assertional one is, of course, a more rigorous version of the idea of a module definition language in Designer's Assistant or the use of meta-slot descriptions in CLAP. Neither of the latter attempts used a well defined set of sentence operators but relied on a mixture of logical

constraints and evaluable procedures. It is likely that in a sophisticated system a mixture of the two approaches would be needed. This is because the designer's of KRYPTON say little about how the inference techniques in the ABox are to be implemented. The original work investigated the use of a well known non-clausal theorem prover due to Stickel(1982). This could not be directly used in a large scale for efficiency reasons, see Section 2.3.

Finally, KRYPTON does not allow the user to have access to either the definitional network in the TBox or the collection of sentences in the ABox. What is accessible is a fixed set of functionally defined operations over the TBox and ABox languages. The service provided by KRYPTON as a knowledge representation system is completely specified by these operations. This is in keeping with good programming practice of defining interaction with an abstract data type solely by a set of well defined operations, or message protocol in O.O. terminology, without any reference to internal implementation details. What the system does is therefore specified independently from how it is achieved.

The functional definition approach is important as it isolates the programmer/ user from implementation details. Incidental properties of an implementation structure cannot be used for ad hoc interpretations or for alternative methods of knowledge representation.

The importance of KRYPTON was to show that a well constructed system can, in theory if not in practice, get round many of the problems associated with frame based systems. The question of whether the reasoner or theorem prover in the ABox can efficiently perform the necessary inferences is an entirely separate issue. There is no indication in the literature that this can be currently achieved. A severe limitation of KRYPTON is that it makes no attempt to reason with default or revisable forms of reasoning although these features could be implemented on top of the TBox/ABox framework.

4.2.7 A Mathematical Analysis of Inheritance

In order to understand more fully the properties of inheritance graphs that do handle defaults and exceptions in an implicit way, Touretzky(1986) provides an invaluable analysis of the problem. His thesis presents a formal mathematical theory of inheritance with exceptions and shows how it can be applied to the NETL system, Fahlman(1979). (Fahlman's NETL machine was an outline design for a massively parallel computer architecture that represented and reasoned about hierarchical frame like networks). His analysis is important as it outlines the relationship between the

mathematical theory developed and non-monotonic and default logics. It also highlights the semantic problems that result from the use of multiple inheritance. The thesis raises some important points that are worth summarising here.

Much of what frame systems try to represent is of the form of normative statements, ie statements that are usually true or that can be assumed to be true in the absence of contrary information. Inheritance reasoning, however, is not necessarily normative reasoning. Wholly non-normative statements can be made, eg most cars have five wheels, as long as exceptions are allowed where necessary (a somewhat abstruse reversal of Brachman's(1985) objections that were discussed earlier). At best, inheritance reasoning can only crudely approximate reasoning. Touretzky gives the following example:

```
Mammals have four legs
The typical mammal has four legs
Normal mammals have four legs
Nearly all mammals have four legs
Mammals may safely be assumed to have four legs
The default number of legs for a mammal is four
```

These statement all have subtle differences in their meaning and use. A mammal may be "normal" but not very "typical". These words are very difficult to define but seem to allow more assumptions to be made than the more restricted statement concerning the default number of legs. All such subtleties are lost in a typical inheritance system that simply has the value four in the legs slot of a mammal frame.

The analysis of Hayes, Section 4.2.4, showed that mapping frames and slots into predicate logic is relatively straightforward. What is not handled easily is the representation of default information and reasoning with exceptions. This is precisely why non-monotonic logic, McDermott and Doyle(1980), and default, Reiter(1980) were developed.

McDermott and Doyle use a modal operator M to mean "is logically consistent" in their non-monotonic logic (NML). Consider the standard example of reasoning about birds which can usually fly. A non-monotonic inference rule might be:

$$(\forall x)\ \text{Bird}(x) + \text{M}[\text{Can Fly}(x)] \rightarrow \text{Can Fly}(x)$$

Now consider an exception to this rule, eg ostriches which cannot fly. This might be written as:

$$(\forall x)\ \text{Ostrich}(x) \rightarrow \text{Bird}(x) \wedge \neg\text{CanFly}(x)$$

Suppose, however, there is an unusual ostrich, O1, that can fly - which any human reasoner would be willing to accept. Since the original exception was only expressed

as an ordinary implication, rather than a non-monotonic one, ostrich O1 cannot be accounted for by Ostrich(O1) ∧ CanFly(O1) as this is inconsistent. A different set of axioms might be:

```
(∀x) Bird(x) ∧ M[CanFly(x)] → CanFly(x)
(∀x) Ostrich(x) → Bird(x)
(∀x) Ostrich(x) + M[¬CanFly(x)] → ¬CanFly(x)
```

These axioms are okay for the ostrich O1 that is known to fly but they do not work for an ostrich for which it is unknown as to whether it can fly or not. To see why this is so, consider an ostrich O2 that cannot fly. The first two axioms generate an expansion in which O2 can fly because he is a bird whereas the last inference rule generates an expansion in which O2 cannot fly because it is an ostrich. Both expansions are consistent in NML. There is nothing to state that the more specific expansion should be preferred over the other one. The original point, however, was precisely to be able to assume that an ostrich could not fly unless there was evidence to the contrary.

In order for NML to handle default rules properly each exception must be explicity accounted for in the inference rules they affect. Touretzky provides the following axioms which work for both O1 and O2:

```
(∀x) Bird(x) + M[¬Ostrich(x) ∧ CanFly(x)] → CanFly(x)
(∀x) Ostrich(x) → Bird(x)
(∀x) Ostrich(x) + M[¬CanFly(x)] → ¬CanFly(x)
```

(NB If the relevance of the example is not apparent simply substitute Reactor for Bird, Well Mixed for Can Fly and PFR for Ostrich, ie plug flow reactor).
The first rule rules out the expansion for O2 being able to fly based on O2 being a bird. Stating that O1 can fly can now be asserted as was first attempted. The problem is that if another type of non-flying bird were added, eg penguins, the first rule would have to be amended and rules two and three duplicated for penguins, eg:

```
(∀x)Birds(x) ∧ M[¬Penguin(x) ∧
   ¬Ostrich(x) ∧ CanFly(x)] → CanFly(x)
```

Rather than modify the rule on every occasion and list all the exceptions individually, Doyle suggested a mechanism for uniform defeasibility of rules. This simply involves naming the rule, eg BCF for birds can fly, and only applying it to an individual x if it has not already been defeated, ie been rendered void, for x. For example,

```
BCF: (∀x) Bird(x) ∧ M[¬DefeatedBCF(x) ∧ CanFly(x)] → CanFly(x)
(∀x) Ostrich(x) → Bird(x) ∧ DefeatedBCF (x)
(∀x) Ostrich(x) ∧ M[¬ Can Fly (x)] → ¬CanFly(x)
```

Even using defeasibility of named rules each exception must still be explicity handled by referring to the names of the rules to defeat. This is one of the key differences

between NML and inheritance systems. In his thesis Touretzky defends the usefulness of inheritance systems precisely because they assume a hierarchical structure which allows exceptions at different levels to implicitly override one another. There are a number of restrictions with both approaches as discussed presently.

Default logic is another formalism developed to handle default or normative statements and was introduced briefly in Section 4.2.4.5 earlier, but see Reiter (1978, 1980), Etherington and Reiter(1983). Default logic is in many ways similar to NML, see Reiter(1980) for some comparisons. Defaults are written as inference rules in the general form:

$$\frac{\alpha(x) \ : \ \beta(x)}{\gamma(x)}$$

A useful special case of this form is where $\gamma(x)$ is the same as $\beta(x)$, e.g.

$$\frac{\mathrm{Bird}(x) \ : \ \mathrm{CanFly}(x)}{\mathrm{CanFly}(x)}$$

This sort of rule is known as a "normal" default. It says that if the "prerequisite" $\alpha x)$ is known and the "justification" $\beta(x)$ is consistent with the current state of knowledge then the "consequent" $\gamma(x)$ may be assumed (x simply denotes a vector of universally quantified free variables). The earlier example of flying ostriches, ie exceptions to defaults, illustrated that default rules can interact generating undesirable extensions. In order to try and solve this problem Reiter and Criscuolo(1981) defined a semi-normal default rule of general form:

$$\frac{\alpha(x) \ : \ \beta(x) \ \wedge \ \gamma(x)}{\beta(x)}$$

$$\text{e.g } \frac{\mathrm{Bird}(x) \ : \ \mathrm{CanFly}(x) \ \wedge \ \neg\mathrm{Ostrich}(x) \ \wedge \ \neg\mathrm{Penguin}(x)}{\mathrm{CanFly}(x)}$$

The seminormal form is necessary to prevent unwarranted inferences based on transitivity, eg using the transitivity of Ostrich(x) $\rightarrow$ Bird(x) to generate unwanted extensions.

Etherington and Reiter(1983) show that inheritance with exceptions can be expressed in default logic using seminormal defaults. As with NML, therefore, the exceptions to each rule must be listed explicitly. The implicit ordering of inheritance networks must also be made explicit although Etherington's(1983) ordering is a dependency ordering rather than a subclass ordering. Touretzky again argues that the hierarchical ordering of inheritance systems can implicitly control inferences and exceptions so rules need never be listed explicitly.

The details of Touretzky's formalism and theorems proved are not presented here, see Touretzky(1986) for full details. Two important results arise from his work that are worth noting.
The first is that "inferential distance ordering" is required to handle exceptions properly. Touretzky defines this ordering as follows: "the inferential distance rule says that A may view B as a subclass of C if and only if A has an inference path via B to C and not vice versa". Consider deciding whether A has a property P. If A inherits P from B but ¬P from C then what conclusion can be made? Inferential distance ordering requires that if A has an inference path via B to C and not vice versa then conclude A inherits P: if the path is via C to B and not vice versa then A inherits ¬P; otherwise there is an ambiguity.

The second result is that multiple inheritance algorithms are unreliable if redundant IS-A links are used in conjunction with a path length ordering, the default type of ordering, instead of an inferential distance ordering. Inheritance based on shortest path length measures will also fail to detect ambiguous assertions. The only other way to avoid these problems is to use a restricted form of network such as an orthogonal class/property system, ie slots with unique names partitioned into disjoint sets. This would cause many problems for the "desired-reactor" example earlier, Section 4.2.5, which uses multiple inheritance to refer to constraints all of which might refer to similarly named slots, eg "temperature", possibly with their own default values. This confused situation could not arise in CLAP, however, and this is precisely why multiple inheritance is not provided.

The other important outcome of Touretzky's work is of course the mathematical formalism which can in theory be applied to any sort of frame based system to help characterize it. It is important to note that no attempt is made to provide a theory of normative reasoning. The formal analysis simply defines the properties in inheritance systems in a precise way.

The final part of this discussion summarises Touretzky's comparison of inheritance systems, non-monotonic logic and default logic.

One of the most important points to note is that the expressiveness of an inheritance language is highly restricted. Assertions contain no explicit conjunctions or disjunctions and in most systems negation cannot be expressed. It is this limited language of discourse that makes a hierarchical ordering of knowledge feasible. Inference in such a restricted language is a trivial operation and consists of little more than graph traversal.

On the other hand, first order logic, default logic and non-monotonic logic permit expressions of arbitrary complexity and incompleteness. Inference is not a trivial problem in first order logic as shown in the discussion on resolution based theorem provers in Chapter 2. Inference in non-monotonic logics is much more difficult and costly.

Consider non-monotonic logic for the moment. The problem of selecting appropriate inference rules to apply is even more pressing than in first order logic since the system must now also reason with multiple expansions. There is no generic way of deciding what constitutes a good reasoning heuristic. NML makes no assumptions regarding the way knowledge is structured and indeed, there is no intuitive way to order expressions in a similar way to ordering classes in a hierarchy. On the other hand, because NML does not make any assumptions about ordering it can handle types of non-monotonic reasoning that cannot be handled by the style of non-monotonic reasoning in inheritance systems. The full generality of non-monotonic logics means that they are undecidable. This is not to say that some restricted forms of non-monotonic logic cannot have efficient inference algorithms.
(Decidability is a technical term in recursion theory used to describe what can be decided, ie solved or computed, by a computer. In simple terms, a "problem" consists of a countably infinite set of instances and a multiple choice question about these instances, eg the question "is this formula valid, contingent or inconsistent?" given a set of propositions in an axiomatic system. An "algorithm" always answers a question with the right answer within a finite delay. A theory is decidable if it admits a decision algorithm. A semi- algorithm, also called a procedure, will provide the right answer within a finite delay when the answer is positive but may fail to produce an answer at all when the right answer is negative. A theory is semi-decidable or partially decidable if it admits a decision procedure. Undecidability can only be proved by showing that no program is a decision procedure for the theory in question. Propositional calculus is a decidable theory. First order predicate calculus is only semi-decidable because it involves quantification. See Manna(1974) for more details)

Default logic is somewhat more tractable. Etherington(1987) showed how some aspects of exception handling in inheritance systems can be expressed in default logic. The issue of hierarchical structuring is avoided, as in NML, by requiring the exceptions to be made explicit rather than implicit, as argued by Touretzky. Touretzky shows that the inferential distance ordering is needed to filter out incorrect orderings of default inheritance in an implicit inheritance network.

Reiter(1980) showed that normal default theories have at least one extension and also

exhibit a semi-monotonicity property. If the set of defaults of a normal theory is increased then the new normal theory has at least an extension which contains the extension of the original theory. An important practical consequence of this is that such theories admit a proof procedure. Unfortunately, semi-normal default theories are not guaranteed to have extensions and do not have a semi-monotonicity property.

Finally, Touretzky suggests two practical uses of his inferential distance ordering rule:

1. The ordering can be incorporated into a well defined system, such as the KRYPTON system described earlier, as just one component of the inference mechanism. This would allow a default or non-monotonic logic system to handle exceptions implicitly.
2. Use it as part of a preprocessor to transform sets of normal default rules into semi-normal ones in accordance with inferential distance ordering.

The author is not aware of a system in which this has been done.

4.2.7.1 The OPUS System

A system that is directly relevant to the work of Touretzky(1986) and Etherington(1987) and the rest of this discussion is OPUS , Nado and Fikes (1987). OPUS provides a restricted but properly defined semantics for handling default values and exceptions to default values in a frame based inheritance system. Their language allows inheritance of default values and exceptions that is not covered by Etherington's formulation and allows quantified statements regarding exceptions to be made, eg:

$$\forall v,oc[SubclassOf(C, oc) + ProExcS(C,v,oc)]$$

This says that default values v for slot S from any parent class oc are not to be inherited since class C provides an exception value for this slot.

Nado and Fikes shown how the default and exception statements can be used by a truth maintenance system to maintain justifications for inherited default values or local exceptions. For example, if the value in a slot was an inherited default value and the default value was subsequently removed or deleted in the originating class then the inherited default value would also be deleted. The reader is referred to the original paper for full details.

It is interesting to note that the implementation of default values in CLAP corresponds closely to the effects produced in the OPUS system, eg how defaults are inherited, overridden, etc. Of course, there is no truth maintenance system in CLAP to provide justifications of changes in slot values but as stated earlier, it was not felt that this facility would not have significantly altered the nature of CLAP.

4.2.7.2 Summary

The whole of this section has been concerned with the representational adequacy of frames and the problems that arise from:

1. confused ideas of what a frame actually represents
2. unclear semantics regarding the interpretation of frames
3. naive or simplistic implementations that allow unwarranted inferences to be made, either regarding the internal structure of frames or the links in an inheritance network
4. use of multiple inheritance in ambiguous ways
5. ad hoc treatment of normative statements involving default values and exceptions
6. inability to express universally quantified statements or construct composite concepts in a reliable way

The latter half of the discussion was concerned mainly with more recent systems that are based on some sort of logical analysis of the problems involved. In general, this has resulted in much more coherent descriptions of frame based systems.

It is by no means the case, however, that current systems can be considered a solution concerning the basic representation requirements stated in Section 4.1. Simple inheritance languages are extremely restricted in their expressive adequacy and could not be directly applied to these complex problems. For example, there is nothing in the frame based approach that can be used to directly tackle the problem of arbitrary tokenism in an explicit way. Systems such as KRYPTON that do have access to definitional terms give no real indication of how to use these definitions in practical inference procedures.

The more sophisticated logics needed to handle reliable forms of reasoning have been introduced in an extremely oversimplified way. In reality there are many problems yet to be resolved before they can be of direct use in the design of knowledge based systems, whether in a frame based environment or a more classical theorem proving based system.

A final summing up on the use of frames is given in Section 4.4.

4.3 Problems of Dealing with Uncertainty

The emphasis of the previous section was on the inadequacies of both rule and frame based systems from a definitional point of view, ie their expressive adequacy for defining ideas. A crucial part of any knowledge representation system involved in problem solving activities, of which process engineering design is an extremely complex example, is the representation of uncertainty. Any reliable system, especially one concerned with integrity of knowledge, must be capable of reasoning

with uncertain statements in a coherent manner.

There are many forms of uncertainty that must be handled in process design. For example, it is common to treat data from different sources in different ways. This may depend on the degree of belief or disbelief in the reported accuracy of the data. Certainty also depends on the way in which the importance of the problem situation is viewed as regards the use of evidence, eg using guesses for overall heat transfer coefficients in preliminary heat exchanger calculations. It is also usual to judge the "ability" of knowing some piece of information and deciding whether it is worth the likely effort needed to obtain it or to proceed with uncertain information, eg in collecting data from pilot plant studies for use in front end process design.

In many ways the more important problem is the effect uncertainty has on the reasoning or inference mechanisms as opposed to its symbolic representation. For instance, the effect of uncertainty can play a major role in the order in which goals are selected to be solved in a problem solving process and the efficiency with which a solution can be obtained. Consider an idealised problem involving some top level disjunctive goal. Since only one disjunct need be solved it is obviously worth attempting the one for which the best prior information or evidence is available. Conversely, if the goal is conjunctive in nature then it is sensible to attempt the subgoal which is most likely to fail first since no further effort will be wasted on the remaining goals.

It is also important to recognise the difference in uncertainty arising from provable or unprovable statements, eg certainty in the statements "propane can be condensed with cooling water at atmospheric pressure" and "the price of crude oil will remain fixed over the next year". The important point from a problem solving point of view is expressed neither by a strength of belief or disbelief in either statement nor whether evidence is immediately available to assess each statement. Rather, what is important is the belief that either statement could be proved if need be. In situations regarded as important or critical the use of unprovable statements will generally be avoided as they reduce the credibility of any arguments they support.

It is hopefully obvious, therefore, from this informal introduction that various forms of uncertainty need to be represented in a representation language and that they will play an important part in many reasoning processes. This is especially true in process design where the engineer is constantly faced with making decisions based on uncertain or inadequate information.

Consider again the type of normative statement that lies at the heart of many

problems in knowledge representation, eg "elephants are typically grey". If "all" or "some" had been used instead of "typically" there would be little problem since there are well established truth criteria for these words which can be easily checked. There does not appear, however, to be any equivalent truth criteria for normative statements. Yet it is precisely this sort of statement that seems to constitute much of our background knowledge, common sense, or even specialist knowledge in a particular domain, eg reactor design. Consider, for example, two candidate interpretations of "typically" suggested by Touretzky(1986):

1. at least x percent of elephants are grey
2. a randomly chosen elephant will appear grey with probability at least p

Both definitions have serious drawbacks. Neither can directly explain, for example, the inference "elephants are typically drab" from the statements "elephants are typically grey" and "grey things are typically drab". This is because the conclusion is not strictly deductive, ie does not necessarily follow the premises, but it nevertheless might be a useful inference to make in the absence of contrary information. Neither do probability based interpretations allow reasonable assumptions to be made about individuals, eg if 99.9 percent of elephants are grey, or that a randomly chosen elephant is grey with probability 0.999, then inferring "x is grey" from "x is an elephant" is still not justified. Non-monotonic and default logics, of course, permit such inferences without any sort of probability based justification.

People often accept the truth of normative statements without hesitation for problem solving purposes. This might imply that all that is being suggested is that the statement is a good assumption to make unless you have some evidence to the contrary. It is seldom the case that people make such statements based on valid or explicit statistics.

In Chapter 3 a number of systems were mentioned that made use of numerical measures of uncertainty, in particular MYCIN, PROSPECTOR and the heuristic graph searching algorithms. In the light of the understanding gained in this research it is my opinion that numerical measures of uncertainty are of limited value for many aspects of reasoning in representation systems. The discussion below briefly discusses the reasons why.
NB This is not to say that probability measures do not have a role to play. Used in appropriate ways in specific situations, eg identifying failure rates of plant equipment items, they can provide important information for future design decisions.

An excellent introduction to the problems of reasoning with uncertainty is Cohen (1985). Cohen uses the term normative to mean theories of reasoning that are

consistent with the axioms of probability. The term "prescriptive" shall be used here instead as it indicates more clearly what "should" be done on probabilistic grounds. It is clear then that much of human behaviour and reasoning is non-prescriptive in nature, as witnessed, if nothing else, by the amount of literature on the subject in cognitive psychology. If this is the case, which I am certainly willing to believe since I can remember no instance in my life where I have made a decision solely based on a known probability or statistic, and the intended aim of the so-called expert systems is to emulate expert reasoning then it is surprising that virtually every heuristic system uses approximate prescriptive-like measures of uncertainty. As it turns out this is mainly due to the paucity of usable alternatives in the environments provided by frame and rule based systems rather than any well motivated reasons for their use. Three types of prescriptive approach are briefly outlined below, namely Bayesian-like methods, Dempster-Shafer theory and fuzzy logics. These methods are still widely used despite well known inadequacies. Section 4.3.1.5 introduces an alternative endorsement based approach and its relation to truth maintenance systems. Finally, Section 4.3.1.6 indicates the forms of logic that can be used to express uncertain or incomplete knowledge.

4.3.1 Bayesian-like Approaches and Other Numerical Measures of Uncertainty

The treatment of uncertainty factors in the MYCIN, Shortliffe(1976), and PROSPECTOR systems, Duda, Gaschnig and Hart(1979), was discussed in Chapter 3, see also Appendix C. It was seen that both systems rely on alternative approximations to Bayes' probability theory to reason about uncertainty. This is a typical approach in many systems. There are serious problems, however, with these approaches that are worth pointing out.

As Cohen(1985) points out, most AI systems adopt a parallel certainty inference approach that is commonly thought, but incorrectly, of being synonymous with reasoning under uncertainty. In general, this approach involves splitting reasoning in uncertain domains into two more or less dependent processes. In the first the conclusions are derived as if under certainty while in the second the problem is to decide how much the conclusions are to be believed. This decomposition works well if the two processes are largely independent of one another.

The overall goal then of parallel certainty inference systems, taking both processes into account, is to determine which states of the world are most likely to hold, numerical inferences typically being used for the first process. This is the situation in both MYCIN and PROSPECTOR, as well as the rule based systems described in Chapter 2 for heat exchanger selection and physical property method selection.

The use of numbers in inference rules are covered by a number of terms, "confidence factors", "levels of sufficiency and necessity", etc. The neutral term "degrees of belief" suggested by Shafer(1976) is used for discussion purposes here.

A simple analysis of inference rules involving degrees of belief is as follows: the premise or antecedent of a rule can be thought of as evidence, for or against, the conclusion of the rule. The overall degree of belief associated with a rule may be thought of as the conditional probability of the conclusion given the evidence. The analysis, of course, is not restricted to rule based systems. Inferences involving slots of frames can be described in the same way. This could even by applied to inheritance based inferences if necessary.

Two simple kinds of numerical degrees of belief can be identified:

1. Those specified a priori to qualify the conclusions, ie initial degrees of belief.
2. Those derived from the initial values as reasoning by a system proceeds.

There are considerable theoretical objections to both types of degree of belief. In the case of initial degrees of belief it is perfectly valid to question what the number means. For example, suppose in the heat exchanger selection system there is a rule that suggests a shell and tube exchanger with an associated degree of belief of 0.75 given some process conditions in the rule premise. Is it that 75% of shell and tube exchangers in some distribution exhibit or satisfy the process conditions? A more likely explanation is that it is a subjective estimate of the degree of belief that a shell and tube exchanger is a good option relative to other known types of exchanger. More generally, such initial estimates will represent a summary of the reasons for believing and disbelieving the inference. Once these reasons have been rolled up into a single number an inference mechanism no longer has access to the reasons for justification of an inference. This may be important information that is lost to the system. Personal experience shows that the situation is, in fact, often even worse. Degrees of belief in rules are often "tuned" on test cases until a set of rules as a whole produce the desired rankings of options for the test cases studied. What the degrees of belief in rule consequents then represent is open to question.

The second form of numerical degrees of belief are those generated or derived by the system when making intermediate inferences. There are two aspects to consider:

1. updating the degree of belief in a hypothesis to account for a new piece of evidence - this is called "pooling" evidence.
2. updating the degree of belief in a conclusion to account for uncertainty in the premise of an inference, ie uncertainty in the evidence itself - this is called "propagating" degrees of belief.

Problems with pooling evidence in a Bayesian framework are well documented and outlined below. It is essential to note, as regards propagating degrees of belief, that Bayes theory does not even allow for uncertainty in the evidence itself. All it allows is a degree of belief to be calculated for a hypothesis given some evidence. For example, it is not possible to say "calculate the belief in hypothesis H given that there may be evidence for H". Yet it is certainly the case that evidence used in an inference rule may often be far from certain, eg estimates of physical property data, process conditions, etc.

If this situation is to be handled then some ad hoc procedure must be devised to account for the uncertainty in the premise of the rule. This procedure is usually referred to as a combining function. Outlines of the combining functions for MYCIN and PROSPECTOR are given in Appendix C.

The parallel certainty inference approach means that this problem is not quite as bad as it seems, ie first reason assuming certainty then revise the belief in the conclusions in some heuristic way. The implementation details of most systems, however, leave much to be desired.

As regards pooling evidence, ie updating a degree of belief in a conclusion given one or more pieces of (certain) evidence, the situation is better defined. In this case a combining function pools evidence for a single hypothesis. Bayesian inference is one such combining function that allows the conditional probability of a hypothesis to be calculated given a body of evidence.

Two other approaches, Dempster-Shafer theory and fuzzy set theory, are described later.

4.3.1.1 Bayesian Inference

The basic form of Bayes' theorem is:

$$P(h|e) = \frac{P(e|h)\ P(h)}{P(e)}$$

That is, if the prior probability of an event or situation h is known, P(h) as well as the probability of a piece of evidence P(e) and the likelihood of evidence e arising given situation h, P(e|h), then the posterior or conditional probability of h occurring given e, P(h|e) can be calculated, eg the probability of a rise in stock share price given the announcement of a merger, the probability of a plant shutdown given a compressor problem, etc.

The first problem comes from finding probabilities for events and pieces of evidence,

P(h) and P(e). In many process design situations these may be very difficult to establish. Detailed plant operating records may be a source of information for certain types of event, eg pump failure. Even if this data can be found there remains the problem of establishing the probability of e given h, ie P(e|h). This will only be possible in situations where sufficient records are kept. For example, if e stands for "temperature > 500" and h stands for "exchanger type = shell and tube" how are accurate probabilities to be established?

In the case of multiple items of evidence the problem is somewhat exacerbated. The form of Bayes' theorem for n competing hypotheses given n items of evidence is:

$$P(h_i \mid e_1 \& \ldots \& e_n) = \frac{P(e_1 \& \ldots \& e_n \mid h_i)\ P(h_i)}{\sum_{j=1}^{n} P(h_j) P(e \mid e_1 \& \ldots \& e_n \mid h_j)}$$

The denominator of the equation implies that the conditional probabilities of all possible combinations of evidence for all possible hypotheses need to be known. This requirement is completely impractical in many situations, especially in process design where inferences are required on previously untried or tested situations. To avoid this problem the assumption of conditional independence is usually made:

$$P(e_i \& e_j \mid h) = P(e_i \mid h)\ P(e_j \mid h)$$

That is, if h is true then the observation of e_i is independent of e_j. This reduces the form of Bayes' theorem above to

$$P(h_i \mid e_1 \& \ldots \& e_n) = \frac{P(e_1 \mid h_i)\ (P(e_2 \mid h_i) \ldots P(e_n \mid h_i)\ P(h_i)}{\sum_{j=1}^{n} P(h_j) P(e \mid e_1 \& \ldots \& e_n \mid h_j)}$$

It should be noted that this equation also assumes that the competing hypotheses h_i are mutually exclusive and exhaustive. This solution, however, is a major problem in itself. In many situations the assumption of conditional independence will not hold especially where there are strong functional relationships between variables representing items of evidence, eg pressure and temperature. This was recognised in MYCIN where an effort was made to keep dependent pieces of evidence grouped into single rather than multiple rules, see Shortliffe and Buchanan(1975). In order to translate MYCIN'S confidence factors for rules into a parameter for Bayes' theorem requires even stronger assumptions of independence of evidence, see Appendix C. Adams(1976) gives a detailed discussion of the shortcomings of this approach.

The problem of conditional independence is an important restriction in practical applications of Bayes' theorem. There are, however, a number of equally important, if not more so, problems associated with the approach and the effect they have on different aspects of reasoning. Some of these problems are:

1. The Bayesian approach does not distinguish uncertainty from ignorance, ie whether a degree of belief was calculated directly from evidence or indirectly inferred from lack of evidence. For instance, three possible states might be chosen but a prior probability, 0.6 say, is only known for one state. The Bayesian approach requires that prior probabilities be given for all choices. This forces some assignment of probability between the other states, eg 0.2 for each of the other two states. These values are based on ignorance not uncertainty. This information is lost to a reasoning system. Any subsequent explanation of decisions without this information will be at best misleading.
2. A single point measure gives no indication of the relative accuracy or precision compared in other competing hypotheses.
3. Single point measures fail to capture the distinction between reasons for believing and reasons for disbelieving. The experience of MYCIN, in which these measures are maintained separately - see Appendix C, indicates that this is very useful information, eg for explaining decisions with justifications for placing relative weightings on different types of evidence, etc with a single point measure this information is lost.
4. Bayes' approach usually requires subjective estimates of many prior probabilities for the problem domain. Humans are notoriously poor at providing good estimates for probabilities, see Bruner(1956), and it is likely that estimates from different sources will be inconsistent or inaccurate.
5. It is bad enough if reasons for believing and disbelieving are rolled up into a single point measure. It is even worse if the numerical measure does not pertain exclusively to belief or certainty at all. A good example, recognised by several authors, is confounding probability and utility in a single degree of belief associated with a rule. Utility, as used in classic decision-analytic techniques refers to the expected value or worth of evidence. In MYCIN, for example, confidence factors for rules can be made artificially high in order to get the system to notice the rule early on, ie there is an implicit high utility associated with the rule which is not encoded in an explicit way. This is a good example of influencing a search control strategy in an unreliable way. It also means that any explanation system without knowledge of this implicit information will fail to distinguish important or relevant hypotheses and those that happen to be likely as well.
6. Confounding different ideas into a single numerical value can have serious consequences when the way in which this value will be used or interpreted is considered. If the measure associated with competing hypotheses are to be directly compared then it is difficult to envisage how consistency in decision making could be achieved. Any system generated explanations will certainly be impaired.

It is perhaps unfair to associate all these problems with Bayesian-like techniques since many apply to other numerical combining functions used in pooling evidence, eg trying to represent different kinds of information by a single number. It is certainly true to say, however, that Bayes' theorem, or approximations to it, fail to adequately represent many of the forms of uncertainty introduce in this discussion.

(NB Point 4 mentioned the use of utility of evidence in decision-analysis theory. A good introduction to the ideas involved is Hillier and Lieberman (1980). Lindley(1985) is a more detailed discussion. The theory is concerned with evaluating the worth of evidence, the utility of obtaining more evidence and formulating problems as optimisations of the expected utility. The usefulness of the approach is

limited since the utilities for uncertain states of the world are assumed to be known. When utilities correspond to direct monetary cost estimates the approach seems plausible. It is difficult to see how it directly applies to more abstract situations, eg "the utility of an assertion?").

White(1985) presents a more detailed discussion of some of these deficiencies in the context of rule based production systems. In particular, the inadequacy of PROSPECTOR's propogation function is highlighted.

4.3.1.2 Dempster-Shafer Theory

Dempster-Shafer theory, like Bayesian theory, is concerned with pooling evidence regarding competing hypotheses. As in Bayesian theory a numerical value for degree of belief is used to represent uncertainty. It also assumes that the hypotheses being considered are mutually exclusive and exhaustive. The difference between the two methods, and it is a very significant one, is that in Dempster-Shafer theory a probability distribution can be constructed over all subsets of hypotheses rather than just individual singleton hypotheses as is the case in Bayesian theory. The reason why this is important is made clear below after some basic terms have been defined.

The frame of discernment, Θ, is the largest set from which subsets are constructed, eg the set of all singletons, and reflects the level of ignorance about which singleton is true. If there is no evidence available for any subset of Θ then the degree of belief assigned to Θ is 1.0. A "basic probability assignment" function, denoted by m, is used to assign probability to subsets of Θ. For example, if Θ = {h1, h2, h3}, it might be known that m ({h1, h2}) = 0.6. Two points immediately follow:

1. There is no need to allocate the 0.6 between h1 and h2 as is necessary in Bayes' theory. The measure 0.6 corresponds to the subset not the individual singletons.
2. There is no need to allocate 0.4 to h3. In fact, what is assigned is $m(\Theta) = 0.4$. Remaining assignments are to the set that reflects the ignorance about which singleton will dominate.

The degree of belief in a particular subset is simply the sum of the basic probability assignments of its subsets. The mechanism used to incrementally update the values of the belief functions as evidence becomes available is called Dempster's Rule of Combination. The method is straightforward and based on computation of intersections of subsets. A simple worked example can be found in Buchanan and Shortliffe(1984, Chapter 13).

The main advantage of this method is that it does not confuse ignorance with uncertainty by allowing subsets of hypotheses to be reasoned about. For example, in an early stage of design it might be known that heat is to be removed from a process stream. It is not necessary nor desirable to introduce different types of heat exchanger

into the reasoning process if all that is necessary is that a probability be assigned to the set of available types that could be used, ie all that can be really said is that it is likely a heat exchanger will be used for the duty - the type is irrelevant at this stage.

Of course, Dempster-Shafer theory is subject to most of the criticisms levelled at Bayesian theory in that it uses single point numerical measures of degrees of belief to represent uncertainty. It also relies on probability assignments which will still be difficult to establish. The use of subsets of hypotheses should help simplify this process. Any use of subjective estimates is problematical, as in Bayesian inference. Nevertheless, Dempster-Shafer theory offers a relatively simple alternative to Bayes' theory when incrementally updating degrees of belief in competing hypotheses using multiple pieces of evidence.

Full details can be found in Shafer(1976). A directly relevant example of its application is given in Garvey, Lowrance and Fischler(1981). The problem described is that of combining evidence of measurements from a number of process sensors in order to support inferences in a wider process model.

4.3.1.3 Fuzzy Logic

Fuzzy logic can be viewed either in terms of an alternative logic or simply as another from of combining function for pooling evidence. The latter viewpoint shall be discussed first.

The term "fuzzy" was introduced by Zadeh(1965), but see Zadeh(1978) for a full account. The central idea is that fuzzy sets are those that do not have precise membership criteria. An entity is described by its "degree of membership" of a set rather than either being described as a set member or not. Uncertainty is represented by what Zadeh calls a possibility: possibility statements being represented by a real number generated by a membership function, μ in the range 0 to 1. For example, the degree of membership of methane in the class of ideal gases might be 0.85, ie $\mu(\text{methane}) = 0.85$.

Zadeh has shown how possibility theory can be axiomatised to provide analogues for most of the operations in probability theory. As a means of representing uncertainty, fuzzy sets have made little impact on AI research. This is perhaps not surprising since the approach is beset with many of the problems that were identified earlier for both Bayesian and Dempster-Shafer theory. For example, instead of simply assessing prior probabilities, real valued membership functions must be devised instead. This seems an even more implausible task than generating subjective estimates of prior probabilities. Further, if the membership functions are devised on different grounds

then-

1. how are degrees of belief to be meaningfully compared?
2. what interpretation can be attributed to a degree of belief that is constructed from contributions from different types of evidence?

The discussion of precise semantics for the interpretation of frames showed that these sort of problems must be answered consistently in a useful representation system.

The fuzzy set approach suffers, of course, from all the usual drawbacks associated with methods that represent uncertainty as a single point numerical value, eg combining reasons of belief with disbelief.

The major criticisms of fuzzy logic arise, however, when viewed as a form of reasoning. Consider for example some object x, a fuzzy set S, and let the membership function be denoted by Sx. Suppose the definitions of set union, intersection and complement are given by

```
(A union B)x = max (Ax, Bx)
(A intersection B)x = min (Ax, Bx)
(complement of A)x = 1 - Ax
```

Now if F is the set of all false propositions, and T is the set of all true propositions then the "truth of proposition p" is taken to be t(p) = (1 - Fp + Tp)/2. Thus,

```
if b is (¬a) then t(b) = 1 - t(a)
if c is (a ∧ b) then t(c) = min(t(a), t(b))
if c is (if a then b) then t(c) = max(1-t(a),t(b)).
```

There are several problems with these definitions which seem completely unreasonable. For example, if the inference "if a then b" is thought to be certain then

$$t((\text{if a then b})) = 1$$

This would imply, however, that either t(a) = O or t(b) = 1, neither of which necessarily follow. More seriously, it allows some degree of truth to the proposition (if a then +a). This means that inconsistences will not be properly handled in a fuzzy logic. Other proposed definitions of the truth of "if" do not avoid this problem either, eg

```
t(if a then b) = min(1, 1-t(a)+t(b)), or
                 if t(a)<t(b) then 1 else t(b), or
                 min(1, t(b)/t(a))
```

For an example of a system that does make use of fuzzy logic see Baldwin(1983).

4.3.1.4 Presenting Recommendations

Before leaving the topic of numerical presentation of uncertainty a final potential problem in their practical use will be highlighted. The problem is that of listing

conclusions or recommendations along with the calculated degrees of belief and how these values will be interpreted by a process engineer (or even another module in a blackboard system).

Consider, for example, the simple statement "calculate the fugacity using the cubic chain of rotators equation of state, confidence = 0.72". If the value of 0.72 is accepted at face value without ensuring that the engineer is aware of the justification for the value then the recommended actions could well be misinterpreted. For example, the following situations could all arise:

1. The method should be accurate but guesses had to be made to estimate necessary parameters for the particular fluid in question, thus lowering the overall degree of belief.
2. It is not yet known if the method will accurately estimate the fluid in question, eg if strongly polar.
3. It is known that method is only a rough approximation and is inaccurate for most fluids.

If the justifications are not properly explained then the (novice) engineer could easily gain a false impression of the relative worth of estimation methods, especially if a normally very approximate technique is recommended simply because of lack of data. The calculations should certainly be updated when relevant data for more precise calculations does become available. This can only be done in a consistent way if both the engineer and the representation system itself are explicitly aware of the justifications for decisions in terms of how uncertainty has been represented.

There is also the subtle problem of the values that are used and the effect on an engineer. Although I have no experimental evidence it is my belief that engineers will intuitively react in a different way to a degree of belief of 0.99 then one of 0.57 say. In the former case the certainty of the conclusion will be accepted without argument or further consideration much more readily than the latter. The different values, however, may simply have resulted from different sources of subjective estimates of prior probabilities having been used when there is, in fact, the same justification for each option. The relative values of degrees of belief in options also has an effect. The set of values {0.99, 0.98, 0.96}, say, will intuitively be interpreted differently from {0.69, 0.68, 0.66} or {0.8, 0.7, 0.5}. Even if the probabilities are self consistent and statistically valid it is important that the factors that were taken into account when generating these probabilities be communicated to the enginner. Otherwise the absolute values, if displayed directly, may cause the conclusions to be interpreted in unwarranted ways. This is extremely important in an interactive environment, as a process design environment surely will be, if there is any dependence or reliance on the engineer to provide feedback or update information. The engineer will be much

more likely to worry about reported degrees of belief at the 0.5, or less, level then he/she will at the 0.99 level. If numerical representation of uncertainty is to be used at all then there should at least be a mechanism to indicate how the values were calculated and what sorts of implicit justifications were used in any subjective estimates of prior probabilities. The ideal solution, at least from the point of view of integrity of knowledge, is that these justifications should be represented and reasoned about explicity by the system. Initial steps in this direction are introduced below.

4.3.1.5 Endorsement Based Reasoning and Truth Maintenance

The significant problems that are associated with numerical measures of degree of belief have led several AI researchers to question the whole approach, not least on the grounds of investigating improved or more plausible models of human reasoning about uncertainty. Typical of the doubts concerning numerical representations of uncertainty are those of Hayes(1974): "Introspection does not suggest to me that intuitive reasonings are essentially imprecise; still less that they are precise in terms of a real valued truth-value in the unit interval (which is what fuzzy logic would have us accept). Even ignoring introspection, fuzzy logic does not seem very useful, for where do all these numbers come from ?".

The last point concerning the origin of "numbers" is especially relevant in many simple process design problems. Much of the knowledge involved in the problem solving process is in the form of "control information", eg applying principles in order to model situations under known assumptions, rather than use of observations with associated probabilities. It is my opinion that it would be very difficult to generate meaningful numerical measures of uncertainty that would consistently apply to diverse or complex situations.

Consider again the basic concept of parallel certainty inferences. If an inference is uncertain it can be augmented by another whose purpose is to calculate the certainty of the first. The Bayesian, Dempster-Shafer, and fuzzy set theories are all examples of this approach. There is no reason, however, why parallel certainty inferences should be solely numerical inferences, if at all. The purpose of the inferences is to indicate how much to believe or disbelieve the conclusions to which they refer. Since the overall goal of the approach is simply to determine which states of the world out of several, or many, options are most likely to hold it is possible that symbolic reasoning techniques could be used instead, or in addition to numerical measures of uncertainty.

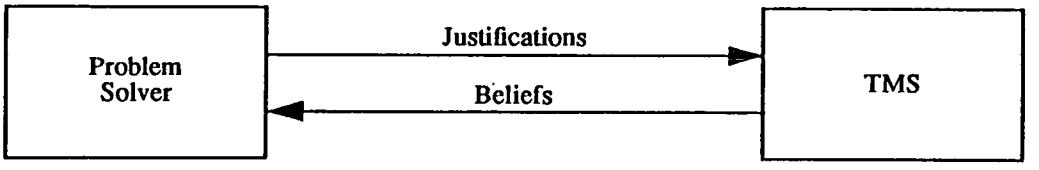
Hayes(1974) makes the point that words like "large, small" are often used, or intended, in a precise way, eg "the boy is small'. What is not precise is the vague

measuring scale associated with the word. The scales used are usually relative measures and context dependent, eg "a small boy" and "a small hill". Hayes suggests, and I agree, that the sense of vagueness be kept localised "within the word used" and should not affect or interact with other unrelated vague measures or inferences. This approach allows different probabilistic or fuzzy measuring scales to coexist within a more precisely defined semantic framework.

Endorsement based reasoning, Cohen(1986), is an example of a symbolic approach to reasoning about uncertainty that does not involve numerical representations of uncertainty or degrees of belief. In order to understand the use of endorsements in reasoning about uncertainty it is necessary to appreciate the basic ideas involved in truth maintenance systems. These are very briefly listed below.

Truth maintenance systems have been referred to in passing in chapters 2 and 3 as means of recording justifications associated with inferences. They are, in fact, much more involved than this. The basic system outlined here is that of de Kleer(1986 a, b, c). This system is an assumption based truth maintenance system, ATMS, and is a development of the TMS of Doyle(1979). Both systems consist of two components: a problem solver and a TMS, see Figure 4.1. The problem solver contains the domain knowledge and any inference mechanisms. Each inference made is reported to the TMS in the form of justifications. The goal of the TMS is to establish what data are believed or disbelieved given the justifications so far.

Figure 4.1: Relation of a TMS to a Problem Solver



Every problem-solving hypothesis is treated as an "assumption" in the ATMS. A set of assumptions is called an "environment" and the set of all data derivable from the

assumptions is called the "context" of the environment. If "false" can be derived from the assumptions then the environment is inconsistent and does not have a context.

Associated with every problem solver datum is the minimal set of environments from which it can be derived. This set is called the "label" of a datum and is used to establish whether a datum is present in a context or not: "a datum is in a context when the assumptions of the context are a subset of any of the environments of the datum's label".

A justification is written as

$$\alpha_1, \alpha_2, \ldots => \beta$$

where $\alpha_1, \ldots, \beta$ are problem solver data, assumptions or falsity.
A justification describes how a node, ie a problem solver datum, is derivable from other nodes. A justification has three parts:

1. The node being justified, β, called the consequent
2. A list of nodes, $\alpha_1, \ldots \alpha_n$, called the antecedents
3. The problem solver's description of the justification

The justifications can be interpreted as material implications,

$$\alpha_1 \wedge \alpha_2 \wedge \ldots \alpha_n \Rightarrow \beta$$

As such, they are simply propositional Horn clauses, see Appendix B. The nonlogical notation "=>" is used because an ATMS does not allow negated literals and treats implication in an unconventional way.

Finally, data in an ATMS are represented by nodes. A node is a data structure of the form:

<datum, label, justifications>

There are four types of nodes: premises, assumptions, assumed nodes and derived nodes. A premise, p, holds universally and has no antecedents. The node

<p,{{}},{()},>

represents premise p in an empty environment, ie no assumptions.
The label of an assumption node is a single environment mentioning itself. For example, assumption A (upper case letters represent default assumptions corresponding to defaults in Reiter's logic) is represented by the node.

<A, {{A}}, {(A)}>.

An assumed node is neither a premise nor an assumption. For example, the assumed

datum b which holds under assumption A is represented as

<b, {{A}},{(A)}>

An example of a derived node is

<'pressure is high', {{A,B},{C}}, {(e), (c,d)}.>

This says the proposition 'pressure is high' is derived from either the node e or the nodes c and d, and it holds in the environments {A,B} and {C}.
A node that does not hold in any environment is said to be an "out" node, other nodes being "in". Out nodes may have justifications and must not be removed as they may become "in" nodes in some future environment.

The main characteristics of de Kleer's ATMS are:

1. It is based on manipulating assumption sets rather than just justification sets, eg Doyle(1979).
2. It handles the notion of default assumptions. Contexts are the equivalent of extensions in default logic.
3. It can express disjunctions of assumptions allowing any propositional expression to be represented in terms of justifications and disjunctions of data.
4. It can handle non-monotonic justifications explicitly rather than search or backtrack through multiple contexts.

Full details of the system can be found in de Kleer(1986a, b, c)

In overall terms it is the task of the problem solver to make inferences about the domain and the task of the ATMS to determine what is and is not believed. In order to properly understand the role of an ATMS in a system that reasons about uncertainty it is important that the following restrictions or problems associated with its use be fully appreciated:

1. For the ATMS to perform as expected, i.e. manifest itself as justifications that achieve what the designer intended, the problem solver must supply the "correct" justifications. The notion of what "correct" implies is far from certain. Two common problems are underspecifying and overspecifying justifications.
2. If a problem solver underspecifies a justification by failing to list all of the antecedents then the label of the justification will be too general. For example, if "a => $\perp$" is recorded instead of "a $\wedge$ b => $\perp$", where $\perp$ represents false, then any context containing "a" will be inconsistent, which is incorrect. Parts of the solution space become unreachable violating the exhaustivity requirements of the ATMS.
3. If a problem solver overspecifies a justification by listing too many antecedents then the label of the consequent will be too specific. For example if "a $\wedge$ b => c" is recorded instead of "a => c" then c will not hold in all the contexts that it should if b is later shown to be false, i.e. c should hold in contexts containing "a" but not necessarily "a and b".
4. The problem solver must be carefully written to ensure that exhaustivity is assured. A common problem is that of querying the ATMS whether a datum x

holds in a context, eg (in? x) might return True if node x has a non-empty label and False otherwise. If a problem solver is poorly written and has code of the form.

(if(in? x)... else ....),

then exhaustivity will be potentially destroyed. If the test fails and this fact is not recorded then if x is later discovered to be in some context the relevant piece of code before the else statement will not be run. The same is true in the converse situation where the test initially works but it is later concluded that x has an empty label.
There are many control problems of this type in the interface between the problem solver and the ATMS. To be used properly an inference system should react to the non-monotonic nature of the ATMS (see point 8).

5. A fundamentally important point to realise is that the ATMS works at a propositional level. Any expression which has internal structure at the problem solver level, eg the inference

$$(\forall x)\, P(x) \rightarrow Q(x),$$

when recorded as a justification is treated as a propositional atom, ie an uninterpreted symbol (see Chapter 2 on the distinction between propositional and predicate logic). The ATMS simply establishes what symbols follow from given propositional ie other given symbols.
6. The previous point means that there are two inference procedures in the overall system, one in the problem solver and the other in the ATMS, which operate on the same expressions but treat them in different ways. This must be very carefully controlled, see point 4.
7. Problem solving is viewed as "a process of accumulating justifications and changing beliefs until some goal is satisfied". The process involves deciding what inference to make next based on currently held beliefs, recording the inferences as justifications and re-evaluating the current set of beliefs. The ATMS establishes belief solely on the justifications recorded thus far, not with respect to any logic in the problem solver. It is the task of the problem solver to ensure the overall process "converges".
8. A correct use of an ATMS requires the problem solving procedure to be implemented as a set of "consumers" attached to nodes. The consumers perform arbitrary computations but cannot make control decisions, based on an internal state, of which the ATMS is unaware. This model must be strictly adhered to if the ATMS is to guarantee its results (full details of the consumer problem solving architecture are given in de Kleer(1986c)).

Given the context of the discussion on the KRYPTON system, earlier it is apparent that a TMS, or ATMS, is simply one part of a much wider inference mechanism. The value of a TMS is the efficiency with which sets of beliefs can be determined. These beliefs can be used to help select the next inference rule to apply in some search or theorem proving application.

Given this understanding of truth maintenance systems it is now possible to assess Cohen's(1986) model of endorsement based reasoning about uncertainty. Endorsements are structures, like nodes in a TMS, that are intended to provide a representation and calculus for reasoning with symbolic information about

uncertainty. Whenever a domain rule is used or inference performed, its conclusion accrues one or more endorsements. Thus, endorsements, like TMS justifications, are simply records that a particular kind of inference has occurred. Endorsements are recorded by endorsers that also perform any other associated computations, similar to consumers in a TMS problem solver. The requirements of an endorser refer to the certainty of propositions. The endorsement of a particularly stringent endorser is therefore equivalent to a high degree of confidence or belief in a proposition, typically the consequent of a rule. Cohen suggests that individual conditions of a rule be required to have a certain level of endorsement before an endorsement can be made for the conclusion of the rule, eg only endorse a rule consequent if the parameters of the rule's conditions do not introduce any new uncertainty. Most conclusions will accrue several endorsements arising from parallel inferences. The certainty of a hypothesis is taken as its strongest endorsement rather than as the result of some summing up procedure of con evidence subtracted from pro evidence. This is because, Cohen argues, that different pieces and types of evidence are not equally believable.

From the description so far it would appear that there is little to distinguish endorsements from justifications in a TMS. This is not the case. Justifications in a TMS are used to decide if a conclusion can be believed, ie has some "set of support", but the "kind" of support is irrelevant. Cohen, however, suggests the use of many kinds of endorsements to represent different kinds of uncertainty, eg different kinds of evidence for and against a proposition. The overall confidence in a conclusion will depend on the types of its endorsements, eg direct, experimental evidence compared to indirect, estimated data. Some types of evidence will be more preferred than others. Unlike justifications, endorsements record aspects of inferences that are relevant to subsequent reasoning about their uncertainty.

Cohen describes endorsement based reasoning in the context of a program called SOLOMON. SOLOMON is a backward chaining, rule based, agenda driven planner or scheduler similar to those found in blackboard systems. The types of endorsements he introduces are, therefore, directly related to this type of problem. They are general enough, however, to be worth listing here:

1. Rule endorsements. Rule endorsements are intended to be the equivalent of numerical degrees of belief. They comment on properties of the rule's conditions, on the effect of the consequent, the "kind" of inference a rule represents, etc. Cohen identifies two groups of rule type endorsements: rule condition endorsements and inference type endorsements. There are ten types of rule condition endorsements. These include -

a. Maybe-too-general and maybe-too-specific - indicates that a condition may be overgeneralised or too specific to merit the conclusion, eg "temperature is high" might be too specific if the conclusion holds for many cases of "temperature is low" as well, as is the case in heat exchanger selection rules.
b. Supportive - a supportive clause is one that increases the certainty if true but does not necessarily lower the certainty in the conclusion if false.
c. Necessary - unlike supportive clauses, necessary clauses must be believed for the conclusion of the rule to be believed.
d. Flexible - a flexible clause is believable if an "approximately equal" clause can be found in the knowledge base that is adequately endorsed. This is similar to the use of compromises in the physical properties interpreter in Designer's Assistant.

There are a number of ways of endorsing negated conditions of a rule, ie not (condition). These include a version based on the closed world assumption and a "hard-not" version. The latter endorsement requires that either not(x) must appear explicity in the database and be adequately endorsed, or a mutually exclusive proposition to x be in the database and be explicity endorsed.

2. Inference Type Rule Endorsements: Cohen suggests that three types of inference endorsements are generally applicable namely model-based, causal and correlational. Model-based endorsements are principled explanations of why the state of the condition leads to a belief in the conclusion. A simple example would be the endorsement of some description of P-V-T behaviour by Charles Law, the ideal gas law etc. Causal endorsements simply relate directly causal expressions for which no "model" exists, eg "old age" and "loss of teeth". Finally correlational endorsements are used when the state in a condition is associated with the state in a conclusion in some, as yet undefined way. Both Cohen and Szolovits and Pauker(1978) suggest that many rules in production systems are categorical in nature and used without reference to the theory that underlies them. If this is the case then correlational and causal endorsements will be useful until more principled models can be established.
3. Task Endorsements: tasks are generated and put on an agenda when SOLOMON attempts to prove the conclusions of a rule. The endorsements of tasks refer to the "potential" conclusions of the tasks. The word potential is used since a task's conclusions are not believable unless its conditions are believable. There are four types of task endorsements:

    a. p-corroborate - task conclusion corroborates the conclusion of another task already on the agenda.
    b. p-conflict - conflict with another task on the agenda.
    c. p-potential-conflict - may conflict with another task on the agenda
    d. p-redundant - task conclusion derived by another task using the same rule and rule endorsement.

4. Conclusion Endorsements: SOLOMON does not endorse the conclusion of a rule itself, since this is fairly pointless other than to say it is appropriate (or not) given the conditions, but it does handle endorsements of conclusions of tasks as they are recorded in the knowledge base. Endorsement types include "unlikely" and "unwarranted", the latter being useful when a conclusion might be used but is not warranted.
5. Data Endorsements: Cohen suggests using "source", "type-of-data" and "accuracy" as general endorsement types. Little indication is given, however, of how these would be defined

Full details of the endorsement types are given by Cohen(1986). In addition, two case

studies are presented which compare the approach with a standard Bayesian approach.

Apart from agreeing with many of Cohen's arguments regarding the need to represent different types of uncertainty in a sophisticated problem solver, I find endorsement based reasoning interesting for a number of reasons.
First, it can be seen as providing a framework for a set of meta-predicates that can be used to represent difference types of uncertainty in a problem solving environment. The endorsement types proposed are very general in nature and can be applied to a wide variety of problems. In fact the notion of "typed" endorsements can be modelled on many-sorted logics.

Many-sorted logics have not been discussed in this thesis. In simple terms, function and relation symbols are sorted in the sense that arguments are restricted to terms of the appropriate sort, eg father-of might take terms of sort male in the first argument and human in the second argument. The types are referred to as many-sortal types and are usually disjoint and/or partially ordered sets, often represented as a tree or a lattice. In a sense many sorted logic is really an efficient extension of first order logic. The sort hierarchies bear a close resemblance to frame based inheritance hierarchies. A good introduction to many-sorted logic is Cohn(1987). The types in this instance would be meta-types such as data, premises, goals etc.

The major problem of endorsement based reasoning is providing a clear semantics to use when propagating endorsements over inferences, ie combining endorsements from individual conditions. In standard logic a problem does not arise since the semantics are denotational in nature, ie the truth value of a rule's conclusion can be composed from the truth values assigned to the rule's conditions. In Bayesian, Dempster-Shafer, or fuzzy set theory combining functions are defined to compute the degree of belief in the conclusion.

Endorsement based reasoning is more complicated since what is to be "combined" are reasons for believing or disbelieving. The simplest approach would appear to be maintaining ever larger sets of support for each derived hypothesis. This could only be done efficiently if implemented in a similar way to a TMS, ie the endorsements represented as special types of justifications. The problem is much more subtle, however, than might appear at first. For instance, consider the rule fragment:

```
if estimated-cost is low
   and .....
   then ....
```

Suppose that the clause "estimated-cost is low" has been recorded or derived by three

different rules. Further suppose that each rule found some reason, ie endorsement, for not believing the conclusion, ie there is not much certainty in this fact. In endorsing the conclusion of the rule there are therefore three "negative" endorsements associated with the first condition not one. The problem is deciding what endorsements to associate with the conclusion, eg one possibility is to include all three endorsements in a similar way to a TMS. An alternative interpretation is that three weak independent endorsements of the same fact constitutes, in a way, a stronger belief in the fact. If this is the case then some modified endorsement should be associated with the conclusion, eg the cost-estimate is likely to be low but what is now uncertain is the value of the cost-estimate.

The definition of a clear semantics for endorsement based reasoning is still an active research activity. It is my opinion that the problem is best handled by a TMS like approach, sets of endorsements being handled by the TMS component as part of the wider inference system defined in the problem solving component. This will clearly require some set of meta-inference rules to guide the use of different types of certainty when interpreting facts in domain level rules.

This type of approach to reasoning about uncertainty goes some way towards Hayes'(1974) suggestion that measures of vagueness be kept local and not used directly in interactions with unrelated inferences. It is also in keeping with the usual parallel certainty inference approach in that domain specific statements can be expressed as normal, certain propositions. The secondary process of deciding how much conclusions are to be believed can be separated out as a problem of maintaining consistent sets of endorsements.

In summary, endorsement based reasoning and the underlying techniques of truth maintenance systems provide a possible alternative to reasoning about different forms of uncertainty that does not need to refer to numerical statements of degrees of belief. Numerical measures of belief could still be used if desired. This is an important point since probability techniques still have an important role to play in well defined areas, eg hazard analysis studies.

4.3.1.6 Modal Forms of Logic to Express Uncertainty

The final topic that shall be discussed under the topic of reasoning about uncertainty is the use of modal forms of logic. It is only recently that serious attempts have been made to implement such logics in computationally tractable forms. This has been helped by the development of truth maintenance systems which have shown that reasoning within multiple contexts can be efficiently achieved.

The relevance of modal logics to the development of a process design environment is that they provide a formal means to represent and reason about imprecise or uncertain knowledge without recourse to numerical degrees of belief. Given the problems identified earlier with numerical representations of uncertainty it is well worth identifying some specific examples of modal logics.

It is extremely desirable to be able to reason about possibilities, hypothetical events, desired goals, uncertain information, etc, in a process design context. In the English language modes of possibility, necessity and permission are expressed by auxiliary modal verbs such as "can", "must" and "may". Similarly, tenses are expressed by auxiliary verbs such a "will" and "have". The modal logics mentioned here are simply attempts to formalise these concepts with an axiomatic framework.

There is no attempt made here to provide or discuss details of these logics. The standard text on modal propositional logic is Chellas(1980). This gives a thorough introduction to the main types and results of modal logic. Smets et al(1988) is an edited collection of papers on the implementation and application of modal and other non-standard logics. Turner(1984) gives a simple overview of many of these logics more suited to novice readers.

The simplest way to introduce these logics is by example. Alethic logic introduces a pair of sentence operators that are interpreted as "it is necessary that" and "it is possible that", represented by the symbols L and M respectively. Only one of these operators is basic in that the other is its dual defined by the equivalence.

$$Lp \equiv \neg M(\neg p),$$

for some proposition p. This is similar to the definition of the existential and universal quantifiers, $\exists$ and $\forall$, in predicate calculus, ie

$$\forall x.s \equiv \neg \exists x(\neg s),$$

for some formula s. A simple statement might be "it is possible to operate a column under vacuum". This might be translated as:

$$M(\exists x.\text{column}(x) \wedge \text{operating-pressure}(x,\text{vacuum}))$$

Deontic logic is concerned with permission and obligation to support models of the verbs "may" and "ought". Said(1985) is a good discussion of deontic logic in the context of interpreting legal definitions. This sort of logic would be useful in user models in a design environment, ie keeping track of what an engineer may or should do in a given context. It is also useful for expressing uncertainty about what may or

ought to be done during the execution of problem solving procedures. Said uses the operators O and P in his formulation of the logic.

Temporal logic introduces operators for the modes "sometimes" and "always", both in the future and the past together with their negations "never" and "not always".
The G operator expresses "always" in the future, ie Gp is true if p will be true in all future times, while the H operator expresses "always" in the past, ie Hp is true if p has always been true in the past. For example, H(compressor(expensive)), might be a typical statement.
Similarly, the operators F and P express "sometimes" in the future and past respectively. For example, stating that a particular plant will require periodic shutdowns at some point in the future might be expressed as

$$plant(x) \wedge F(\neg operate(x) \wedge ...).$$

The duality between the operators is given by

$$Fp \equiv \neg G \neg p \text{ and } Pp \equiv \neg H \neg p$$

A well known alternative approach to temporal reasoning is discussed in Allen(1983). This is based on maintaining beliefs between specified time intervals and embodies the notion of the duration of an event, ie recognising that events take time and that there is not an instantaneous change from one problem state to another. An interesting discussion on the use of temporal logics to describe computations is given in Moskowski(1986).

Finally epistemic logic introduces operators to indicate "believes" and "knows". The operator for belief is usually indicated by one of L, B or Believes. A simple statement might be "it is believed that the mixture is "azeotropic" which could be represented as
-

$$mixture(A) \wedge component(A,water) \wedge component(A,propanol) \wedge B(azeotropic(A))$$

There are many similarities between these formal systems. The important point for the purposes of this discussion is that they allow possible or hypothetical statements, ie forms of uncertainty or imprecise knowledge, to be expressed in a framework with well defined semantics. For example:

1. All formation rules of ordinary predicate logic are also formation rules of modal predicate logic.
2. If F is a formula and M is a modal operator then MF is also a formula

This, of course, was how Reiter's default logic and McDermott's non-monotonic logic were introduced earlier in Section 4.2.7. These logics are two more examples of

"modal" logics, in the general sense of that word. The common feature of these logics is that they allow, in a general sense, possible worlds to be reasoned about in which different beliefs hold. For example, the ATMS in Section 4.3.1.5 dealt with multiple contexts to handle non-monotonic and default inferences. It was suggested that endorsement based reasoning be formalised along similar lines.

The concept of the use of possible worlds as a uniform semantics for different kinds of modal systems was introduced by Kripke(1971). The idea is straight forward in the case of propositional logics: a (modal) formula is evaluated within a set of possible worlds which are provided with an "accessibility relation". The truth value of a modal formula depends on the possible world being considered. In simple terms, given a pair of modal operators L and M, Lp is true in a world if and only if p is true in all accessible possible worlds. The dual, Mp, is true if p is true in at least one accessible possible world. For example, in epistemic logic the beliefs of an agent are modelled as a set of possible worlds, as is the case with the contexts of an ATMS. In temporal logic the various possible worlds represent different states of the world at different time instants.

The notion of an accessibility relation is meant to capture the intuitive notion that certain worlds are accessible to each other while others are not. For example the accessibility relation in temporal logic will define the sequential order of the various time instants, only neighbouring states being accessible to each other.

Appendix D gives a very brief description of the formal nomenclature used to describe any such system. It can be seen that this closely resembles the description of the ATMS given earlier.
Each modal logic can, therefore, be characterised by identifying the modal axioms that are used in its formulation. These axioms all have standard labels. For example:

1. The T or knowledge axiom scheme: $Lp \supset p$. L stands for "is known", so T asserts "what is known in true". This is the usual interpretation of knowledge as true information.
2. The K or distribution axiom scheme: $L(p \supset q) \supset Lp \supset Lq$.
3. The S4 or positive introspection axiom scheme: $Lp \supset LLp$. This form of introspection says that if an agent believes a proposition then he believes he believes it.
4. The S5 or negative introspection: $Mp \supset LMp$. This is equivalent to $\neg Lp \supset L \neg Lp$. This says that if an agent does not believe a proposition then he believes that he does not believe it.
5. The Bercan scheme: $(\forall x\ Lp) \supset L(\forall x p)$

The weakest system is called *T* and only contains the knowledge and distribution axiom schemes. Adding the Barcan scheme and positive introspection yields the S4 system. Adding negative introspection yields S5. If the non-monotonic inference rule

'one cannot infer ¬p' - Mp,

ie If (not p) cannot be inferred then p is "believable" or "possible" or "consistent", is added then the non-monotic versions of S4 and S5 are obtained. McDermott's non-monotonic logic is simply non-monotonic S4, although this is now known to be incorrect and should be non-monotonic S5 without Lp ⊃ p, see Moore(1988).
Another important logic is Moore's(1984, 1985, 1988) autoepistemic logic. This logic involves complete S5 structures, see Appendix D, that is those structures in which every world is accessible from every world. The logic attempts to model ideally rational agents who reflect upon their own beliefs. An earlier discussion, Section 4.2.4.6, indicated that this type of reflexive reasoning is an important part of problem solving. The importance of this topic is outlined later in chapter 5.

For the moment, it is clear that formal modal logics, especially non-monotonic ones, provide a means of reasoning about certain aspects of uncertainty that will be important in a process design environment. Belief or certainty in a conclusion, as represented by sets of reasons for believing or disbelieving the conclusion, can be entertained by use of a possible worlds semantics. What is currently missing from the endorsement based model of uncertainty is a working set of restricted inference rules to use in such a framework.

All such logics, in predicate form, are undecidable. Any practical implementation will require good heuristics concerning the selection and application of inference rules when proving goals. The difficulty of providing such heuristics should not be underestimated given the experience of the development and performance of resolution based theorem provers. Development of truth maintenance systems, however, have shown how aspects of the overall task can be tackled efficiently.

A directly relevant example to process engineering is the recent interest in qualitative simulation, ie qualitative models of process operations, eg Stephanopoulos(1989), Waters(1989). The models are usually expressed in terms of confluence equations, as proposed by de Kleer and Brown(1984) in their formulation of a qualitative physics but see Kuipers(1986) for an alternative formulation. The changes in variables in a model are usually represented as one of three states: increasing(+), decreasing(-), or indeterminate. It is obvious that combining such variables together in a simulation through the use of confluence equations will result in huge numbers of possible combinations of state variables if implemented in a simple or naive manner. This has been borne out by Waters(1989). De Kleer(1986c) shows how various types of constraints can be handled in a consumer based problem solver that communicates with an ATMS. For example, the operating states of a transistor are "on, off,

saturated and reverse-active". A fragment of a qualitative model might be:

Q_1 e {ON, OFF, ...}
Q_1 = ON -> $\partial v_{b,c} = \partial i_c$
Q_1 = OFF -> $\partial i_c = 0$
...

The first confluence states that the voltage varies proportionally with the current when the state of the node Q_1 is ON. The values for voltage and current cannot be substituted into the second confluence equation because the states of the node are mutually exclusive and would contradict each other. De Kleer shows how a variety of other forms of constraint can be represented in a suitable form for use by an ATMS. Given such a representation it is possible to see how a qualitative simulation might be realistically implemented. For instance, consider a question of the form "will or can high pressure occur in a pipe or vessel?", as is typical of part of an Hazard and Operating study. Instead of selecting values for input variables, running a simulation and then checking all the possible outcome states, whether explicitly or implicitly via backtracking search, an alternative is simply to reason backward from the goal state and establish from which possible contexts this fact could be derived. This is the sort of problem in which good use can be made of an ATMS. The ATMS is only concerned with maintaining justifications recorded so far, the control or search strategy of the problem solver being irrelevant as far as the truth maintenance system is concerned. Any goal directed search, as suggested here, can therefore make good use of any domain dependent heuristics to guide the search or "simulation".
Although the form of uncertainty is somewhat odd in this example, the state or value of a discrete variable, it shows how reasoning can be performed in a possible world semantics.

4.3.1.7 Summary: Problems of Representing Uncertainty

The previous sections of this chapter were concerned with discussing the problems associated with the use of rule based production systems and frame based inheritance. The inherent structure of these systems says nothing about how uncertain or imprecise knowledge should be represented or reasoned about. In addition, it is frequently the case that no indication is given as to how to handle uncertain facts or conclusions correctly and the effects this has on the underlying inference mechanism.

This section has highlighted the common approaches to representing uncertainty as degrees of belief, as found in both production-rule and frame based systems. The vast majority of systems still rely on some variant or approximation of Bayesian theory. As a general means of reasoning about uncertainty this was shown to be far from adequate. Alternative methods of computing degrees of belief were also introduced, eg Dempster-Shafter theory and Fuzzy logic. Of these, the

Dempster-Shafer theory seems a good alternative but suffers from many of the same problems as Bayesian theory as it is still based on single point numerical measures of degree of belief.

In a complex problem solving environment, such as process design, it is necessary to be able to reason about different types of certainty in order to reason consistently when making inferences, applying production rules, inheriting attributes, using default information, etc. It is my opinion that the numerical approaches to representing uncertainty considered here are not adequate. Particularly difficult problems arise when uncertainties of a different nature are to be compared, or worse combined, in a meaningful way. Representing belief in a conclusion as a single number also increases the difficulty of providing meaningful explanations as to how conclusions were derived and the reasons why some conclusions are preferred to others. Finally, the provision of consistent sets of subjective estimates of prior probabilities, degrees of belief, etc is prone to error. Updating these probabilities, adjusting them to take account of new evidence, rules, etc is not a trivial task in medium or large scale systems.

Two more radical but related approaches were discussed as alternatives to representing uncertainty as degrees of belief. The first of these was endorsement based reasoning, itself closely modelled on truth maintenance systems. In this approach uncertainty is represented as sets of endorsements, ie reasons for believing or disbelieving, associated with each piece of evidence or derived conclusion. The major problem preventing the direct application of this technique is the development of a set of inference rules to propagate endorsements over inferences in a well defined way.

The second approach considered was the direct use of various types of modal and non-monotonic logics. One form of this type of logic, default logic, has recently found widespread adoption and use in frame-based or frame-like systems. The motivation for the development of truth maintenance systems also arose from the need to handle default information in a reliable way. There is no general way, however, of efficiently implementing theorem provers for such logics. Their practical use in problem solving environments will require the use of good probably domain dependent, heuristics to guide the selection and application of inference rules. The attraction in their use is that the predicate forms have the full expressive power of first order predicate logic in addition to the representation of modal formulas. In addition the concept of possible world semantics is clear and easy to understand in the context of maintaining different reasons, or sets of reasons, for believing the

certainty of derived conclusions. Justifications of conclusions are, therefore, explicitly available for both inference and explanation purposes. The development and attempted implementation of these logics is, however, still an active area for research in AI.

4.4 SUMMARY

The introductory sections of this chapter restated the representation requirements of an integrated process design environment in light of the research, presented in chapter 2, on state of the art symbolic programming techniques available at the time, ie rule-based, frame-based and hybrid blackboard systems.

It was concluded that although the overall framework of Designer's Assistant seemed a plausible one for a design environment, given the availability of flexible, hybrid interpreters such as CLAP, the details of both the control and interpretation mechanisms available were far from adequate. Indeed, the basic characteristics of an "intelligent" design system, far less an "expert" one - as indicated in section 4.1, suggest that current approaches to knowledge representation are not even close to satisfying the stated requirements. In order to emphasise the importance of this conclusion a detailed discussion of some of the more important problems associated with representation styles in rule and frame based systems was presented in this chapter. Many of the problems were seen to stem from the lack of a well defined semantics resulting in inconsistent interpretations of expressions in the particular representation formalisms. This was seen to be particularly true in the treatment of uncertainty based on numerical values of degrees of belief, a problem common to both rule and frame based systems.

Although some of the pitfalls could be avoided using advanced A.I. toolkits, there are few guidelines on how to tackle these complex issues and hence few, if any, safeguards against the creation of inconsistent or incoherent data structures or expressions in a system.

In an industrial context, one imagines that pragmatic engineering approaches to software development will tend to overlook such issues anyway. This will inevitably lead to significant problems in the development, use and maintenance of software. A particularly worrying example is the use of demons and the question of procedural attachment in general. Such techniques are problematical in that they will often introduce frame modifications or modify the inference mechanism in ways that can never be made explicit to the system. How such systems recover from unwarranted inferences and are able to remove the corresponding misinformation from working

memory is not a trivial problem and one that could have major ramifications in any design or diagnostic software.

It is clear from the literature that related research by other workers in the area of process engineering has not been concerned with addressing these important issues, see section 3.6. Rather much of the literature is concerned with formulating simple, idealised problems in isolation in a symbolic programming environment. It is not until such problems are viewed in a wider context that the considerable representation problems discussed have become much more apparent. The work presented in this thesis so far, however, has been directly concerned with the expressive adequacy of representation techniques rather than their detailed application in isolated problems. This was necessary in order to introduce and highlight the problems that will be involved in the development of an integrated design environment. The initial development of both CLAP and Designer's Assistant were first attempts that tried to address some of the problems involved by providing a variety of programming mechanisms that could be used in the development of a more sophisticated system.

Despite these developments it was concluded that the representation and interpretation techniques being used were inadequate to cope with the range and complexity of tasks and skills required in an intelligent design environment. In particular, it was far from clear how integrity of knowledge could ever be maintained in such a system using the techniques available. The various logic based descriptions of the information being represented and reasoning techniques required helped to confirm this conclusion. Simply reverting to full predicate forms of non-monotonic or default or epistemic logics, as opposed to propositional forms, is not a practical solution either due to the computational intractability involved.

Given the problems involved it seemed clear that both frame and rule based systems in their current forms did not appear to offer long term solutions to the problem of providing a suitable representation for the development of an integrated process design environment. It was decided, therefore, not to continue with the development and application of CLAP and Designer's Assistant but to investigate the fundamental issued involved in knowledge representation and the basis techniques that will be needed in a sophisticated system. This work is described in the remaining chapters below.

Chapter 5
A Study on the Nature of Concepts and Mental Modelling

The purpose of this chapter is to introduce three research topics from the fields of cognitive science, psychology and linguistics which, when taken together, provide the theoretical background for the representation developments presented in the next chapter. The topics considered are:

1. Conceptual Categorisation
2. Prototypical theories
3. Mental Models

Definitions of the important aspects of these fields of study are presented below in turn. The study of knowledge representation issues from a "cognitive viewpoint" is deemed both necessary and desirable given the following observations on the research presented thus far:

1. the use of either formal logic systems or frame-based systems, the latter based on intuitive but vague ideas regarding the use of "prototypes", do not appear to offer long term solutions to the problem of satisfying the requirements of an integrated process design environment.
2. The development of intelligent or even expert process design aids and tools requires a set of basic conceptual skills, eg reasoning from first principles, learning in a consistent way, flexible forms of communication, etc, that would be extremely difficult to implement in the current representation formalisms.
   The development of a unified representation scheme to handle a variety of types of knowledge or information that can be interpreted consistently for various diverse purposes needs to directly address the previous two points.

It was felt, therefore, that in order to gain any insight into the theoretical and psychological issues that will directly affect the basic capabilities of any such system, research of this sort must be undertaken.

Subsequent chapters demonstrate how an understanding of these issues can be used in the development of a representation system more directly suited to the highly complex problem area of process design. It should be clearly recognised that at the time of development it was thought that the problem of trying to provide a representation which was expressive but concise enough to cope with the very large amounts of information needed to solve even trivial problems was a very difficult one, perhaps computationally intractable. This is very well discussed in Hayes (1979b), (1985). Nevertheless, it was felt that the primary research aim of this thesis was to investigate and develop an adequate representation system capable of handling fundamental (eg basic laws, axioms, etc), heuristic and procedural methodologies in a problem solving environment. This is in contrast to the alternative option of demonstrating limited application of, what I believe to be, inadequate representation techniques on simplified, isolated problems. It is in this context that the following

discussions should be understood. Finally, it should also be realised that the research of other workers presented below does not represent a single line of investigation or research area. Instead, I have drawn together a number of essentially independent works, but with obvious common themes, into an elective set of ideas for my own particular purposes.

5.1 Knowledge Representation - The Epistemological Level

As an introduction to the representation issues of interest here, the analysis of Brachman (1979) (but see also Woods (1975)) concerning the confusion of interpretation of frame-like structures and links in a network is briefly summarised. In this work Brachman identified the importance of what he called the "epistemological level" of knowledge representation.
(Epistemology is a branch of metaphysics which deals with the nature and validity of knowledge. Metaphysics is a branch of philosophy concerned with the "ultimate nature of being and knowing" Penguin English Dictionary (1979), or more usefully, the study of ultimate abstract principles).
The word epistemology is used here to mean a study or theory of the nature of knowledge. It is this knowledge structuring level that will be seen to be of considerable importance in much of what is to follow.

It is clear from the discussion in chapter 3 that much confusion has arisen regarding the interpretation of frame-like structures. Intuitive interpetations of the semantics of frames can be traced back to the work of Minsky (1975) who suggested the use of frames to represent concepts. In fact, much of the content of frames is seen in the earlier development of "semantic", or "associative" networks. It is not surprising to find then, that there existed considerable confusion in the status and interpretation of semantic networks. This confusion is well discussed in Brachman (1979) and Woods (1975) and only the main points are highlighted here. The notion of a semantic network is generally attributed to Quillian (1967) who proposed an associated network model of "semantic memory". The original intent was to try and capture the objective part of the meanings of English words in a formal network representation. Quillian's representation consisted of nodes, considered to be "word concepts", interconnected by various types of associative links. Links from one node could point to other nodes, also representing word concepts, a combination of links and nodes together making up a definition. The overall structures of the representation was, therefore, an intricate, interwoven network of nodes and links. Quillian considered each word concept to be the "head of a plane", the plane holding the definition.

Within a plane pointers were used to represent the structure of the definition. A small set of pointer types were proposed including:

1. Subclass - a word concept that is a subclass of another.
2. modification - some modified use of another word concept.
3. disjunction/conjunction - the usual logical connectives.
4. subject/object - to indicate subject/object relationships between nodes

Pointers were allowed to point outside the defining plane, ie definitions could refer to words that had their own definition, introducing the usual type/broken definition.
Finally, Quillian introduced the notion of a propagating or "spreading activation" search along the links between nodes as a general inference mechanism. This idea has been used since in many systems, most notably NETL (Fahlman 1979). Touretzky (1986) discusses in detail the flaws in this approach, see also section 4.2.7. An important part of the inference mechanism was the use of "subclass" and "modifies" pointers. It was taken that properties true of a class were assumed to be true of all its subclasses except for any modifications, the latter simply being a combination of an attribute, a node within a plane, and a particular value for the attribute. Quillian's networks did not distinguish between generic definitions and instances, ie allow for the process of instantiation. This was addressed, however, in Carbonell's (1970) development of Quillian's networks.

The important point is that intuitive understandings of Quillian's basic ideas were widely applied to very different situations, many applications adopting their own, ad hoc semantics when interpreting the networks. In many cases the semantic network is simply viewed as a useful data structure. It is at this point the use of the term "semantic" begins to lose meaning as the concept of a representation language is lost: the term associative network is therefore preferred.
Consider a heat exchanger, hel01, represented by the two nodes:

```
he1010 --is_a--> heat-exchanger
```

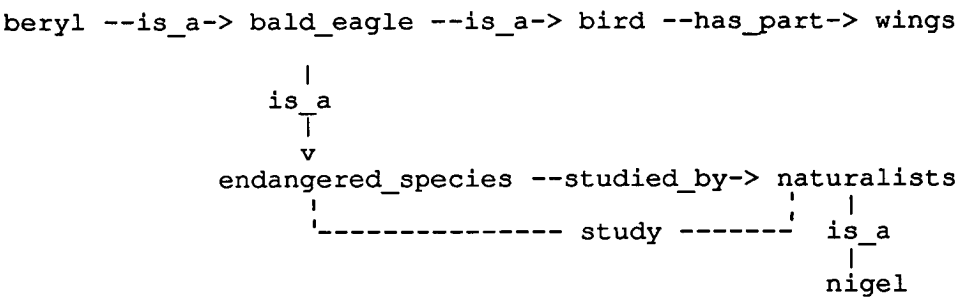
The two nodes are linked by the infamous is_a node corresponding to Quillian's subclass pointer. Inheritance of properties can be represented in a deceptively simple way, eg

```
he101 --is_a--> heat_exchanger --has_part--> hot_stream
```

This is adequate for binary relations but presents difficulties for higher arity predicates. A common idea developed was to use "case structures" that allowed a node to have a set of outgoing arcs grouped together. Early examples of this approach in the area of natural language understanding, and in particular the analysis of verbs, are Fillmore (1968) and Simmons et al (1973). The case first referred to agents, time, location etc, but little attention was paid to hierarchically classifying information. A

similar line of research is given in Rumelhart and Norman (1973), Norman et al (1975). Nodes were used to represent concepts, events and episodes, ie sequences of events. The interpretation of links between nodes, however, was not at all clear. In particular, the is_a link was used to denote both subset relationships and type- token relationships.

Ross (1985) gives the following example:



A spreading activation search might erroneously conclude that Nigel studies Beryl, the problem being that is_a has been used to represent both membership of a class and properties of the class itself. In addition, this network fails to recognise that some class properties are not inherited by the class members, eg Beryl herself is not an endangered species, merely a member of one. In many of the semantic network publications the significance of the relations denoted by links is not properly explained or defined. This leads to the confusion indicated above, and discussed at length in chapter 3 on the problems of interpreting frame-like structures. (In many respects, frames are simply an extension or reorganisation of the ideas involved in case-structures in associative networks). The contribution of Woods (1975) is very important as it challenges the logical adequacy of semantic network notations. In his later review, Brachman (1979) summarises the research that attempts to deal with these problems, eg the attempt of Hendrix (1976) to handle quantification by using formal groupings of nodes called partitions.

A first attempt to clarify some of the representation issues involved was Shapiro (1971). Shapiro introduced the distinction between the item, or conceptual, level of the network and the system level. The system level is to do with the structural connections between assertions of facts to items participating in those facts. The semantics of these system relations were defined by the set of routines operating on them. Item relations, on the other hand, are concepts that happen to be relational in nature and are represented by nodes like other non-relational concepts. The work is significant in that it was one of the first to separate out primitive system relations from higher level conceptual relations. This helped make explicit two different kinds of levels and types of representation.

The review of Brachman discusses this latter theme in some detail. Brachman

identifies four levels at which semantic networks appear to be interpreted. Each of these levels consist of different types of primitive elements which are used for interpretation purposes. They are as follows:

1. The implementational level - the primitives involved are atoms, pointers, numbers etc, and are concerned with the data structure level.
2. The logical level - the primitives involved are logical entities such as propositions, predicates, logical operators, etc.
3. The conceptual level - the primitives involved are word-senses, case relations, etc
4. The linguistic level - the primitives involved are arbitrary concepts, words, expressions, etc.

These levels require some clarification. The implementational level makes no claims about the structure or content of the knowledge being represented. It is simply a data structure out of which more complex forms can be built. As such it can simply be viewed as a useful data organisation method.

The logical level takes nodes to be predicates or propositions with the links between them representing logical relationships, eg and, there_exists, etc. Viewed at this level a network has little to offer in representation terms over first order predicate calculus. It does, however, indicate how normally non- indexed statements can be meaningfully factored using the network as an organising guide.

The conceptual level is what Brachman refers to as the "real net". A network at this level is characterised by language independent conceptual elements, ie primitive object and action types, and conceptually primitive relationships out of which all other concepts can be constructed. The essential aspect to grasp is the notion of language independence and conceptual primitives.

Finally, the linguistic level takes this one stage further by considering the primitive elements to be language specific. In addition it is expected that the primitive elements will change in meaning as the network grows. For example, this might correspond to learning new uses or definitions of words. The links in these networks represent arbitrary relationships that exist in the world being modelled.

The problem with most semantic network representations is that they freely mixed primitives from the different levels of representation. This makes it very difficult to provide a clear definition of the meaning of expressions in networks in terms of the actual primitives being used. It is also the case that each system designer would adopt his own set of primitives to work with making comparisons between systems very difficult. Brachman (1979) and Woods(1975) give detailed discussions of these issues.

From his observations Brachman suggests characterising knowledge representations,

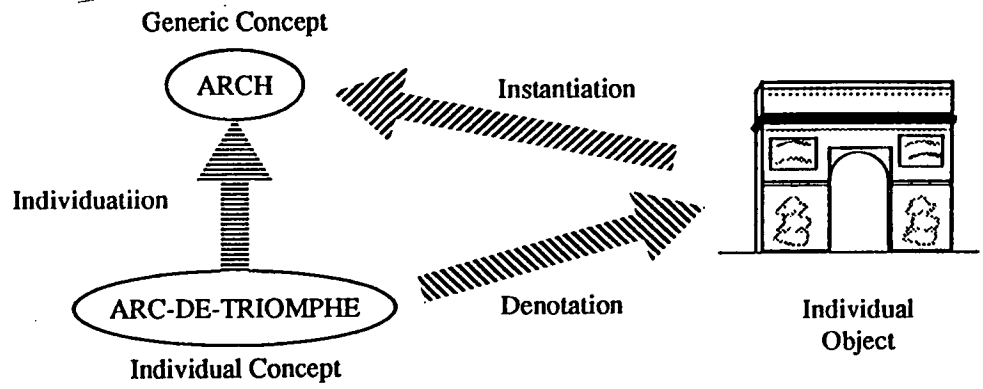
not just semantic networks, at a fifth level, an epistemological level lying between the logical and conceptual levels. Brachman informally suggests that this level is necessary by noticing how most systems require the creation of groupings, partitions, case structures, frames, etc. This in turn suggests the organisation of conceptual knowledge into compound units and their interpretation by some more formal means (this is in fact what is implicit in the use of frame based inheritance hierarchies). More formally Brachman observes that concepts, frame-like objects, etc, are thought of as intensional descriptions of objects in the world. (In simple terms an intensional description is one whose structure can be used to recognise a conceptual entity and compare it with others. The notions of intension, extension, sense and reference, etc, however raise deep philosophical issues that are touched upon later). If they are intensional descriptions, as claimed for example by Bobrow and Winograd (1977) regarding Units in KRL (see section 4.2.4) then there will be relationships describing how parts of an intension relate to the intension as a whole and how one intension relates to another. This sort of relationship must account for commonly held notions such as inheritance between concepts. Brachman's epistemological level is therefore concerned with "the formal structure of conceptual units and their inter-relationships as conceptual units (independent of any knowledge expressed therein)". In particular, it introduces knowledge structuring primitives rather than the specific knowledge primitives found at the higher conceptual level.

5.1.1 The KLONE system

Brachman developed the KLONE system to demonstrate the issues involved in explicity representing epistemological level relationships such as concept structuring and attribute-value inheritance. The details of the system are not of direct concern to the remainder of this study but the terms he introduces are worth repeating here. Full details of KLONE can be found in Brachman(1978).

The basic element of KLONE is called a Concept. Concepts are regarded as intensional entities and are used to represent the objects, attributes and relationships of the domain being modelled. No Concept is used to directly represent extensional, i.e. world , objects. To understand the difference between generic Concepts and Individuals, Brachman gives an example involving the Arc de Triomphe in Paris. Consider a generic Concept ARCH. Brachman uses the word "instantiation" to denote the relationship between the real world object and the generic Concept, i.e. the Arc de Triomphe in Paris is an instance of the Concept ARCH - see Figure 5.1.

Figure 5.1: Instantiation, Individuation and Denotation



The word individuation is used to represent the relationship between an Individual and a generic Concept, in this case between the Individual ARCH-DE-TRIOMPHE and the Concept ARCH. The Individual is said to denote the real world object. It is important to appreciate the intuitive difference in meaning between these relationships.

KLONE attempts to account for the internal structure of objects by use of Structural descriptions (SDs) to relate the various Role/Filler Descriptions (or more simply Roles). A Role represents the various attributes, parts, etc, that objects in the world are considered to possess. A Role contains two important types of information:

1. A description of entities that can be used for the value of and attribute in an instance of the Concept, how many values are expected, whether it is important to always have a value in an instance, etc. This is all information to do with the Role Fillers.
2. The functional role of Role Fillers within the entire conceptual complex represented by a Concept.

A Role is therefore a special epistemological entity that brings together both Role Filler information and their functional roles within the overall Concept. Its use arises from the key assumption that objects, in general, have a complex internal relational structure and cannot, therefore, be treated as atomic entities or simple lists of properties. This assumption is an extremely important one and shall be returned to later. In the meantime it should be noted that most frame-based systems do treat all attributes in a simple, uniform way using simple list structures.

Each structural description, SD, is a set of relationships between two or more Roles. In any Individual Concept the SDs must be satisfied in much the same ways as the Role Filler constraints are satisfied. The examples of SD relationships that Brachman discusses are simply first order logic statements which refer to the Roles by their Role Name, as defined by the system designer. For example, given the Roles UPRIGHT and LINTEL in the ARCH concept, an SD might state that all the UPRIGHTS in any

particular INDIVIDUAL will support the LINTEL,

$$\forall x.\ \text{upright}(x) \wedge \text{support}(x,\text{lintel}).$$

The only complication arises when the definition of one Concept is parameterised by another Concept, usually needed to handed existentials. The simple example Brachman (1979) discusses is the DISTANCE Concept with Roles FROM and TO. Suppose the VERTICAL-CLEARANCE is a Role of ARCH that is defined to be the distance between the arch's lintel and the ground. A DISTANCE individual, D1 say, could therefore have its TO Role satisfied by a GROUND individual. The filler of the FROM Role, however, is fixed and must be satisfied by the ARCH's particular LINTEL. D1 is therefore parameterised by ARCH, or some other Concept, in particular situations.

Brachman uses the term "paraindividuation" to describe this situation, D1 being described as a ParaIndividual.

An important epistemological relation that Brachman considers is inheritance. Individuation is a relationship between two Concepts such that the subdescriptions of the generic Concept are satisfied in some way by the Individual. There will, therefore, be a set of relationships between the generalised Role descriptions and the particular values found in the Individual. Inheritance is generally more complicated than this in that sub-classes or concepts often restrict or overwrite some of the parts of the description inherited from the parent concept. Despite the obvious differences in usage many systems use a single is_a link to construct the network or hierarchy. Brachman suggests, and I agree that this is an "oversimplification of a multifaceted relationship" when viewed from an epistemological point of view. What is required instead is a structured epistemological relationship between Concepts that transmits both the Role and SD descriptions as a whole. In an epistemological framework these entities cannot exist independently. More importantly, any modifications to the descriptions must be explicitly represented. In KLONE these modifications are represented as inter-Role or inter-SD links which state the relationship between the original Role or SD and the modified one. Modified Roles are then treated like other additional Role descriptions. KLONE only supports limited types of modification, eg Role restrictions.

In summary, Brachman proposes that the following five relationships must be accounted for in a knowledge representation language based on structured conceptual objects:

1. the relationship between a Concept and one of its Roles,
2. the relationship between a Concept and one of its SDs,
3. the internal structure of a Role, ie the relationship between a Role and one of its

facets
4. the internal structure of a SD
5. relationships between parts of SDs and Roles

The fundamental, and as yet unanswered, question from a research point of view is to establish the types of primitive necessary at the epistemological level. There are two important aspects to this problem, namely the "adequacy" of the representation and the definition of its semantics. The adequacy of an epistemological representation is concerned with how well it

can support the construction of higher level conceptual descriptions. the definition of appropriate semantics is much more problematical. In informal terms, this concerns the legal operations on primitives and establishing some consistent means for their interpretation. The term "semantics" is used here in its popular sense in which "meanings" of primitives are provided by the procedures that operate on them. Establishing the adequacy and semantics of epistemological, conceptual or linguistic levels of representation raises extremely complex issues. Such problems have long been considered in the areas of philosophy and psychology, and more recently cognitive science. These problems cannot, however, be simply ignored or delayed if any progress is to be made. It is precisely because epistemological issues were not well understood that such confusion arose over the interpretation of frame-based systems and semantic networks. Subsequent attempts to redescribe these systems in terms of formal logics does little to help the initial construction or expansion of a knowledge base in a complex problem domain involving a variety of types of knowledge, eg heuristic, procedural, factual, axiomatic, etc.

Addressing one of the basic research objectives stated in section 4.1, namely to provide a unified and systematic representation system and a consistent framework for its interpretation, would also appear to require tackling these issues. Given this background information is now possible to return to the main topics of interest identified at the start of this chapter, page 333. The main theme of the remaining discussion will be to show how all of conceptual categorisation, prototypical theories and mental models can be used to help clarify the essential nature of mental concepts and how these ideas can be used to develop more sophisticated representation mechanisms suitable for use in process engineering design.

5.2 Conceptual Categorisation

The purpose of this section is to attempt to briefly summarise the work of Bruner, Goodrow and Austin, "A Study of Thinking" (1956). This work was seen as an important investigation of the basic processes involved in human categorising or conceptualising. The psychological treatment was also unique in that it brought together in a common framework mental phenomena such as conceptualising, identification, inference and decision making which had previously been treated as separate but related issues.

Before discussing the main ideas of Bruner et al's work it is worth emphasising why the work is of immediate interest to this thesis:

1. The study will help to clarify the essential nature of concepts and how they are used in problem solving or learning situations.
2. The ideas presented give rise to plausible models or frameworks for human problem solving behaviour. These models involve significant cognitive tasks, eg object recognition or identification, learning new concepts, etc. At the same time they are simple enough to understand to provide a starting point for the implementation of an A.I. system intended to deal with real problems in an "intelligent" sort of way.

Due to limitations of space it will not be possible to discuss many aspects of Bruner et al's work. In addition the discussion will refer to mental phenomena such as "concepts", "attributes of concepts", "learning", etc. In order to avoid any confusion with notions of objects or frames, slots, attributes, etc, as referred to in computer programs in earlier chapters, all such mental phenomena will be written as capitalised words. For example, when referring to a Concept and an Attribute no assumptions can be made about what these words refer to or what their internal representation might be. In particular, the notion of an A.I. type object with a static list of attributes must be temporarily forgotten. (It will become clear, however, that A.I. type objects are over simplified attempts to capture the ideas expressed by Bruner and other cognitive science researchers. Indeed, the original motivation for Minsky's Frames (1975) was taken directly from such research.

The basic idea of categorising is most easily introduced by a simple example. The range of experiences of most people consists of a huge variety of discriminably different objects, events etc. Bruner et al cite the simple example of colour recognition. There is estimated to be more than seven million discriminable colours. In a relatively short period, two or three weeks, people came into contact with a sizeable proportion of these. It is clear that if any system, human or otherwise, was to fully register these colour differences in objects and treat each event as unique it would soon be overwhelmed by this and other tasks it had to perform. Indeed, even

the linguistic tasks of acquiring a vocabulary to cover such a world of colour differences in a precise way is something that no human ever attempts. It is probable, therefore, that any system that did attempt different forms of reasoning by fully registering differences between attributes of interest would have to perform massive amounts of processing when tackling real life problem situations. This naive approach is seen in the way many frame based systems simply use the list of attributes that define frames in a checklist manner, ie simply checking off each attribute in turn for each frame in turn. It is apparent that people do not normally process information or think in this way. Bruner et al claim that this is due to our ability to categorise. Categorising involves the ability "to render discriminably different things equivalent, to group the objects and events and people around us into classes, and to respond to them in terms of their class membership rather than their uniqueness" (page 1, Bruner et al 1956).

Treating different objects in an equivalent way is fundamental to the understanding of categorisation and how it might be modelled in an A.I. system. The process of categorising involves an act of invention. For example, the notion of a "pump" might be created or used to describe a certain collection of objects, "prime numbers" to describe an assortment of numbers, etc. The essential point about these categories is that once they've been mastered they can be used without further learning, ie new exemplars of the class can be readily identified and the category itself can become a tool or basic entity for further use.

Bruner et al claim that the learning and utilisation of categories represents one of the most elementary and general forms of cognition by which man adjusts to his continually changing environment. As such, attempting to understand the mental processes involved will be of key importance in the development of A.I. systems needed to help solve real engineering problems. New information, example problems, expertise, etc, will have to be continuously incorporated into a process design system. This will not simply involve creating new categories, eg to cope with new developments such as printed circuit heat exchangers, but requires knowing how to use new information to utilise existing categories more effectively in the future.

The remainder of this section will introduce the main types and aspects of categories of direct relevance to this work. Section 5.3.2 identifies the important differences between conceptual categories, as described below, and the intuitive ideas embodied in frames, objects, etc. The following discussion is necessary simplified since an in-depth description would require considerable background knowledge in the area of cognitive science. References are provided throughout this chapter for those interested in gaining a broader appreciation of the issues involved.

5.2.1 Basic Category Types

The existence of categories can only be inferred by observing common responses by people to arrays of objects or events. There are two broad types of categorising responses, namely identity responses and equivalence responses each of which involve different kinds of categories.

5.2.1.1 Identity Categories

Much of the work of Bruner et al is concerned with aspects of human thought which are either pre-verbal, and hence difficult to verbalise, or perceptual in nature. Such issues will not be directly addressed in this work. It is important to appreciate however the basic ideas involved since:

1. It avoids confusion when deciding what sorts of reasoning processes can be reasonably modelled.
2. Aspects of perceptual categorising will be of prime importance to A.I. systems concerned with direct visual, audio inputs etc, eg in the field of robotics.

Identity categories are a feature of categorisation that shall not be studied in any detail here. This type of categorising may be defined as identifying or classifying a variety of objects or stimuli as "forms of the same thing". Bruner gives two common examples to illustrate the idea:

1. The moon in its various phases, crescent moon, quarter, half and full moon, evokes the same nominative response of "the moon" despite its considerable change in appearance and change of position in the sky.
2. Flicking through a collection of photographs of the same person taken over many years gives a strong impression of the same individual in a process of growth despite considerable changes in external characteristics.

Identity categories are therefore associated with identity responses, that is identifying a variety of stimuli as having the same identity, ie being forms of the same thing. A study of this type of categorising would be very important in the development of any real-time system that attempted to perform sensor based identification of objects, events or trends and make any sort of inferences based on its responses.

This work is only initially concerned with process design in an abstract or idealised environment and will not involve any perceptual-like identification, eg using video or camera scans to read graphs or figures, monitoring process controllers etc. As such the notion of identity categories will not be considered further. It is important to note, however, that identity categories can arise in many "abstract" situations such as reasoning about an entity with changing characteristics in a simulation. In the absence of identity categories a system will have no choice but to assume single identities for the entities it reasons about.

It is essential that such limitations be clearly identified whenever they arise so that they can be more adequately handled in future developments.

Further details on identity categories can be found in Bruner et al (1956) as well as in several works on childhood learning (eg Piaget, 1951). Identity categories are difficult to characterise, identity responses being clearly affected by the early stages of learning. Piaget's work on child learning raises many issues that will have direct parallels in the development of more sophisticated A.I. systems that attempt to learn.

5.2.1.2 Equivalence Categories

The remainder of this discussion focusses on the concept of an equivalence category. Equivalence categories arise when individuals respond to "a set of discriminably different things as the same kind of thing or as amounting to the same thing". For example a set of different looking objects may all evoke a common response and be referred to as "pumps" or "heat exchangers" etc. A more abstract example is recognising that various different problem descriptions amount to the same thing, eg a non-ideal flash calculation described in a variety of ways. Similarly complex problems can often be broken down into sets of smaller problems that can be recognised as being equivalent to previous problems for which solution techniques are known.

Another common engineering design heuristics to treat an object or system as being equivalent to another, usually simplified in many ways, that can be more readily analysed.

These are all forms of equivalence categorising and are typical of the way in which engineers tackle different aspects of design problems.

There are three broad types of equivalence category, affective, functional and formal respectively. These are now briefly described in turn.

5.2.1.3 Affective Equivalence Categories

As with identity categories, this subtype of equivalence category is not dealt with in the work that follows. Affective responses that result in sets of things being grouped together as "alike" or the "same kind of thing" are difficult to verbalise. This is because there appears to be a lack of correspondence between affective and linguistic categories. For example, engineers and mathematicians often refer to "elegant solutions of problems". While there may be certain properties associated with such solutions, eg transformations of equations which result in simple analytic solutions, design of compact, energy efficient process unit modules, it would be very difficult to formalise what constitutes an "elegant solution" to a problem. Yet it seems clear that when an "elegant solution to a problem is presented it evokes a common affective response that allows the solution to be labelled "elegant". These exemplars undoubtedly influence the way in which engineers try to solve new problems or formulate general problem solution methods. As such they must ultimately play an

important role in the development of general problem solving techniques that involve making choices between partially formulated solutions to the same problem.

5.2.1.4 Functional Equivalence Categories

In the cases of functional and formal equivalence categories the problem of identifying the properties or characteristics of objects, events, etc used in categorising becomes, at least in principle, much simpler. This is because an equivalent response is now based on an "external function" rather relying on some internal human state to render a group of objects equivalent.

In the case of functional categories the objects in question must fulfil a "concrete and specific task requirement", eg things to pump fluids, things to store liquids in, etc. The "object" might be a conceptual abstraction such as an algorithm or procedure to perform a specific task, eg a procedure to solve a set of linear equations. The use of functional categories leads directly to the interesting possibility of formulating both interpolative techniques, ie performing tasks covered already by existing categories but in a non-explicit way, and extrapolative techniques, ie making guesses as to how perform new tasks. The development of this type of equivalence category for use in a design environment is covered in chapter 6.

5.2.1.5 Formal Equivalence Categories

Formal equivalence categories can be characterised by specifying the intrinsic attribute properties required by members of a class. They have the important characteristic that the diacritica of a class of objects can be stated short of describing their use. This sort of information is tied up in functional equivalence categories, see above. A great deal of information can therefore be described in terms of formal equivalence categories. This will include general object descriptions, eg pumps, vessels, etc, abstract object descriptions, eg problem solving methods, decision strategies, etc, as well as situation or event descriptions.

At first sight this definition may seem a slight variation on the definition of class members in an inheritance hierarchy to those unfamiliar with the work of Bruner et al. Whilst it is certainly true that when describing equivalence categories and their use Bruner et al constantly refer to required attributes or defining properties, it shall be explained shortly that the working definition of what is meant by a Concept is very different in nature from objects, frames etc, as described in chapter 3. In particular there are no direct parallels of functional equivalence categories in frame based systems.

It is worth noting that the careful specification of defining properties in science or engineering often requires the construction of artificial languages for description purposes. This is particularly noticeable in abstract areas of thought such as

mathematics or logic and to a lesser extent in engineering with its proliferation of "buzzwords". The emphasis in formal categorising is placed more on the Attribute properties of class members and less on what Bruner et al refer to as utilitanda properties. The degree to which categories are formalised depends on a gradual learning process. Bruner et al give the example of gradually moving from "things to hit this tent peg with", to "hammer", and from there to "mechanical force". At each step the definitions become freer of any reference to specific use. Formal categories are useful precisely because classes can be devised whose "defining properties are not determined by the suitability of objects to a specific task" (page 6, ib). Much of science and engineering is based on developments which take these abstract formal categories as the starting point for further investigation. It seems clear that any A.I. system which has to "intelligently" gather new information will have to operate in a similar sort of way and hence the interest in studying this work further.

Rather than move directly onto specific developments derived from the work of Bruner et al, as described in chapter 6, it is necessary to first describe the role of equivalence categories in the wider inference processes in which they are involved. One of the key differences between frame based systems and what follows is the recognition that the development of an on-going system that takes into account the evolving nature of categories is an essential part of "intelligent" behaviour.

5.2.2 What is Achieved by Categorising?

It is important at this point to try and give an overview of the potential benefits to be gained from developing an A.I. system based on the concepts embodied in categorisation theory. (Before reading the following comments it is worthwhile reviewing the basic requirements of a process design environment stated earlier in section 4.1.

Bruner et al cite the main benefits that are achieved by conceptional categorising. These benefits were not written with the development of A.I. systems in mind but are clearly relevant to the problem of designing sophisticated reasoning techniques:

1. Reduction of the Complexity of the Environment - the most important effect of categorising is to reduce the complexity of the environment with which a system has to deal. This is achieved by treating discriminably different events in an equivalent way. This in turn greatly reduced the cognitive load or amount of processing that must be performed when solving problems. This is true for a variety of different types of tasks, eg object recognition, problem formulation, etc. Furthermore it seems relatively clear how this aspect of categorisation can be achieved in terms of abstraction and use of defining properties to label useful categories. This is discussed in more detail later.
2. It provides the means by which objects and events in the world can be identified - this statement is somewhat obvious but highlights the assumption that some

mechanism exists that allows discriminable objects to be placed in equivalence categories. Being able to categorise entities is an extremely important aspect of an intelligent problem solving system. Much of the skill of a design engineer lies in the ability to recognise what is actually involved in a problem and then explicitly formulate the problem in such a way as to allow convenient solution. This inevitably involves identifying, ie recognising and categorising, the important aspects of arbitrary problem descriptions according to some criteria. The somewhat subtle importance of making the problem to be solved explicitly known to a reasoning system is one that has been largely overlooked in the A.I. literature. This is similar to the discussion of self-awareness earlier.

3. It reduces the necessity of constant learning. This means that once a category has been established it can be used for future acts of categorising without the need for any further learning. For example a system that has available to it a "car" category does not need to be told whether every object encountered is or is not a car. It "is" a car if it satisfies the appropriate defining criteria in some acceptable but as yet undefined way. The extrapolative value of categories is an important but obvious difference between categorising and the process of simply learning by rote that a group of miscellaneous objects are labelled by some class name X.
4. It provides direction for instrumental cognitive activity. Categorising objects using discriminable defining features makes it possible to know in advance what actions are appropriate or inappropriate to take in certain situations. Guiding reasoning or search processes is, of course, a fundamental problem in heuristic A.I. programs and little success has been reported in the general use of "meta-level" information in sensible ways as it is often "fired" regardless of the problem being solved.
   This direction of activity is also important when faced with new objects or events that cannot be categorised with any certainty. If the object has any discriminable properties that have been previously found to be important in defining other categories then an attempt can be made to "bracket" the new object. A process of continuous regrouping can then occur until an appropriate grouping can be found or a new category emerges. This form of instrumental activity has already been demonstrated in intuitive ways in several A.I. "learning" programs, eg Winston (1975), which operate in this bracketing fashion.
5. Categorising permits classes of events and objects to be ordered and related. It appears to be clear that people operate with category systems in which classes of objects and events are related to each other in various kinds of superordinate systems. The wide appeal of both inheritance hierarchies and semantic networks are evidence of this intuitive organisation of information. It is important to note that what is related - in order to give meaning to the world, as Bruner et al claim, are classes of events rather than individual events. Once an object or event has been categorised it is possible to "go beyond" the category by virtue of the placement of the category in the superordinate system and the causal relationships in which it is involved. This latter claim requires careful consideration of the sorts of problems that were encountered in the subsequent use of semantic networks are to be avoided - see section 5.1.

A final general observation made by Bruner, Goodrow and Austin regards the anticipatory and explanatory nature of categorising. Anticipatory categorising involves identifying the most definite defining attributes as early as possible in order to give identity to events and objects. This anticipatory strategy is an essential part of human behaviour and will be equally important in A.I. systems used in an advisory role, eg observing that a tank will overflow and taking appropriate action beforehand.

The anticipatory nature of categorising therefore provides necessary lead times to allow responses to objects or events to be adjusted.
This ability is particularly striking in the case of empty or fictive categories, ie formulating the existence of categories that have not yet been encountered. Two well known examples of postulated empty categories were the existence of the neutrino particle and the planet Neptune. In the latter case Neptune was not observed until 23 years after Bessel proposed its existence. This type of anticipatory thought is closely related to the claim, point 4 earlier, that categorising provides the necessary direction for instrumental activity.
It is this ability to "go beyond" what is currently known that makes the model of categorising such an attractive one for the development of a process engineering design environment. A large percentage of industrial design problems exhibit some novel or previously unencountered features that can usually only be solved by applying basic principles, axioms etc to the problem in a constructive, "insightful" way. Insight in this context is precisely the direction for cognitive activity alluded to above.

The five benefits gained from categorising are, of course, based on observations and experiments in human subjects. The psychological nature of Bruner, Goodrow and Austin's investigation meant that they were not concerned with proposing a detailed model of the categorisation process. It would be naive therefore to think that their arguments could be translated directly into some formal representation system for immediate use in a knowledge based system. The potential benefits to be gained, however, in terms of intelligent forms of reasoning in a knowledge based system based on the ideas of Bruner et al are such that it was decided to develop this work further. The challenge then is to identify what aspects of categorisation theory are amenable to direct representation and implement them in a suitable form. This work is described in chapter 6. The sections below briefly discuss the more important aspects of categorisation processes that Bruner et al identified and that are directly relevant to the representation development that follows. The discussion covers two main areas:

1. The nature of Attributes used to define categories.
2. The overall framework needed to support categorisation techniques

The first area is essentially concerned with what distinguishes equivalence categories from frames or A.I. type objects. The second area provides pointers on how to develop a general framework for a reasoning system based on categorisation ideas.

Two final introductory points must be made that are fundamental to all that follows.

The first point regards the difference between categorisation at perceptual and conceptual levels.

Much of categorising in everyday, real-life situations occurs at a perceptual level. Perceptual categorising consists, as does conceptual categorising, of placing stimulus inputs by virtue of their defining Attributes into certain classes. In both cases the identification involves a form of fit between the properties of the stimulus input and the category specifications. The principal difference between the two levels is the nature of the entities being categorised in terms of the "immediacy to experience of the attributes by which their fitness to a category is determined", (page 9, Bruner et al 1956). Many, so called, common sense understandings of Concepts such as force, pressure, motion, distance, etc, involve perceptual level categories. As noted earlier, as a Concept like pressure becomes more formalised there is a gradual formation of a more abstract, conceptual level category that is very different in nature form the perceptual level notion.

It is important to realise that the same linguistic words, eg force, pressure, heat etc, are often used to label a number of concepts at different points along the perceptual - conceptual spectrum. These distinctions must be kept in mind when deciding what can and cannot be reasonably represented in a representation scheme. Since the work here involves no equivalent of perceptual identification, eg in terms of sensor input, some means of working around perceptual level concepts must be provided in order to allow higher level conceptual categories to be constructed. A partial solution to this problem is presented in the following chapter.

The second essential point to appreciate is the invented or constructed nature of equivalence categories. In an engineer's world of experience, in the context of process engineering design, there exists a near infinitude of ways of grouping events and using defining criteria to form equivalence classes. Any single engineer will only ever make use of or become familiar with a small number of such groupings and criteria. The important question from an A.I. point of view is why certain groupings and categories are used in preference to others? As an initial reply Bruner et al suggest that "science and common sense inquiry alike do not discover the ways in which events *are* grouped in the world; they *invent* ways of grouping. The test of the invention is the predictive benefits that result from the use of invented categories. Do such categories as lions, atoms,...exist? In so far as they have been invented and found applicable...they do".

Recognising the invented status of equivalence categories greatly changes the emphasis on how conceptual categorisation should be viewed and studied. The development of equivalence categories can be assumed to involve a continuous

reformulation process that attempts to take account of past events and current requirements in order to meet some internal needs. This is an important point and one that underlies much of the structure of the proposed knowledge representation system described in chapter 6

5.2.3 Attributes, Attribute Combinations and Concept Types

This section introduces the basic observations of Bruner et al on the nature of Attributes that are used to help define equivalence categories. The discussion that follows is greatly oversimplified and the reader is referred to the original work for full details. Due to the oversimplification it may appear at first sight that Attributes, taken at a surface level, are very similar to the slots of frames or objects. It is not until the discussion in the following sections on the problem solving and learning aspects of categories is presented do the differences begin to appear. A comparison of these ideas with the use of frame based inheritance hierarchies is given in section 5.3.2.

In general terms an Attribute is simply a descriptive term relating to some object or event which can vary either discretely or continuously in value. The nature of the descriptive terms used can vary from affective to formal in that "elegance and beauty" can be Attributes in the same way that "volume, mass, defining geometry" could be. In much the same way that the "invention", use and labelling of categories is seen from a nominalist point of view, so too is the selection of particular Attributes to suit particular needs. That is in many situations there are no ultimate rules regarding the selection of Attributes. In the case of technical terms or ideas in scientific or engineering fields, ie areas involving the use of formal equivalence classes in general, there is however considerable social consensus as to the relevancy or need of certain attributes, eg the necessary Attributes needed to classify a certain shape as a triangle, or what constitutes a heat pump cycle, etc.
Attributes may be required to take on certain values or may be allowed to vary over a range of discrete or continuous values. The range of allowable values has important consequences in the selective use of Attributes and is closely tied to the particular learning process used in the initial formation of concepts in which the attribute is used. In general, the range of positive or acceptable values for an Attribute will vary as a function of the stage of category learning and typically narrows as a Concept becomes more fully understood. This can often correspond to a reduction in the number of Attributes used, as irrelevant Attributes are ignored as learning proceeds.

In summary, Bruner et al describe an Attribute as "any discriminable feature of an event that is susceptible of some discriminable variation from event to event". What is important to note is that the nature of the Attributes can be very different and

contribute in different ways to the different types of equivalence categories, ie formal, functional and affective. The choice of Attributes and allowable attribute values is not a fixed decision and will vary with the degree to which a Concept has been learnt.

5.2.3.1 Defining, Criterial and Non-Criterial Attributes

There are two levels on which Attributes can be characterised:

1. Their nature - affective, formal, etc, as indicated above.
2. Their functional role - whether they are defining, criterial or non-criterial.

A criterial Attribute is some discriminable feature of an environment that is used to infer the identity or category placement of some entity. That is, a criterial Attribute functions as a signal that can be used as a basis for "going beyond by inference". In this sense any type of Attribute can be a criterial Attribute regardless of the nature of the Attribute.

Criterial Attributes are distinct from defining Attributes, Defining Attributes arise in formal equivalence categories and are external or official statements of the defining properties of a class as given, for example by legal definitions or scientific/engineering conventions. Whether or not the criterial Attributes actually used for category placement agree with the defining Attributes will depend on a particular person and situation.

Bruner et al give the simple example for "safe driving" with a defining Attribute of speed. Suppose, for example, the legal definition of safe driving is that speeds up to 50 mph are safe and allowable. There are two cases worth noting:

1. The criterial and defining Attributes are the same but vary in allowable values. A person may consider himself to be driving safely while driving at 65 mph.
2. The criterial Attributes used may only be part-related or even entirely different from the defining Attributes. A person may consider safe driving to be directly related to a perceptual notion of the car "feeling under control" regardless of speed.

As indicated in the previous section the choice of concepts and criterial Attributes is not universally defined but related to satisfying some internal requirements or needs. In most cases involving some sort of social interactions the advantages and potential benefits resulting from the use of one set of Attributes in a criterial way must be weighed against other possible Attributes, eg the defining Attributes if they are known. In the case of safe driving, a 60 mph driver must consider the risk of a speeding fine. In many engineering contexts defining Attributes are not so clearly or simply stated.

For example, deciding whether to categorise a separation problem as ideal or non-ideal and model it appropriately may depend on the perceived benefits to be gained from a more rigorous model, the "resources" available to solve the model,

availability of experimental physical property data characterising the system etc. In such cases the use of criterial Attributes is of considerable significance to the problem solving process.

The distinction between criterial and defining Attributes is also fundamental to any learning process, ie the process of Concept attainment. Any competent engineer or A.I. system that has to act or perform in some social environment must be aware of or be capable of learning which Attributes are to be taken as defining if sensible communication of information between systems is to occur. This is true in both learning and explanation type situations. However, it should be emphasised that the process of concept attainment does not necessarily mean that a persons or system's resulting set of criterial attributes correspond directly to some widely held set of defining Attributes. In particular situations an engineer may actively choose to model a system at a level of detail that he knows to be inaccurate for whatever reasons, eg time/resource constraints, as a precursor to more detailed modelling, etc
Experts in a particular field may well employ criterial Attributes that go beyond or knowingly contradict currently accepted definitions. What is required for intelligent problem solving behaviour is that the engineer or expert be explicitly aware of the relevant defining Attributes.

Non-criterial Attributes are those which do not produce a change in the likelihood of an object being categorised in a certain way when their values change, eg the colour of a person's eyes, the placement of baffles in a heat exchanger, etc. In most cases sorting Attributes into criterial and non-criterial ones is not a clear cut decision but one that results over an extended period of time dependent on the use of the parent Concept and the learning or training set of exemplars encountered. (For example, observing the colour of someone's eyes to be luminous green may well call into question the validity of categorising them as human! Bruner et al therefore refer to Attributes, both criterial and defining, as having a degree of criteriality or definingness. An Attribute of maximum criteriality will result in an object being categorised as "X" when that attribute is observed as being present, or having a permissible value.

The use of criterial Attributes is interesting in that it raises the question of the inferential value or validity of both Attributes and category placements. Much of the content of rules in production rule systems, see chapter 3, is concerned with, what Bruner et al refer to as, the causal relationship between criterial or defining Attributes and equivalence category membership, eg, if the stream components are a liquid phase then separation by distillation may be possible".

Unlike most production rule representation systems, however, it is not necessary to attempt to associate some certainty factor with such a statement. All that is initially required is to recognise that there is some causal relationship between an Attribute and some parent Concept, or equivalence category, and whether the Attribute is being used in a criterial or defining way. Handling uncertainty is then concerned with explicitly knowing one's belief in the inferential validity of an Attribute, eg whether it is only usually partially valid, and how this will affect category placement.
The important point here from a representation point of view is that there is a need for an explicit representation of the causal relationship between Attributes and their parent Concepts and the way in which Attributes are used in category placement, eg in a criterial or non-criterial way, the validity of the cues etc.

It is worth noting here certain cues, ie Attributes, do not obtain a criterial status based purely on their "goodness" or validity for prediction. Bruner et al observe that in many instances the status or criteriality attached to certain Attributes is higher or goes well beyond that which would be permitted by their defining value or "ecological" validity. There are two areas in which this has important implications:

1. Experienced engineers or technical experts may unwittingly overstate the importance attached to certain Attributes when attempting to verbalise problem solving heuristics. When new information is learned engineer's will often change their point of view with little trouble. Any radical changes in certainty factors, on the other hand, would cause major problems when updating, say, a rule based system.
2. In problem solving situations the most important criteria for deciding how much reliance will be placed on a certain Attribute will depend on the internal objectives of the person's categorising decision and the perceived benefits that will result thereof. In a rule based system working with uncertainty factors this would effectively mean that confidence factors were variable and problem dependent. In such a system it would be very difficult to carry information consistently through a sequence of problems since important contextual information has been lost by reducing uncertainties to single valued, point measures.

In both cases an explicit representation of the relationship between Attributes and Concepts, as mentioned above, is needed to provide an adequate model of the cognitive processes involved.

5.2.3.2 Attribute Combinations and Basic Equivalence Category Types

It is rarely the case that the question of equivalence category membership can be decided on the basis of a single Attribute. More typically, category membership will depend on a person's response to the values of a number of Attributes taken together. The inter-relationships between these Attributes and the way they are combined together to form working definitions of category membership are of considerable importance. Bruner et al studied three basic category types, namely conjunctive,

disjunctive and relational types, each involving different ways of combining Attributes. The basic features of each Concept type are simply defined as follows:

1. Conjunctive Category - "are defined by the joint presence of the appropriate values of several attributes". Due to the difference between criterial, non-criterial and defining attributes this is a weaker definition than one requiring logically necessary attributes related by the AND operator.
2. Disjunctive Category - As with conjunctive categories, this is a weak form of logical necessity of certain suitably valued Attributes grouped by an OR operator.
3. Relational Category - "are defined by a specifiable relationship between defining attributes". Relational Concepts are usually higher order constructs that relate constituent Concepts and Attributes in some defined way. The rules of inference associated with such concepts require that values of different Attributes bear a specified relation to each other. A simple example is the notion of an equation and the relation between its left and right hand sides. Simple engineering examples are Maxwell type definitions of various thermodynamic Concepts.

The essential difference to note here between the use of the terms conjunctive, disjunctive, and relational Concepts and logic based representations of equivalent objects or frames is the way in which both the nature and relationship of Attributes to their parent Concept alter the evaluation of the equivalent "truth" conditions. For example, all Attributes in a conjunctive category do not have the same status as regards allowing category membership. This is not the case in a simple frame or rule based system using a logical AND, or OR, operator to combine and test slot values. As discussed previously, chapter 4, most frame based systems implicitly assume that the set of describing slots are taken to be conjuncts in the overall frame definition. This is too simplistic a model to capture many of the subtleties of conceptual categorisation as observed by Brunner et al.

As with much of conceptual categorisation, there does not appear to be any unique or "correct" way of combining Attributes in working definitions of equivalence category membership in that effectively equivalent but different combinations of Attributes can exist. Bruner et al observed, however, that the particular combination pattern used can radically alter the way in which learnt Concepts are applied to new problems or used to take account of new information.

This is an extremely important point as regards the development of knowledge based engineering systems. It is my belief that much of an engineer's ability lies in his ability to abstract problems in a way that will allow basic, well learned principles to be applied in the analysis and productive solution of those problems. The application of principles in the current context precisely refers to the use of learned concepts. It seems clear then, given the work of Bruner et al, that the internal formulation of a Concept will significantly impact on the competent use, or otherwise, of that Concept,

ie whether the Concept has been well "grasped" or not. This observation confirms the inituitive conclusion of chapter 3 that not only are some aspects of simple representation schemes inadequate, eg using production rules to express algorithms while implicitly relying on a particular rule firing order, but that in the context of my interpretation of the ideas involved in conceptual categorisation they are also in some sense "wrong". This is because many system representations of some Concept do not allow it to be conveniently used in new or novel situations, nor do they allow Concept definitions to be easily refined in the light of new, possibly contradictory, information or data.
It is not surprising then that rule sets, for example, are notoriously difficult to expand and update in a consistent way as there is no simple way of grasping the "intent" of each rule without bringing into question the operation of the overall system.

5.2.4 The Process of Concept Attainment

Having described the basic aspects of conceptual categorisation there remains the essential questions, from a representational point of view, of how equivalence categories are established or learned and how they are subsequently used in problem solving situations. The work of Bruner et al and subsequent related research in cognitive science indicates that there are no simple answers to these questions. A particularly complicated question that raises deep philosophical arguments, and is one that is still far from being resolved, is just exactly what the basic nature of certain category types involves and how both category types and Attributes are related to linguistic labels, ie a theory of word meaning in cognitive terms. It is now generally accepted that important parts of human thought are highly perceptual, often pre-verbal in nature, and not readily open to introspection or verbalisation. If this is indeed the case then accurate models of categorisation processes will remain long term research goals for some time to come.

What is important in immediate research terms is my belief that many aspects of handling functional and formal equivalence categories can, in fact, be modelled provided the scope of study is restricted to abstract forms of conceptual entities, eg the abstract description of distillation separations as often found in text books. It must be emphasised however that much of the "appeal" of such descriptions relies on more primitive or perceptual notions of linguistic labels such as force, pressure, liquid, vessel etc. This level of "understanding" will be outside the capabilities of any reasoning techniques proposed here in much the same way as there is no attempt to reason about identity and affective equivalence categories. (It is worth observing that when Concepts are introduced that do not immediately relate to such common sense,

real life interpretations then considerable difficulty in their use and comprehension can be observed. A clear example of this sort of phenomenon in process engineering is the introduction of concepts like vapour pressure and entropy. Entropy is a particularly elusive concept that, I suspect, many practising engineers will have little feel for).

The work of Brunner et al is still important, therefore, in that although no detailed models of the process of Concept attainment and use are given, a discussion of the most important factors and conditions affecting Concept attainment behaviour is provided. These ideas are briefly described below.

5.2.4.1 Factors Affecting Concept Attainment

There are two broad aspects to the process of Concept attainment, ie learning or grasping Concepts, that involve different strategies or styles of behaviour. The first is that of initial concept formation. This specifically refers to the "process of finding predictive defining attributes that distinguish exemplars from non-exemplars of the class one seeks to discriminate". This can be thought of as the first phase of attaining a concept that establishes, for example, what is it about a pump that "makes it" a pump. This stage of learning typically involves a good deal of social interaction, taught instruction etc. Once a Concept has been formed the process of Concept attainment involves gradually refining the use of Attributes in more effective ways in "problem" solving situations. The essential point that Bruner et al make is that Concept attainment involves an extended sequence of decisions. The working definition of a Concept at any instant is wholly dependent on the previous history or pattern of decisions involved in the acquisition, retention and use of information that served to meet problem specific objectives in forming the Concept at the time. That is, a Concept is not a one-off definition. It has a dynamic nature that depends on many factors including previous exemplars encountered, previous use of earlier "definitions", currently perceived needs and objectives etc. Bruner et al group the factors affecting concept attainment into the following five areas:

1. Definition of the Task - In problem solving scenarios an explicit awareness of the task definition and one's objectives is central to the process of either Concept attainment or use. Typical questions that might be asked include "what is the purpose of the task?", "What does a person think he/she is supposed to be achieving?", etc. Making a problem task, associated constraints and strategy formulation all explicit is an important part of the "intelligent" use of abstract Concepts.
2. The Nature of Instances Encountered - The nature, number, and order in which instances, ie exemplars of a Concept, are encountered can significantly affect the formation or refinement of a Concept. Some specific questions to consider are "How many properties have to be considered and how many of them are defining Attributes?", "Are the learning instances encountered at random or is there any control over the instances selected?", and "Will a set of training

instances fully define a Concept?".
The expected nature of a Concept, eg conjunctive or relational, strongly influences the choice of adopting one strategy over another in Concept formation or refinement. The use of analogies is a powerful strategy, both in Concept formation and use, ie in problem solving, and allows a person to behave in this anticipatory way.

3. The Nature of Validation - ie the ability to check, confirm or discard hypotheses, plays an essential part in the mechanisms of Concept attainment. This feature is seen as particulary important in the context of knowledge based systems responding or acting in "intelligent" ways and is expanded in below.
4. The Consequences of Specific Categorisations - what is the cost of categorising a specific instance wrongly or the gain to be made from a correct categorisation? This sort of consideration is very important in forming the actual strategy to be used in a particular situation. This is just as important in the context of a knowledge based system that is to assist in the real-life solution of engineering problems as it is to the engineer himself. For example, what merits a good solution under a particular set of circumstances, merely a better than chance guess or a "correct, elegant solution"?
   The process of making recommendations or providing solutions is closely tied to how much one wants to "understand" a solution or solution technique. An engineer, for instance, may quite happily use Gear's technique for solving sets of stiff differential equations in the solution of some problem without knowing the details of the solution technique. This may be acceptable behaviour as long as the technique appears to work, possibly confirmed by an alternative solution method. The benefits to be gained from this assumed categorisation are considerable but significant problems will arise if the solution technique breaks down and the engineer has no real understanding of the Concept of stiff differential equations.
   The general question of the "cost" of making recommendations is important in the design and practical use of a knowledge based system. The "cost" to a system that suggests, for example, problem formulations that would require enormous resources to solve is likely to be that the system will be abandoned. Taking into account the potential consequences of recommendations in particular situations is a problem that has been given little attention in knowledge based system research. In their original work Bruner et al tentatively suggest the use of Payoff matrices using quantatative estimates of expected value to make choices between alternative decisions. This idea is based on a classical decision theory approach. The whole question of the use of numerical estimates of likliehood was discussed and criticised in chapter 4. Brunner et al suggest themselves that this is unlikely to be a general mechanism and is therefore not discussed further here.
5. The Nature of Imposed Restrictions - the precise nature of local, situation dependent restrictions strongly affects the way in which a Concept is attained or used. For example, is it possible to keep a complete record of instances encountered? Is there a price, eg computing time, attached to the testing of instances? Is there a time constraint requiring speed decisions? Such constraints have a strong effect on engineer's problem solving strategies and the way they attempt to learn new ideas.

The combined effect of these five areas is the formation of what is called a strategy. A strategy refers to the pattern of decisions involved in the acquisition, retention and use of information that serves to meet certain objectives. By noting regularities in these patterns of decision Bruner et al identified various idealised strategies for dealing with Concept formation for conjunctive, disjunctive and relational categories.

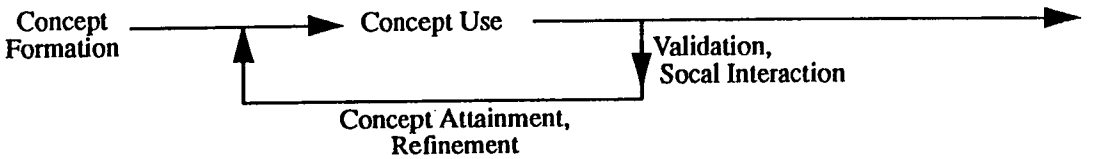
It should be noted that, due to the factors listed above, the strategies employed by people are not "fixed things". One of the most creative aspects of Concept attainment behaviour is the fact that the pattern of decisions does indeed reflect the demands of the situation, eg the nature of the Concept being sought, the imposed restrictions, possible consequences that may occur, etc.

5.2.4.2 An Overall Framework for Concept Attainment

On first sight the variety of factors that influence the process of Concept attainment would seem to indicate that modelling such a process would be very difficult. In an ultimate sense this is undoubtedly true. On the other hand, just as the scope of categorisation has been restricted here to formal equivalence categories and limited forms of functional categories, so too can the scope of the complexity of Concept attainment be restricted to allow a useful start to be made on the problem.

The overall framework used here is very simple and shown in Figure 5.2 below.

Figure 5.2: A Framework for Concept Use and Attainment



There are two key issues:

1. the use of Concepts and their attainment is part of an ongoing process. Concepts are not isolated, static definitions.
2. the framework makes explicit the use of the factors, conditions etc, that influence the interpretation and use of Concepts. These factors have hitherto been either ignored or treated in ad hoc ways in other knowledge based systems.

It is not until such a framework is developed, which takes into account the relevant problem/situation complexities, can one expect to see knowledge based systems behave in "intelligent ways" that satisfy the requirements for a process engineering design environment.

Details of an attempt to formalise such a model are given later in chapter 6.

5.2.4.3 Validation and the Learning Process

An important part of the learning process is the nature and frequency of the opportunity that exists for validating decisions. In everyday life it is not very common, nor desirable, to confirm or expect to be able to confirm every categorising decision that is made. Indeed, an A.I. system that asked whether everything it did was

correct or not would be impossible to use. More typically a person has to work with reduced opportunity for validation. A good example of this is on the field of hazard and safety analysis in which certain Concepts used in plant design will hopefully not result in validation in terms of negative responses to decisions, ie safety related accidents occur. Until such an accident does occur and safety procedures have to be tightened, an engineer has to cope with the absence of positive validation and continue in the belief that the initial ideas will continue to apply in new but conceptually similar instances.
(NB Tightening of safety procedures in this example corresponds exactly to what is labelled the "refinement" step in Figure 5.2 above).

The work of Bruner et al identified that validation of decisions plays a part in four aspects of the overall learning process. These points are reproduced here because of their long term importance in a representation system.

1. Validation evokes problem solving behaviour - the opportunity to check categorisation decisions against an ultimate, external criterion has the effect of making one less casual towards error. That is, when validation is possible then errors in decision making can be observed and studied. This is necessary to stimulate learning if errors in decisions lead to undesirable consequences for the engineer or system involved. This point is somewhat obvious but non-trivial to model since a knowledge of such consequences will set the scene for future behaviour.
2. Validation provides the means whereby learning proceeds - checking intermediate hypotheses is an important part of the process of Concept attainment and the formulation of problem solving strategies. This point re-enforces the fact that the successful use and attainment of Concepts is a strongly social process that should be reflected in the design of an A.I. system.
3. Validation provides a basis for the regulation of problem solving behaviour. This is the means by which engineers develop heuristics, or keep a score, on the use of certain Concepts based on the outcomes of past decisions about similar instances.
4. Validation regulates one's level of aspiration - following on from the previous point, repeated validation provides a basis for self assessment. This strongly influences the limits which engineers set themselves in future tasks. This is an important problem for systems which have to make "reasonable" recommendations and then subsequently act on those recommendations since it affects the types of problem solving strategy that will be considered.

A general observation on the reduced frequency of validation is that it results in a move away from a problem solving behaviour towards some form of all-or- nothing behaviour. If, as a result, a system becomes tolerant of errors then a certain margin of error will come to be accepted as inevitable. Problem solving behaviour likewise decreases when scores cannot be kept of the outcomes of past events or when hypotheses cannot be tested. In such situations systems will exhibit a strong bias in decision making and it is important to know why such bias exists if it has to be subsequently corrected.

5.2.5 Further Aspects of Atrributes : Typical and Generic Instances

The final topic of discussion regarding conceptual categorisation concerns more subtle forms of Attribute use in problem solving situations. Most of these ideas, however, are inevitably applicable to the concept formation/attainment process since the two processes are inextricably related to each other. When a category has been learned the problem of how to use it or refine it given new examples then arises. An important factor in any further behaviour is the number of Attributes, defining or criterial, that affect categorisation. It is clear that if there are many defining Attributes then the process of scanning the values of the Attributes in order to reach inferential decisions may turn out to be a costly and time consuming activity. In such cases there will be a tendency to reduce the cognitive effort, especially when time pressures are operating, and certain Attributes will not be fully considered that otherwise would be under more leisurely conditions. This simplifying and often useful behaviour can be observed in two basic processes. The first and most obvious method involves the idea of Attribute reduction. An engineer will often tend to rely on the most criterial Attributes, ignoring less criterial Attributes, regardless of whatever the basis of criteriality might be. This sort of Attribute reduction can happen in several ways. It may be that some Attribute value in the presence of other criterial Attributes has proved highly predictive in the past, or it may be that the Attributes used are those that are most immediately discriminable. The extent of attribute reduction will usually reflect the constraints imposed upon a person. In cases where certain outcomes have serious consequences then it is unlikely that much Attribute reduction will occur. A similar situation arises when the degree of belief in the validity of any of the Attributes is low. In most other situations, however, an Attribute reduction strategy may be acceptable.

A second method, of less immediate relevance to this thesis, is the development of so called "configurational attributes". This refers to the situation in which a frequently observed object or event takes on a configurational quality itself, classically referred to as a Gestalt quality in psychological literature. The interesting feature about Gestalten is that it appears it is no longer necessary to scan through the set of defining properties for categorisation purposes. Rather, the object or event becomes a property or seemingly irreducible attribute in its own right. This form of recognition appears to be highly perceptual in nature and is often the result of transforming formally defined Attributes or Concepts into more immediate perceptual cues. The technique is an advanced form of attribute reduction and introduces the useful trick of predicting or "filling in" Attribute values based on the presence of one or two highly criterial Attributes. This has obvious parallels with the use of default reasoning techniques in A.I., see chapter 4.

An important driving force behind the whole of the conceptual categorisation process is the aim of reducing the disorder and confusion of a system's environment and the effort needed to come to terms and rationalise about that environment. This is one reason why the orderliness of exemplar encounter and the problem of reducing cognitive effort are so important in determining the behaviour people adopt.

It is clear from the studies made by Bruner et al that a key feature of this process is adopting a focus or reference point from which one can proceed. Most of the effective learning strategies described by Bruner et al use some sort of initial focus and this is reflected in many teaching methods: start off with a base reference case and gradually proceed by removing various simplifying assumptions. From a problem solving point of view the important question is how order can be maintained after a Concept has been initially learnt. There are two useful mechanisms observable in categorising behaviour that are related to the use of Attribute reduction and configurational Attributes, namely the use of typical and generic instances. The formation of a typical instance of a category involves the phenomenon of an adaption level. A typical object or event is formed in a gradual way and attempts to account for or summarise all exemplars of a category that have been encountered so far. As such, it will use typical or average values for each of the defining attributes. Engineers frequently use rule of thumb guidelines of this sort. A good example is the use of quick sizing and costing estimates based on data averaged over many case studies. These calculations can then be used for the guidance of more detailed investigations. Generic instances are related to but not the same as typical instances. A generic instance of a Concept is a representation of a Concept in terms of idealised values, rather than average values, of the defining or criterial attributes. Further, a generic instance will be stripped of any noisy, non- defining Attributes, ie Attributes that change value from instance to instance but which are non-defining. The function of a generic instance appears to be somewhat different to that of a typical instance. In problem solving situations a generic instance is often used as a base case search model when considering what classes of objects, techniques, etc. would be relevant to the solution of as yet unsolved problems. As such they provide an important initial problem solving focus which provides direction for more detailed forms of reasoning. This in turn may involve a specific category for which a typical instance is available to allow further progress to be made.

A stunning example of this idea is the "fiction" of an ideal gas. An ideal gas is clearly an idealised representation rather than a typical one. Nevertheless, this simple representation of a gas and its associated describing equations has had a profound effect on the development of all real gas models and equations of state proposed to date. This particular generic instance is interesting in that the ideal "model" has

physical relevance in the limiting case of atmospheric pressure for many gases.

The use of strategies that involve typical and generic instances can greatly enhance one's problem solving capabilities especially in cases where "creative insight" is required, eg novel plant synthesis problems. These ideas, when combined with the more general use of (reduced) criterial attributes, appear to provide simple but powerful forms of reasoning that could be applied in a knowledge based system.

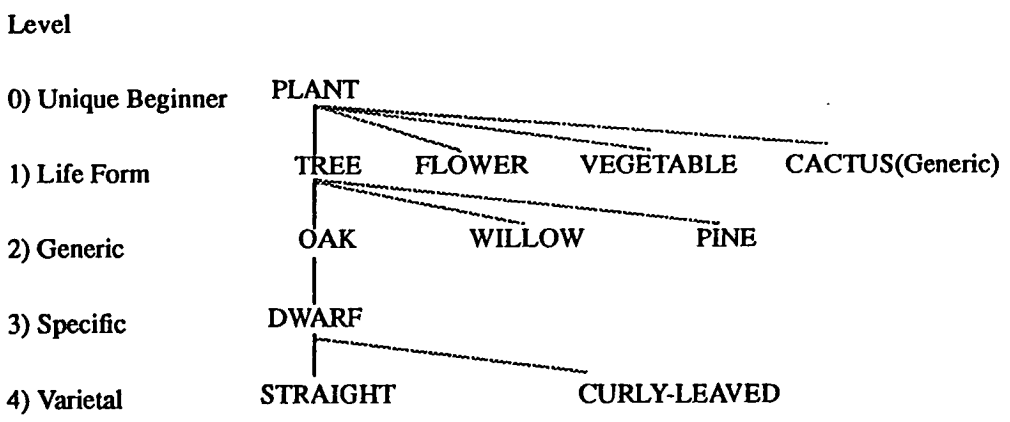
Section 5.3.2 provides a brief discussion on the difference between the ideas described here, including the notion of prototypes, and the confused situation that has arisen in frame based systems and object oriented programming in general.

5.3 Prototypes, Family Resemblances and Categorical Frameworks

It was stated at the beginning of this chapter that the study of Concepts and mental modelling would focus on three areas, namely Conceptual Categorisation, the use of prototypes and mental models. This section introduces the idea of a prototype as a basis for conceptualisation. The work is described here as an extension of my interpretation, as described above, of Bruner, Goodnow and Austin's (1956) work on conceptual categorisation. The ideas involved are obviously related but little mention is made of Bruner et al's work in the later work on prototypes.

It seems clear, given the considerable amount of research in psychology, that in certain areas of vocabulary many labelled categories or Concepts are related by some superordinate taxonomic structure. The nature of these structures was studied in a series of papers by Berlin, as summarised in Berlin (1978). The work suggests that taxonomies usually consist of less than five, sometimes six, layers and that many are of the same general form. An example is shown in Figure 5.3 below.

Figure 5.3: A Feature Based Taxonomy



(Despite the deceptive simplicity of this hierarchy it should be noted that it is far from adequate in terms of the ideas on conceptual categorisation as it provides no information on the use and nature of Attributes used to "justify" the labels or category types. This is essential information for maintaining and using the categories).

What is useful about Berlin's analysis is the use of levels to differentiate different forms of category. At the highest level (0) occur the broadest divisions of the observed world. The next level (1) consists of "life forms" which are typically few in number and never terminal, ie they always have sub- categories. In relative terms, the most numerous type of category exists at level (2), termed "Generic" by Berlin. (NB The use of Generic should not be confused with generic instances as described in section 5.2 earlier. Generic, as used here, refers to a Concept or Category not an idealised instance). Occasionally a generic form appears at level (1), as in the case of CACTUS in Figure 5.3. This usually happens when the Concept is particularly distinctive in some way.

The most important observation of Berlin was that one level, the generic level, contains proportionately many more members than other levels. In addition, they are often of a special significance in that they are typically learned earlier by children than the categories at other levels. In Bruner et al's terms they are both perceptually and functionally distinct from other Concepts. Berlin provides no evidence nor reasoned arguments, but intuitively suggests that categories are formed in this way to provide "maximum use of information".

In an attempt to provide a more detailed account, Rosch produced a series of papers (Rosch (1973), Rosch and Mervis (1975), Rosch et al (1976), Rosch (1978)) on the exact relationships between a category and its members. In particular, Rosch introduced the notion of a "prototype", see below. This idea has been widely referred

to in the AI literature to justify the use of frames or objects in various guises. It is clear from the literature that this term is often used in confused or misleading ways. The discussion below briefly describes the precise nature of a prototype and how it relates to the notions of a Concept, typical instance, generic instance, etc.

The simplest way to introduce prototypes is by means of an example. In a series of experiments Rosch showed that people could make consistent judgements about how good an example of a category a particular example was. For example, "apple" was considered a better example of the category "fruit" than "olive" was, car is a better example of a vehicle than skateboard, shell and tube is a better example of a heat exchanger than spiral, etc. In addition, the truth of statements such as "an apple is a fruit" are confirmed significantly faster than statements involving poorer examples, eg "an olive is a fruit". In particular, children make more errors of judgement on these so called, peripheral sentences involving poor examples, of a category. Rosch labelled the good examples, or central instances, of a category prototypes.

The work of Rosch et al raises, or emphasises, three important points that were implicit in the work of Bruner et al on conceptual organisation. The first point is that categories have some form of "internal structure". This, of course, merely confirms the earlier suggestion that the use of Attributes and their relation to one another is important information in the working definition of a Concept. Rosch's work visibly demonstrated that a simple model of word meaning based on sets of necessary properties, ie a conjunction of a series of properties all of which must be possessed by an instance in order to satisfy category membership, is psychologically implausible. If it were, there should be no difference in response times to judgements of category membership for all members of a given category. This is consistently observed not to be true.

The second point, which is implied by the first, is that the assumption that all members of a given category are of the same status, or "equidistant from the category" as Rosch puts it, is not valid. Simply stating that the category membership relationship is one of proper inclusion is not sufficient for many life form categories and their generic members, ie levels (1) and (2) in Berlin's nomenclature. It is insufficient for problem solving or learning purposes because it does not take into account the internal category structure.

The third aspect of Rosch's work is that it suggests a reason why taxonomies, as observed by Berlin, are structured in the way they are. Rosch suggests that the best examples of a category, ie the prototypes, will be "maximally different from other categories on the same level of linguistic contrast". If this is a general principle then it suggests that all formed categories will tend to be maximally discriminable. Further, in order to best exploit perceptual, functional and behavioural differences, the best

examples of a category would be those that did not arise at all in other categories, or were not prominent in other categories at the same level of linguistic contrast. It also follows that the level at which functional, perceptual, etc, differences are most pronounced should correspond to the level on which most categories are formed. This is in agreement with Berlin's observation that the generic level contains proportionately many more members than other levels.

The notion of maximum discriminability is usually related to that of cue validity. Cue validity was discussed in section 5.2.3 in the context of the selection of criterial Attributes. A somewhat oversimplified definition of cue validity is given in Rosch and Mervis (1975) for Attributes of all types as:

> the frequency of a cue being associated with the category divided by the total frequency of that cue over all relevant categories

If this definition was used in the formation of maximally discriminable categories then a cue of high validity for a bird, for example, would be having feathers since this cue is not important in many other categories. Having "legs", however, would not be a cue of high validity since it is one shared with many other Concepts, eg most animals, insects, tables, chairs, etc. Using Bruner et al's terminology, "having feathers" would be described as a highly criterial, probably defining, attribute of the Concept bird. Somewhat confusingly, Rosch goes on to suggest that the "cue validity" for an entire category is the sum of the cue validities for each of its attributes. I find two objections to the use of the term "cue validity" in this way:

1. What is meant by cue validity in terms of an overall category as a whole rather than any one of its constituent Attributes, unless we are referring to some Gestalt phenomenon?
2. The use of a simple summation of cue validities over all Attributes, including non-criterial Attributes since Rosch does not make Bruner et al's finer distinctions between Attribute types, is of questionable value as a measure. For example, a Concept with many non-criterial Attributes could result in a significant proportion of the summed value being contributed by the non-criterial Attributes.

A term such as information content or measure, preferably biased toward criterial Attributes, would seem more appropriate especially given its intended use. Rosch et al (1976, p383) state that "categories within taxonomies of concrete objects are structured such that there is generally one level of abstraction at which the most basic category cuts can be made. In general, the basic level of abstraction in a taxonomy is the level at which categories carry the most information, possess the highest cue validity, and are, thus, the most differentiated from one another.

In other words this information measure can be used to help explain the form of taxonomies at the life form and generic levels and the relative number of categories

involved.

In comparison with Berlin's work, Rosch suggested that taxonomies are simply organised at three levels. The topmost level is termed a "superordinate" category, eg animal, which has a number of members at the "basic" level, eg dog, cat, etc, which in turn include a number of "subordinate" level members, eg labrador, spaniel, etc. Given the discussion above, Rosch et al claim that categories at the basic level, similar to Berlin's generic level, will be the most informative. Their experimental findings would suggest that this is generally true. This is an important point in the context of the development of reasoning systems to work with Concepts within the overall framework shown in Figure 5.2.

Rosch (1978) suggests that the formation of categories is based on the principle of cognitive economy. This basically states that categories will be formed so as to be maximally informative such that "cue validity" is maximised. This statement must be qualified because as it stands it implies that a large number of singleton categories would be formed, each individuated on the basis of a single attribute, so as to give categories with the highest cue validity. Categories are only formed where the Attributes are perceived to reliably co-occur in particular groupings in the observed world. This is in agreement with Bruner et al's claim that many formal equivalence categories arise from more basic perceptual ones, the latter being necessary to identify reliable groupings of Attributes.

Rosch and Mervis (1975) suggest that the principle of cognitive economy can be extended to help explain the formation of the internal structure of categories in as far as the use of prototypes is concerned. For this purpose Rosch appeals to the notion of "family resemblances". The term family resemblances is taken from the well-known philosophical work of Wittgenstein, Philosophical Investigations (1953), in which he argues against the idea that Concepts can be defined in terms of necessary and sufficient conditions. As an illustrative example, Wittgenstein considered the Concept or category of games, eg chess, football, card games, patience, a child playing with a ball, etc. The following extracts summarise the main points:

> Consider for example the proceedings that we call 'games'. I mean board-games, card-games, ball-games, Olympic games and so on. What is common to them all? - Do not say: 'There must be something common, or they would not be called games' - but look and see whether there is anything common to all. - For if you look at them you will not see something that is common to them all but similarities, relationships, and a whole series of them at that.
>
> ... And the result of this examination is: we see a complicated network of similarities overlapping and criss-crossing: sometimes overall similarities, sometimes similarities of detail.

> I can think of no better expression to characterise these similarities than 'family resemblances' ... (Wittgenstein, 1953, p31-32).

The basic point Wittgenstein is making is that there are few, if any, properties shared by all games. Instead, one game shares some features with the next and that game in turn shares some features, possibly different ones, with the next game, and so on. As a result some pairs of games might not share any features with each other although they would at least share one feature with another game.

Rosch used these ideas to define a measure of family resemblance similar to that of cue validity, see earlier. Each Attribute of a member of a category, as listed by the subjects taking part in the experiments, can be given a score. That score is the number of members of the category which possess the Attribute. For each member of a category the degree of family resemblance is simply taken to be the summed total of the scores associated with each of its Attributes. Rosch tried to show how judgements of prototypicality should correlate with the degree of family resemblance since prototypes would be those members which a) shared most properties with other members of the category, and b) shared fewest properties with members of contrasting categories. This was found to be the case in several of Rosch's studies.

In summary then, Rosch states that

> categories form to maximise the information rich clusters of attributes in the environment and, thus, the cue validity of the attributes of categories; when prototypes of categories form by means of the principle of family resemblances, they maximise such clusters and such cue validity still further within categories. (Rosch and Mervis, 1975, p602).

It appears then that the importance of the basic level in a taxonomic structure and the salience of prototypes within categories can be part explained by the simple principle of cognitive economy. This is an important result in the context of the development and use of categories in a knowledge based representation system.

5.3.1 Further Discussion and Criticisms

Having introduced the basic ideas of prototypes and family resemblances it is worth briefly discussing some criticisms of Rosch's work and how these ideas relate to earlier descriptions of typical and generic instances, attribute reduction, etc in conceptual categorisation.

A well informed analysis of Rosch's work can be found in Pulman (1983). This work is written from a linguist's point of view and is concerned with the cognitive/philosophical adequacy of theories of word meanings or concepts. As such it is a good introduction to many of the technical linguistic problems that have to be

addressed in knowledge based systems.

Pulman identifies two basic problems that prevent direct use of Rosch's findings:

1. The exact status of prototypes and the nature of their Attributes in a theory of conceptual categorisation is unclear.
2. Rosch's account of category formation is not well defined.

Pulman's arguments are summarised below but have been re-interpreted given the discussion on types of conceptual categories and structuring of Attributes earlier. Pulman makes no reference to the work of Bruner et al in his work.

In Rosch's studies it was assumed the Attributes which were used to calculate cue validities were perceptual or functional givens and independent of the category to which they belonged. It is clear, as Rosch (1978, page 41) herself points out, that this is not the case. Pulman points out three such examples:

1. An "attribute" leg, for example, is perceptually quite different in Concepts such as man, bird, insect, table, chair, etc, although being labelled by the same word.
2. Rosch makes use of complex, so called, "function attributes" such as "sit-on-able-ness" or "can be sat on" in defining types of seats. The use of such "attributes" however seems to require prior classification of the object as some sort of chair since many other items are just as sit-on-able as chairs, eg tables, boulders, suitcases, walls, etc.
3. Many of the "attributes" are "relational attributes" such as large, small, etc. These attributes must also involve prior classification of objects since large, for example, in the context of insects is not the same as large when describing animals.

It is clear from the examples that the point Pulman is making is that many attributes are "not identifiable prior to or independently of recognition of the category they belong to", (page 97). Before discussing the consequences of this observation it should be noted that both Rosch and Pulman use the term "attribute" in an extremely loose or ill-defined way compared to my interpretation of that term in conceptual categorisation. For example, nowhere in Rosch's or Pulman's discussions are references made to Bruner et al's basic use of three forms of category, namely affective, functional and formal equivalence categories. As a consequence the nature of Rosch's "functional attributes" becomes very confused and/or vague. In a conceptual categorisation framework these "attributes", eg sit-on-able-ness, would be better described as functional concepts, eg things to sit on, which in turn permits discrimination between man-made artefacts, eg chairs and naturally occurring categories, eg boulders, to be handled in a general way. Further, Bruner et al specifically discuss the close relationships between functional concepts, eg things to sit on, and formal equivalence categories, eg chairs, and how examples of the latter category often arise from the former. It is important then to understand the subtleties between the different types of Concept and their use in order to avoid the potential

confusion over the use of Attributes arising from "naive" descriptions of words, Concepts, etc.
A second point worth noting is that no use is made of Bruner et al's idea that Concepts have internal structure which in part comprises of combinations of Attributes either conjunctive, disjunctive or relational ways. Further, no explicit distinction is made between the use of defining, criterial and non-criterial Attributes. This is a serious omission since such considerations must play a central role in both category recognition and category formation processes.

The omission of functional Concepts and relational type formal equivalence categories leads Pulman to describe "functional attributes" as two, or more, place predicates, ie relations, between ourselves and some other object or objects. He describes "attributes" with values such as small, large, etc, as two place relations between individual objects and other objects or some implied standard of comparison. It is clear that relational "attributes" in this sense are not independent of the categories involved, which is Pulman's point, since they can only be interpreted once categories have been formed or recognised.
Whilst in general agreement with Pulman on this point I object to his use of "relational attributes" due to the confusion they would introduce in any representation system. The deceptive use of comparative words such as small, large, high, etc, in an absolute or adjectival way, eg X is small, might be naively taken to mean that they can appear as Attribute values in the same way, for example, as someone's age. This is not the case as Pulman recognises, there being some implied relationship with another object or some "typical" value appropriate to the category. In my interpretation of Bruner et al's ideas these sort of phenomena would be properly handled in a general way by making use of relational type Concepts to express this comparison. The linguistic description of this information is obviously important in communicating such ideas to others. Extreme care must be taken, however, in translating such phrases into possible Concept descriptions if the confused situation that has arisen with frame/object descriptions is to be avoided. In particular, different types of Attribute cannot all be lumped together and treated in the same way, as is the case in simplistic frame based systems.

Despite these reservations regarding terminology, Pulman has two important points to make. The first is that both the processes of category recognition and category formation are logically incoherent or circular if they proceed via computation on "attributes" that require prior recognition of the category itself. Rosch recognised this problem and argued that her ideas on prototypes and family resemblances were not concerned with a particular processing model (1978, page 28). In particular, Rosch

makes the following points:

1. The ideas or set of formal principles concerned with the structure of category formation are not tied to any particular processing model.
2. Formation of categories refers to their formation in the culture or society.
3. The process by which attributes are individuated is the same as the process by which basic categories are formed.

The first point is presumably intended by Rosch to ensure that the principle of cognitive economy is not taken directly as a theory of category formation or recognition in order to avoid the problems of circulating. Rosch's third point is not much help either since if attributes are themselves categories then how are these categories formed? Once again the argument results in an infinite regress since the latter categories are presumably themselves formed from attributes, and so on.

The second point, which appeals to the notion of category formation within a culture - similar to Bruner et al's idea of socially accepted defining attributes in formal equivalence categories, does not break the circular argument either. This is because it still depends on the assumption that "attributes" can be individuated independently of their categories which, as indicated earlier, cannot always be the case.

In order to break out of this circularity Pulman presents the following argument which arrives at essentially the same conclusions as Bruner, Goodnow and Austin made regarding the invented status of Concepts and Attributes. At some point categories must be defined in terms of Attributes which are perceptual "givens" in the sense that they have no further conceptual description. This seems quite plausible in certain low level categories such as the Attributes involved in the recognition of physical objects in terms of shape, colour, etc. However, this argument cannot be carried to all types of category. Many categories involve relational, functional or behavioural "attributes" (ie dogs bark, birds fly, etc) that do not have a direct relation to such perceptual basis. Any claim that they do seems either false or empty because the transition process to lower level perceptual Attributes of which they might be comprised is largely inaccessible and, therefore, untestable.
In order to treat these functional and behavioural "attributes" it is necessary to recognise that their salience is taken for granted. That is, to say that some object or "attribute" is salient is simply to recognise that it plays a recurrent, identifiable role in daily transactions. This means that for many purposes it is possible to treat these "attributes" as basic in nature, but not primitive, in the sense that they could be broken down into more basic elements if necessary. This implies that certain categories or "attributes" have to be taken for granted or as givens prior to the formation of other categories.

If this is correct then the conclusion Pulman reaches is that Rosch's findings do not constitute a complete explanation for category formation or recognition. Rather, the use of cue validity as an explanation for the formation of category structure is useful in that it helps to identify the range of possible individuals that will be named as prototypes in response to personal/social preferences.

Pulman's discussions regarding the status and nature of "attributes" in Rosch's prototypes effectively recognises therefore the constructive or invented nature of categories that Bruner et al had already identified. As noted earlier Bruner et al suggest that the process of equivalence categorisation essentially involves "coding and recoding processes employed by organisms who have past histories and present requirements to be met", (p7, 1956). In the context of the present discussion, the requirement is to provide a label to denote some perceptually identifiable attribute that can be taken to be a perceptual given until more detailed analysis requires that its definition be more fully explained. Bruner et al quote Stevens (1936) which sums up this situation well:

> Nowadays we concede that the purpose of science is to invent workable descriptions of the universe. Workable by whom? By us. We invent logical systems such as logic and mathematics whose terms are used to denote discriminable aspects of nature and with these systems we formulate descriptions of the world as we see it and according to our convenience. We work in this fashion because there is no other way for us to work.

In summary then, Pulman's criticisms of the exact status of prototypes and the nature of their Attributes are well founded. A much clearer discussion of the situation could be provided if use had been made of Bruner et al's distinction between category types and the status and combination of Attributes within those categories.

Pulman's second main criticism of Rosch's work on prototypes is that the account of category formation is not well defined (see page 369). This point is worth discussing briefly. Consider two categories A and B, whether at the superordinate, basic or subordinate level. Taking into account the notions of prototypes and family resemblances any member of A could be taken as a member of B, ie A is a "subclass" of B, if either

1. most of the Attributes of B are possessed by every A, where A is a subordinate level category and B is a basic level category, or
2. every member of A shares at least one Attribute with B or shares an Attribute with some other A' which itself shares an Attribute with B. In this case A and A' are basic level categories and B is a superordinate level category.

(Notice that this definition is quite different from that normally found in inheritance class hierarchies).

Rosch's account of category membership is derived from both common Attributes (at the subordinate and basic levels) and family resemblances (at the basic and superordinate levels). Categories, however, are supposed to be formed at all levels for the same reason, namely maximisation of cue validity. If this is the case then it is difficult to reconcile the existence of superordinate categories. This is because categories are supposed to arise from the fact that certain groupings of Attributes have a higher composite cue validity than others. If this simple interpretation were taken at face value then family resemblance type categories would never arise simply because they cannot be described in terms of co-occurring bundles of Attributes. The fact that such groups of shared Attributes do not exist is the whole point of family resemblance theory. A typical example that Rosch refers to is the category "furniture" for which her subjects could find few, if any, shared "attributes".

If Rosch's ideas are interpreted in a simple minded way then at best such categories would only contain a number of prototypes since there are no Attributes valid for the whole category. The only Attributes having a significant cue validity would be those common to more than one prototype but these would not be sufficient to generate or account for all the peripheral members of the category as well.

It appears then that Rosch's principle of category formation based on maximisation of cue validity is only tenable at the basic and subordinate levels and only partially so at the superordinate level. Further, this requires an extremely loose interpretation of the term "attribute" as described above.

Once again the whole phenomenon of structuring of categories can be more easily understood if the ideas of Bruner et al are taken into account. The whole emphasis of that work is placed upon the use of identity and equivalence categories, whether they were of the affective, functional or formal type. The main purpose of categorisation, both at the perpetual and conceptual level, is to

> render discriminately different things equivalent, to group the objects and events and people around us into classes, and to respond to them in terms of their class membership rather than their uniqueness, (Bruner et al, 1956, p1).

This statement can be interpreted to mean that the categorisation process will, by default, always attempt to work at the highest level of abstraction or equivalence suitable to the context of the problem at hand or prevailing situation. That is, the default level at which a Concept is considered is the level which avoids the need for consideration of any unnecessary detail. The obvious benefit of such an approach is that it greatly reduces the cognitive strain or effort involved in describing or handling problem information.

This constant use of Concept abstractions, in an upward sense to the level most appropriate to the problem description, should be contrasted with the use of object instances in inheritance hierarchies. In the latter case it is noticeable that the tendency is almost always to consider an object instance at its most specific level in the hierarchy, ie to take account of all inherited slots plus any local slots and consider the instance as a leaf node in the inheritance hierarchy. The whole emphasis is therefore on downward inheritance rather than upwards abstraction. This is an important point and one that is developed further in the following chapter.

In the context of Pulman's criticism of Rosch's account of category formation and the placement of categories within some superordinate structure, it seems clear that the desire to create useful equivalence categories in the Concept attainment process provides the necessary driving force to account for the formation of superordinate level categories based on family resemblances. It is proposed here that such categories are not formed based upon the maximisation of some simple measure of cue validity. Rather they result from a need or desire to communicate information at a higher but still meaningful level of abstraction. For instance, it is easier to refer to "the furniture in the house" in a loose, general way rather than to provide an extensive list of specific items to convey the same information. Indeed, if the context of the situation/conversation/problem etc is such that no detailed knowledge of what actual pieces of furniture are in a particular house then in many ways the abstraction "furniture" is a more appropriate term to use, eg it suggests uncertainty as to the actual contents of a house but indicates that typical or generic instances and/or prototype members are likely to be involved.

The role of family resemblances and equivalence categories in a Concept formation process is summarised later in section 5.3.2.

5.3.2 SUMMARY

The value of Rosch's work has been that:

1. It highlighted the use of prototypes as an important conceptual "tool". Considerable experimental evidence has shown that many prototypes are formed using a simple principle of maximisation of cue validity.
2. It confirms, along with the work of Berlin et al, that Concepts or categories appear to be arranged in superordinate structures of similar construction and limited depth. The work, however, does not take into account Bruner et al's finer distinctions between Concept and Attribute types.

The work of Pulman (1983) is of considerable interest to those who wish to gain a deeper insight into the use of prototypes, and Concepts in general, as a means of providing a theory of word meaning (most of the content of this discussion is,

however, outside the scope of this thesis). In his work Pulman proposed the study of prototypical verbs, along similar lines to Rosch's original work, but discovered the results to be much more difficult to summarise. Much research remains to be done in this area.

Pulman is also a good introduction to some of the philosophical arguments concerning theories of categorisation in general. Some of these arguments are touched upon in the following section.

5.4 MENTAL MODELS

The previous two sections introduced the ideas associated with Conceptual Categorisation and the use of prototypes and family resemblances. The discussion was mainly concerned with the principal types and features of mental Concepts. This involved identifying the differences between functional and formal equivalence categories as well as the difference in status between different types of Attribute and their mode of combination. Further it was suggested how the use of prototypes and family resemblances helped explain the structuring of Concepts into superordinate structures that provide part of the basis for a theory of word meaning. Other than the overall framework put forward for Concept formation, see Figure 5.2 , little has been said, however, on the exact details of how Concepts, generic and typical instances, the use of prototypes, etc, are utilised in reasoning processes. Indeed, from the point of view of development of a knowledge representation system, no mention has been made of the types of reasoning processes and possible mechanisms, other than equivalence category identification itself, that could exploit the use of Concepts to improve upon the limited capabilities of knowledge based systems as discussed in chapters 3-4.

In order to characterise more clearly a framework for reasoning with Concepts this section introduces the basic ideas of mental models as put forward by Johnson-Laird (1983). This work has widely been recognised as one of the most influential contributions to cognitive science in recent years. The content of the work is both broad ranging and extensive. In many places the technical content of the philosophical arguments is too detailed to be discussed adequately in an exploratory thesis on knowledge representation such as this. All that is presented here, therefore, is a simplified overview of the scope of Johnson-Laird's study. The background to those ideas concerning the formation and use of mental models that can be directly used in conjunction with those of Conceptual categorisation are described in some detail. It should be noted that Johnson-Laird's original work is not explicitly based on either the work of Bruner, Goodnow and Austin (1956) or Rosch and her co-workers, eg Rosch (1978).

5.4.1 Scope of Johnson-Laird's Study

Johnson-Laird's work on mental models is an ambitious attempt to understand and model human reasoning. In order to grasp the extent of this task consider the following questions that Johnson-Laird poses:-

- What happens when we understand a sentence? We are aware of understanding it, and still more aware of having failed to do so. Why can't we follow the mental

process of comprehension as we can follow the action of tying a shoelace?
- Why is it that the words that are most frequently used in conversation are precisely the ones that have many different meanings? In a rationally designed language, ambiguous words would be avoided and, if they had to be admitted, they would be used the least frequently.
- What is the meaning of words such as possible or time? You are familiar with them, and have no difficulty in using or understanding them in a typical sentence. But, now that you are asked what they mean, you are tongue-tied for a moment and then can only offer synonyms rather than analytical definitions.
- Why is it that when we consciously try to characterise a Concept, we try to do so in a cold-blooded, cut and dried fashion like lawyers defining a tort, whereas, as Wittgenstein pointed out, if we are prepared to look and see how ideas are used in daily life, we often find nothing so clear-cut, only indefinite and open ended Concepts?
- How is it possible for you to make a valid deduction even if you have not learned logic? Is there an innate mental logic? Or have you picked up rules of inference from other people? If so, how did *they* acquire them ?

In order to try and answer such questions requires an analysis of the fundamental aspects of human thought. Such a study is necessary for the further development of knowledge based systems since the issues involved lie at the heart of the description of "intelligent" reasoning processes.

Johnson-Laird's work covers five main areas, each relating to a specific topic:

1. The puzzle of mental logic, questions concerning thought.
2. The nature of mental representations and how they relate language to the world, questions concerning meaning.
3. The process by which the meanings of sentences are constructed from the meanings of their parts according to the grammatical relations between them, questions concerning grammar.
4. How the interpretation of discourse is built up from the meanings of sentences, questions concerning discourse.
5. The nature of intentionality and self-awareness, questions concerning consciousness.

The central idea that Johnson-Laird puts forward that ties together these areas is that humans construct mental models of the world and do so by invoking tacit mental processes. Further, these mental processes have a basic, recursive nature that is required to allow people to understand discourse, form mental models of both real and imaginary situations, and to reason by manipulating mental models. It is assumed that the core of the ability to explain, understand and make use of Concepts rests on having a "working model" of the phenomenon in the mind.

Johnson-Laird recognises that this idea is not a new one. He takes as a starting point the work of Craik (1943) who proposed that thinking involves the manipulation of internal representations of the world. In particular Craik suggested that at least three distinct reasoning processes must be involved:

1. A translation of external processes or forms into some internal representation.
2. The derivation of other internal representations by some sort of inference

process.
3. A retranslation of these internal representations into actions or at least a recognition of the correspondence between the internal representations and external events.

Points 1 and 3 are particularly important as they recognise the need to deal with referential phenomena in a satisfactory way. The relationship between mental models and external entities, ie the treatment of references, is essential both to the remainder of this introduction and the developments described in the next chapter.

5.4.1.1 The Doctrine of Functionalism

Johnson-Laird justifies many of his arguments by making appeal to the notion of a doctrine of functionalism. Put simply, this states the usefulness of a model is not enhanced by extending it beyond a certain level, ie by embodying more knowledge beyond a certain point. As Johnson-Laird points out, "like clocks, small-scale models of reality need neither be wholly accurate nor correspond completely with what they model in order to be useful", (page 3, Johnson-Laird 1983).

For example, a senior process design engineer, a process operator and an undergraduate engineering student will have different levels of understanding, ie detail of model, of the operation of a heat exchanger say. For many tasks however the level of detail in a process operator's model of a unit's operation is perfectly adequate. Even an expert in the field of heat exchanger design will not understand the implications of any one design since heat transfer and fluid flow effects cannot yet be modelled to sufficiently detailed levels, eg the characterisation and modelling of the onset of boiling within an exchanger.

It is suggested then, reasonably enough, that there are no complete mental models for empirical phenomenon and beyond a certain level there is little to be gained from adding more information as regards the usefulness of a model. As a consequence many models that are used by people merely simulate or mimic a phenomenon rather than refer to the underlying principles ; yet they are still useful from an explanatory point of view. Many models of processes will therefore be little more than high grade simulations but nevertheless useful provided they are reasonably accurate for the purposes required.

5.4.1.2 Mental Models and Effective Procedures

One of the main themes running through Johnson-Laird's work is that any psychological theories in cognitive science, and in particular mental models, should be expressed as effective procedures. The term effective procedure is used in a computational sense and is suggested as a means to test the explanatory adequacy of a theory. This has some immediate implications relevant to this work:

1. It implies that a theory can be implemented as a computer program suitable for use, for example, in a knowledge based reasoning system.
2. It does not imply that a theory should necessarily be implemented as a program, eg an equivalent, semi-formal axiomatic formulation would suffice.
3. Even less does it imply that the whole theory could be implemented as a program but it would be a considerable bonus if it could be

Johnson-Laird is quick to point out that he is not claiming that all of human thought can be reduced to simple forms of computation. What is suggested rather is that effective procedures be used as a criterion of effectiveness in that theories be stated explicitly and in a way that takes little for granted. This is a weak condition on the adequacy of a theory since it merely ensures, as Johnson-Laird puts it, that the theory is not a "wooly phantasy". The development of effective procedures for reasoning processes will be by no means trivial. Even in mathematical terms, many different effective procedures can be formulated for any given computable function.
Consider for example, the following quote regarding what it is to "understand" a Concept:

> Understanding certainly depends on knowledge and belief. If you know what causes a phenomenon, what results from it, how to influence, control, initiate, or prevent it, how it relates to other states of affairs or how it resembles them, how to predict its onset and course, what its internal or underlying 'structure' is, then to some extent you understand it, (page 2, Johnson-Laird 1983).

Developing effective procedures to capture these sorts of aspects of reasoning is a formidable challenge. Due to the arbitrary nature of human behaviour it would appear that there is little possibility of ever being able to model accurately a particular individual's reasoning processes. On the other hand there are no a priori reasons that would appear to rule out the study of a wide range of reasoning abilities, eg modelling inference and learning techniques, Concept attainment, recognising mood and emotion in discourse, etc.

Johnson-Laird suggests himself that much of his work is "an argument to the effect that the construction of mental models, the communication of their contents, and reasoning on the basis of such representations, are nothing more than computational processes", (page 12).

An initial summary, then, is that engineers understand the world by constructing working models of it in their minds. These models may refer to, for example, real or imagined processes, plant items, their purpose and mode of operation, theoretical models describing their behaviour, etc. The models will invariably be incomplete and simpler than the entities they represent, making use of procedures that attempt to mimic or simulate their behaviour. The theory of mental models itself must also

provide a working model, the purpose of which happens to be a device or means for constructing other working models. It must be capable of being formulated as an effective procedure that must specify constraints on the classes of possible models as well as the constituent elements and operations that comprise them. As such it forms a meta level statement of the reasoning processes involved. The bulk of Johnson-Laird's work is concerned with providing details or discussions on the meta level issues involved. Some of the more immediately relevant ideas are introduced below.

Before leaving this introduction it is worth emphasising the usefulness of limited models. Consider for example the very limited models people have regarding the use of numbers compared with the axioms of real number algebra, or the way in which the Concept "time" can be described. Despite these limited models people can work very effectively on problems involving these Concepts.

This is a familiar theme in process engineering in which engineers regularly make use of simplified models to analyse and solve complex, real life problems. What is not well understood at this stage is the mechanism or effective procedure required to formulate such models. The attraction of the study of mental models, in conjunction with the earlier ideas on Conceptual categorisation, is that it provides a means to investigate the reasoning techniques involved.

5.4.2 MENTAL LOGIC

One of the most important aspects of Johnson-Laird's work is the study of the status of a "mental logic". Most theories of reasoning invariably assume the existence of a mental logic. The assumption of the existence of a logic in the mind, however, runs the risk of paradox since how could such a logic be acquired by someone who could not already reason soundly? Conversely, the claim that there is no logic in the mind also appears to be false since how could a species have ever invented logic if it was incapable of valid inference?

After extensive discussion Johnson-Laird comes to the conclusion that it is possible to reason validly without the use of logic formalisms. Before stating how this can be achieved it is worth repeating here several observations that led him to make this conclusion:

1. People frequently make unwarranted or incorrect inferences when faced with reasoning tasks.
2. It has been shown experimentally that the content of propositional statements has a striking effect on the reasoning process. Inference rules in propositional calculus are, of course, valid regardless of content and based purely on form. It is apparent that problems posed in an abstract form are less easily handled than those involving familiar objects, verbs etc. Johnson-Laird suggests that in the

latter case people can construct relevant mental models of the relationships involved that make valid reasoning more likely.

3. It is clear that people follow extra-logical heuristics when making spontaneous inferences, ie when choosing which conclusions to draw from the infinite number that are implied by a set of premises. For example, given the premises
"If there is no error message then the program is compiled" and
"There is no error message"
then Johnson-Laird suggests most people draw the conclusion "the program is compiled" rather than "there is no error message or the program is compiled", etc. The meta principles governing the selection of inference rules lie, of course, outside the formalism of logic but are essential to its successful application, see chapter 2. A common heuristic that emerges is that inferences are guided by the principle of maintaining the semantic content of the premises but re-expressing them with greater linguistic economy. The study of logic has emphasised the evaluation of given statements or conclusions rather than the spontaneous production of non-trivial conclusions.

Apart from these main points there are several other obvious, niggling queries such as "Why has it taken centuries for theories of logic to emerge if we innately possess these reasoning skills?", "Why do people find the study of logic a difficult subject?", "Which form of logics, from the many studied to date, do people make use of?", "Is it reasonable to assume that people can cope with handling the semi-decidable or undecidable decision procedures found in predicate and non-monotonic or modal logics?".

Given these problems Johnson-Laird suggests that it is necessary to investigate forms of valid reasoning without recourse to logic in an attempt to explain psychological forms of reasoning. He does this by developing possible effective procedures that could be used for reasoning with propositions and syllogistic forms of reasoning. The exact formulation of these procedures is not that important here. Rather it is the principle of the approach that is important and the observations made regarding the use of mental models. (The reader is referred to the original work for detailed descriptions of the procedures involved). Some of these observations are repeated below.

5.4.2.1 Propositional Based Reasoning

Johnson-Laird shows that there are least two effective procedures that can be used for making valid inferences without making use of rules of inference. All this implies however is that propositional style reasoning can be achieved using procedures. This is not that surprising since theorems are proved by meta level search procedures in standard logics, albeit making heavy use of inference rules. What is important is the observation that propositional based reasoning using a truth functional approach is only a possible mode of thought for most people rather than an habitual one.

Johnson-Laird provides several interesting examples, (see p41-63,1983) which suggest that a more likely explanation is that mental models are formed to represent

events described by a set of premises. This model must be based on the content of the describing propositions as well as implicit inferences based on the general knowledge of the reasoner. Johnson-Laird also suggests that linguistic assertions in daily life can only occasionally be interpreted in a truth functional way, eg the on-off logic of electrical switches. The following quote emphasises this point:

> Reasoning is not a matter of recovering the logical forms of the premises and applying rules of inference to them in order to derive a conclusion. It is here not even a matter of substituting truth values for constituents of propositions and working out their compositional consequences. The heart of the process is interpreting premises as mental models that take general knowledge into account, and searching for counter examples to conclusions by trying to construct alternative models of the premises, (p54, 1983).

This is a strong claim and is in opposition to the intuitive approaches taken in developing AI knowledge based systems, as described earlier in chapter 3. On the other hand, the notion of some sort of informal model which takes into account both context dependent information and general knowledge can be seen in many of the more sophisticated applications.

Johnson-Laird strengthens his indirect case against a mental logic by considering connectives in assertions that are not truth functional, eg "because and "if". Of particular interest is the use of the word "if". Johnson-Laird convincingly argues that although there is a uniform procedure for interpreting conditionals considerable variations arise in its logical behaviour. In particular it appears that the logical properties are in part determined by the nature and content of the propositions that they interrelate. This variation in interpretation would be difficult to justify using some sort of mental logic.

As regards the use of mental models and "if" statements, Johnson-Laird suggests that mental models are primarily constructed using the antecedent information. The consequent of the conditional is then evaluated with respect to the model. For example, if the consequent is an assertion then the validity of the conditional as a whole depends upon the validity of the resultant model, possibly projected to subsequent events if the assertion involves tense. If the consequent is a question it can be answered with respect to the model and if it is a request it can be carried out when the model corresponds to observed reality. In a mental model framework it is relatively clear then how to account for content in inferences and cater for the interpretation of non-truth functional connectives.

5.4.2.2 Syllogistic Reasoning

In order to provide more convincing evidence for the use of mental models Johnson-Laird considers the topic of syllogistic forms of reasoning. This is an interesting topic and is worth briefly discussing.

Syllogisms are deductions based on two premises, both premise and conclusion being in one of four forms:

1. All X are Y
2. Some X are Y
3. No X are Y
4. Some X are not Y

For instance, a typical syllogism is

```
All B are A and All B are C
therefore some A are C
```

The occurrence of B in each premise allows an inference to be made and is called the middle term. The logic of syllogisms is relatively simple since there are only four possible arrangements of the middle term. These are shown below for the possible arrangements of the middle term B.

```
A - B    B - A    A - B    B - A
B - C    C - B    C - B    B - C
```

There are only 64 distinct forms, therefore, of pairs of syllogistic premises (4 types of first premise, 4 types of second premise and 4 arrangements). Syllogisms were first formulated as early as Aristotle. Their study in logic, however, has largely been neglected since the introduction of general quantifier theory by Frege in 1879.

Why are syllogisms interesting? They are interesting for a number of reasons:

1. Syllogistic forms of reasoning are often used to dismiss the study of syllogisms! (see p71, J-L, 1983)
2. Syllogisms often arise in many types of deduction. For example many inferences based on general assertions involve syllogisms, eg

```
a) Do you have a TV set?  Yes
   Do you have a licence? No
   Well, it requires a licence. Why?
   All TV sets are required by law to have a licence

b) Some arguments that are likely
   to be hard to understand are syllogisms.
   No argument that is used spontaneously
   is likely to be hard to understand.
   Therefore, some syllogisms are not used
   spontaneously in daily life.
```

It appears that simple sorts of syllogism are easily understood and used frequently when making logical type deductions. It is also clear that certain types of deductions, eg example b) above, are much harder to follow.

Johnson-Laird spends considerable time explaining how effective procedures can be formulated for various reasoning tasks using syllogisms. A simple example taken from Johnson-Laird is given here since it illustrates a fragment of the mental modelling approach.

Consider the premises "All the artists are beekeepers" and "All the beekeepers are chemists". Johnson-Laird suggests the use of a simple tableau as a sample representation of the first premise:

```
artist = beekeeper
artist = beekeeper
artist = beekeeper
         (beekeeper)
         (beekeeper)
```

The beekeepers in brackets are those that are not artists. The number of entities used is entirely arbitrary as long as it adequately represents the situation. The tableau is easily extended to take account of the second premise:

```
artist = beekeeper = chemist
artist = beekeeper = chemist
artist = beekeeper = chemist
         (beekeeper) = (chemist)
         (beekeeper) = (chemist)
                       (chemist)
```

From this tableau it is easy to confirm the deduction that "All the artists are chemists" is true. Deductive inference usually requires search for counter examples as well. Consider the premises "None of the authors are burglars" and "some of the chefs are burglars". The first premise is modelled by:

```
author
author
author
.....................(Dotted line indicates disjoint groups)
                burglar
                burglar
                burglar
```

The second premise is added as before:

```
author
author
author
                burglar = chef
                burglar = chef
               (burglar)  (chef)
```

Two tempting but unwarranted inferences would be "None of the authors are chefs" or "None of the chefs are authors". Another unwarranted inference is that "some of the authors are not chefs", as shown below.

```
author
author
author = chef
         burglar = chef
         burglar = chef
```

```
          (burglar)
```

This is because all the authors might possibly be chefs as well:

```
author = chef
author = chef
         burglar = chef
         burglar = chef
        (burglar)
```

A valid conclusion that can be drawn is that some of the chefs are not authors.

Johnson-Laird goes on to develop an effective procedure for reasoning syllogistically. The formation of the tableaus corresponds to the formation of mental models, albeit in a highly idealised and simplistic form. Despite the limited model, as indicated earlier, useful inferences can then be performed.

It is a simple matter to translate these ideas into a symbolic programming language, such as Prolog or Pop11, that allows the construction of arbitrary data structures. A possible representation of the premise form "All of the X are Y", always given by

```
x = y
x = y
    (y)
    (y)
```

could be (in Prolog)

```
model-entry(x - y)
model-entry(x - y)
model-entry(y)
model-entry(y)
```

A more sophisticated representation which does not rely on an interpreter "running out" of entries would be:

```
model-entry (xl..xn - yl..yn..OtherYs).
```

This version requires the distinction between variables and hypothetic instances). A number of other equally useful forms could be imagined. The important point is that it is easy to set up the limited kinds of mental model proposed by Johnson- Laird.

The other syllogism forms are equally simple:

```
Some of the X are Y          x = y
                             x = y
                            (x)  (y)

None of the X are Y          x
                             x
                             ......
                                  y
                                  y

Some of the X are not Y      x
                             x
                             ......
                             (x)  y
```

Suitable Prolog forms are easily created to capture these tableau structures.

The rest of Johnson-Lairds discussion on syllogistic reasoning is concerned with refining the procedures involved and explaining experimental results that show why certain syllogisms are difficult to follow, namely those with premises requiring two or more models.

5.4.2.3 Other Forms of Deductive Reasoning

Having illustrated the approach in the special cases of syllogistic reasoning the argument for the use of mental models is extended to more general aspects of inference. In the discussion the following four points are established:

1. A general theory of inference based on mental models can account for both implicit and explicit inferences. Explicit inferences are those that usually cause most difficulty and depend on searching for alternative models that may rule out currently held conclusions. Implicit inferences are essential in understanding written and spoken discourse. These type of inferences make use of the context and content of the discourse as well as background knowledge. (This will presumably be in the form of typical instances or prototypes rather than simple default values given the discussions in the previous two sections). As a consequence, a single mental model can be formed. No attempt to search for alternative models will be made until evidence or reasons for doing so should arise.
2. The theory solves the paradox of how children, or knowledge based systems for that matter, can learn to reason or acquire rules of inference before they can reason validly in the first place. The paradox is solved because it is based on the false assumption that children either need to acquire inference rules or possess them innately in order to make valid deduction. Valid reasoning can, in fact, be performed without recourse to logic.
3. The theory is compatible with the fact that humans are certainly capable of making both valid and invalid inferences.
4. It is also compatible with the way in which formal logics are developed. Certain inferences are much more difficult for people than others. This difficulty provides the motivation for the search for systematic principles governing valid inferences. Hence the formulation of classic axiomatic systems.

These are general points but very important ones as they effectively imply the process of Concept attainment can be implemented within a framework of effective procedures. This includes the refinement of both working definitions of Concepts, as constituent part of a mental model, and the reasoning procedures used for their interpretation. The claim that effective procedures for these mechanisms can be formulated is a significant one. The problem with Bruner et al's account of Concept attainment and that of Piaget concerning a child's ability to learn to reason, namely by internalising their own actions in some way and reflecting upon them, is that neither describe or suggest a mechanism by which the necessary reasoning can actually be performed.

The theory of mental models suggests that these mechanisms should be amenable to specification as general procedures.

5.4.2.4 Summary: The Status of Mental Logic and its Consequences for Knowledge based Systems

The theory of mental models shows, perhaps more clearly than any other study in cognitive science, that it is not necessary to assume the existence of a mental logic to account for valid reasoning. Difficulties in performing certain types of inference can be described in terms of the difficulty of constructing integrated models of a set of premises and the subsequent testing of those models for contradictions, see also Morris (1987,p7). The role of logic is clearly reduced to that of a task variable which removes it from its widely held view as a putative account of reasoning processes. This includes the specialised area of reasoning about logical problems themselves.

The conclusion has important consequences for many areas of cognitive science and artificial intelligence and in particular those concerned with the continuity and consistency of comprehension and inference processes, eg the development of knowledge representation systems. For example the requirements of an advanced knowledge based system for process engineering design require features such as Concept refinement, self consistent learning, catering for vagueness, etc. Other research in knowledge representation and the discussion of logics throughout chapters 2 to 4 indicate that such a system would be truly difficult to implement in logic terms. It does not seem practical at this point, therefore, to pursue the development of a process design environment that requires logic based reasoning as the basis or means in which other reasoning capabilities are implemented. Rather inferences based on formal logics should be viewed as results from a computational tool. This reduces the status of formal proof procedures to that of other useful problem solving techniques, eg equation solvers, although it is clear that the former has wider application in most types of search problems. This is in direct contrast to many of the AI systems referred to in chapter 3 whose reasoning and representation techniques are either explicitly or implicitly formulated in a logical framework.

5.4.3 Types of Mental Representations

Having summarised the background context in which the theory of mental models was developed it is now possible to describe specific aspects of the theory that will be of subsequent importance to the development of a knowledge representation system. Johnson-Laird proposes that at least three types of high level mental representations are worth considering:

1. Propositional representations
2. Images
3. Mental Models

The first two notions draw on a wide range of psychological research for which Johnson-Laird provides numerous references, (chapter 7, J-L, 1983). The claim that images form a distinct sort of mental representation is a controversial one. Proponents of this view suggest that images represent objects, i.e. there are analogical structural relations between image parts corresponding to the perceptible relations between the parts of the object they represent. Further they are amenable to mental transformations, corresponding to changes in view of an object, and the mental processes underlying their experience are similar to those underlying the direct perception of an object or picture.

(An interesting account of the use of images to represent spatial information and their use in problem solving is Kosslyn (1980)).

Critics of these ideas claim that images are only epiphenomenal, ie purely interpretations, and are based on a single underlying form of mental representation corresponding to propositional statements. As Johnson-Laird points out there is in fact very little difference in the underlying functionality claimed by the two schools of thought except that images are said to represent objects whereas propositions are only said to be true or false of them. If the argument is about the level of description then it can be trivialised as follows.
Consider the useful Concept of a two dimensional array in a program that can be used as an entity in specifying effective procedures for some reasoning task. Using the doctrine of functionalism, it is irrelevant that at one level all that is required is a string or vector of symbols while at a higher level it is useful to reason about an array of locations. It does not matter whether a phsyical two dimensional array of locations exists. All that does matter is that the embodiment should "function" as an array. In a similar vein Johnson-Laird suggests there is no reason to suppose that the mind is not organised along similar lines re the use of images and propositions.

Propositional representations, then, are mental representations of linguistic

expressions. For the meantime this can be assumed to consist of strings of symbols. Images are taken to be "perceptual correlates" of mental models from a particular point of view, (p157, J-L, 1983). The rest of this discussion shall focus on the differences between propositional representations and mental models. Given the discussion above, however, it is not difficult to extend these arguments to the use of images. The ability to directly handle and reason about graphical images of objects is of considerable importance to the long term development of work station, graphics based computing environments.

A principle assumption of the theory of mental models is that verbal descriptions are initially represented in a propositional form, ie expressions in a "mental language", that requires a semantics to map these propositional representations into mental models. The search for alternative interpetations of models requires that an independent representation of a set of premises be available, ie in propositional form. These propositional representations can refer to real or imagined situations but are always interpreted with respect to a mental model.
Two basic claims are made regarding the structure and content of mental models. The first is that unlike propositional representations, mental models do not have an arbitrary structure but one that is analogous to the state of affairs it represents. At first sight this claim seems highly questionable. Johnson-Laird points out, however, that the analogical structure can vary considerably and may only require that separate entities be used to stand for individuals. The second claim is that mental models, and images, are highly specific, eg it is only possible to form an image of a specific triangle, say, rather than a "general" triangle, whatever "general" may mean. This is interesting in that although a model must be specific it can still represent a general class of entities. For example, the interpretation of a specific model depends on a number of processes all of which may treat the model as a representative sample of a larger set. This is presumably exactly the case when interpreting Rosch's prototypes, which are specific cases, or Bruner et al's typical or generic instances.

The last example raises the fundamental issues that the "function" or purpose of a model cannot be ignored. The structure and content of a model must be supported by a description of the processes interpreting the model if valid deductions are to be made regarding the model's content. The problem of "understanding", of course, is that natural language is often inherently vague and the structure of discourse is typically fragmented. This can easily lead to arbitrary assumptions or unwarranted inferences being made as switches are made between one interpretation process and another.

The assumption that propositional representations can be mapped into mental models requires directly addressing theories of word meaning and coherence of discourse, see points 2 - 5, page 377 above. This is an extremely difficult and complex task as it involves many psychological, philosophical and linguistic issues. No attempt is made, therefore, to even attempt to summarise Johnson- Laird arguments even though most of these are themselves only introductory in nature. Instead only a selection of the technical terms and issues involved shall be described. The reader is referred to the original text for further explanatory details.

5.4.3.1 Intension, Extension and Formal Models of Language

The mapping process of a propositional representation, as a string of symbols, into a mental model resembles the type of semantics used in formal logics. The use of logics in the analysis of natural language in this way is referred to as model theoretic semantics. The central principle of this approach is the same as that of classical logic, namely that the principle of compositionality applies. That is, the semantics are compositional or denotational, see chapter 2. Frege's work on logic draws the distinction between the "sense" of an expression and its "reference". The "reference" of an expression is simply the entity the expression refers to in the world. The "sense" of an expression is less clearly defined but is concerned with the part of the expressions meaning that connects it with its reference and can be thought of as the "essence" of the meaning that can be expressed by different, subjective "ideas" of it, ie different "ideas" can be associated with the same "sense" as is the case when one person's "idea" of something is different from another's "idea" of it.

The somewhat vague definitions of sense and reference have since been replaced by notions of intension and extension in formal studies of semantics. Johnson-Laird quotes Frege's example of the expressions the Morning and the Evening star. These both have the same extension, namely the planet Venus, but they have different intensions. One means the star observed in the morning while the other means the star observed in the early evening. In abstract terms, the extension of a declarative sentence is its truth value and its intention the conditions which have to hold for the sentence to be true.

In an ideal, abstract language, such as predicate calculus, one expression within a sentence can be replaced by another with the same extension without affecting the truth value of the sentence. For example, if "the reactor overheated" is true then so too is "R101 overheated" provided that the reactor and R101 have the same extension. The idea that the intension of a sentence can be built up from the intensions of its component parts in a way that only depends on their grammatical mode of combination is a principle that is central to model-theoretic semantics. Johnson-Laird

argues, however, that it is an assumption based on an ideal state of affairs. Further, the assumption appears to be invalid and gives rise to intractable difficulties when trying to take into account various observed phenomena in the interpretation of word meanings. An example quoted earlier was the widely observed effect of subject content on the ability to interpret and reason about propositional statements. If this is the case then it is an extremely important observation since it is precisely an extended theory, and implementation, of word meaning that is required in a self-consistent knowledge based system. It is worth outlining then the salient features of the model theoretic approach and the main problems it fails to address.

Denotational semantics and the assignment of truth values for a formal language are carried out not with respect to the real world but to a model. A formal model is an abstract construct that consists of a function that maps syntactically well formed sentences to elements in some specified model structure. In applications of computational linguistics this structure is typically some sort of nested tree structure, similar to proof trees as described in chapter 2. Many applications also require the use of the notion of possible worlds as discussed earlier in the context of the use of modal logics, section 4.3.1.6.

In such logics an assertion, i.e. proposition, is necessarily true in a particular world if and only if it is true in all possible worlds that are accessible from that world. Different assumptions can be made regarding the form of the accessibility relation, see also Appendix D, corresponding to different axiomatisations of modal logics. This formulation provides a clearer definition of the terms intension and extension introduced above:

> The intension of a predicate - the property that it captures - can be treated as a function from the set of possible worlds to sets of individuals; the extension of the predicate is the set of individuals to which it applies in the particular possible world under consideration..... The intension of a sentence - the proposition that it expresses - can be treated as a function from the set of possible worlds to truth values (true and false); the extension of the sentence is its truth value in the particular possible world under consideration. (p172, Johnson-Laird, 1983).

One of the most well known applications of model theoretic semantics to natural language analysis is that of Montague (1974) who makes of an intensional logic based on lambda calculus, see Appendix B. This work is briefly described in Johnson-Laird (p174,1983). For those unfamiliar with computational linguistic techniques Grishman ("Computational Linguistics", 1986) and Winograd ("Language as a Cognitive Process", 1983) are excellent introductory texts. The limited success, however, of systems based on the model theoretic approach in adequately handling many common linguistic forms has led several researches to question the whole

validity of this approach. In particular, the following general points are worth noting:

1. The idealisation of assuming that language, and word meaning in general, can be mapped to a model without reference to the general cognitive processes involved leads to intractable difficulties with the semantics of sentences involving propositional attitudes such as belief.
2. The notion that there is a function that provides each basic lexical item with an interpretation in a model structure is an oversimplification that relies on the vocabulary of a language being expressed as sets of necessary and sufficient conditions. The previous discussions on Conceptual categorisation and the use of prototypes indicate that for many types of word this is very unlikely to be the case.
3. Model theoretic semantics merely assumes the existence of such functions without specifying how they can be formulated or are intended to work.
4. Propositions expressed in sentences in ordinary language, and hence in text, rules, etc covering some topic of process engineering, generally depend on the context in which they occur, e.g. time, place, situation, listener, speaker, etc. To assume that "context" can be denoted by a possible world is fraught with danger since it suggests that "context" is some sort of passive external object characterised by a set of indices corresponding to time, place, speaker, etc. This is naive on two counts. First, the notion of the context of an utterance, or a written sentence, requires that there are at least two contexts: one for the speaker and one for the addressee. The difference between the two, as Johnson-Laird argues (chapter 14, 1983), is in fact essential for intelligent communication.
   Second, just about anything could be considered relevant to the context of the interpretation of a sentence since listeners in general have no means of knowing what will or will not be relevant information until a sentence is heard or read. It is unlikely then that context can be represented as a static, predefined sentence of indices waiting to be "filled in".

In order to understand how the theory of mental models can help provide a framework in which to tackle these problems it is necessary to consider first some aspects of the fundamental nature of the meanings of words.

5.4.3.2 Aspects of Word Meaning

The precise nature of the meanings of words and Concepts remains an unresolved problem in the fields of philosophy and cognitive science. The theory of mental models does not attempt to provide a definitive set of answers to the questions that have puzzled philosophers for many years, such as those quoted at the start of this section. Instead, Johnson-Laird attempts to show that by considering word meaning within a cognitive framework based on mental models many of the standard arguments, some of which are given below, appear to be false antithesis.

The main objective of the remainder of this thesis is to use these arguments in conjunction with the ideas introduced in the previous two sections to provide an outline for a knowledge representation system that can address the fundamental requirements of a process design environment.

5.4.3.3 Necessary and Sufficient Conditions

A question that has aroused much controversy is whether or not word meanings can be expressed as necessary and sufficient conditions. It appears from the discussion in chapter 4 that intuitive approaches taken in the development of simple knowledge representation systems have given rise to a confused situation. It was pointed out that many frame based systems are implemented implicitly assuming that some slots and slot values be treated as necessary conditions. At the same time, as Brachman (1985) made clear, the ability to override inherited slot values makes it impossible to represent definitional conditions or contingent universals in a rigorous way, i.e., exactly the methods required to represent necessary and sufficient conditions. More recent developments, e.g. the OPUS system, Nado and Fikes (1988) and Krypton, Brachman, Fikes and Levesque (1983), have only attempted to clarify the situation rather than analyse the need for necessary and sufficient conditions to represent meanings. The reason why this problem is so important is that it fundamentally changes the nature of the representation of Concepts and the process used to interpret them.

In the study of philosophy there are two schools of thought of direct relevance to this problem, namely Realism and Psychologism. The Realist approach, which was argued by Frege in modern times but goes back to Plato, maintains that meanings are independent of the mind. By that, it is essentially meant that the referents of expressions, propositions, etc., are "real entities awaiting discovery rather than mental constructs awaiting invention", Johnson-Laird (p183, 1983). This requires that a Concept be defined by the essential characteristics of the objects of which it is true, being true for some objects and false for others. Frege's notion of some sort of "real sense" of a word or sign as opposed to different individuals' "ideas" of it, where the terms sense and idea are used in the technical way by Frege, lends itself more readily to the position that word meanings can in some way be properly defined as necessary conditions.

Given the considerable research in cognitive science, as represented by the previous two sections on Concepts and prototypes, it is difficult to see how this position can be justified. Before explaining why, it is worth noting two points that Johnson-Laird makes regarding the Realist approach:

1. If there is a "real" sense of a sign that can be the common property of many individuals how is it passed from generation to generation without entering the mind? e.g. Frege states "one can hardly.....that mankind has a common store of thoughts which is transmitted from one generation to another", (see p183, J-L, 1983). If these meanings do enter the mind then in what way are they different

from individual's "ideas" of that "sense"? Yet Frege specifically states that the sense of a sign is not part of the individual mind.

2. If people have different ideas about the object denoted by a sign it does not necessarily imply that they cannot also have a common idea about the sense of a sign. Only if the latter were true would Frege's rigorous distinction between sense and idea appear valid.

The whole basis of Conceptual categorisation, as described earlier, implicitly argues against the Realists approach, or more precisely my interpretation of it does. It fundamentally recognises the invented and often arbitrary status of concepts developed in a cognitive framework that takes account of the effect of social interaction on the process of Concept attainment. The perceived social acceptance of certain Attributes in determining the criterial status of all Attributes that constitute a Concept goes a considerable way to explaining how Frege's store of thoughts are passed from one generation to another. Further Frege's account is oversimplified in that it concentrates primarily on the intensions of objects. Bruner et al's work shows that a much richer set of Concept type is required to account for the wide range of phenomena people reason about, e.g. perceptual level concepts that are pre-verbal (space, motion, colour etc), functional concepts, identity and formal equivalence categories etc. In addition, these Concepts are bound up in a superordinate structure and cannot be treated independently of one another.

These observations are in line with arguments put forward by proponents of Psychologism. In particular, Johnson-Laird points out the work of de Saussure (1960, written 1907-1911). de Saussure argues that is only possible to define Concepts in terms of its relationships to other Concepts. The links between the words and Concepts involved will reflect the relational nature of an individuals lexicon and society's approval of the use of these words. de Saussure, as Frege, takes the view that language transcends any particular individual but argues against there being any objective realm to provide necessary and sufficient conditions to define the sense or intension of any term.

5.4.3.4 Concepts, Prototypes and Family Resemblances

In order to gain a better insight into word meaning and the question of necessary and sufficient conditions it is useful to reconsider aspects of Conceptual categorisation and the use of prototypes.

In the case of categorisation and the use of formal equivalence categories it seems clear that many Concepts have Attributes that are not combined in a simple AND/OR Boolean way. Many concepts have instead a relational or functional nature, e.g. a table not only has legs but the legs "support" the top of the table. Many intuitive descriptions of AI type frames or objects presuppose some common characteristics and values which run through all examples of a given class. For many types of natural

kind Concepts the work on family resemblances has shown this to be a false assumption. If there is a common element it tends to be in terms of function, or other perceptual level characteristics, rather than form. A familiar example of this in process engineering is what sorts of object can be considered to be heat exchangers. The actual form of the object is of secondary importance, i.e. non-criterial, as it can involve numerous variations ranging from shell and tube constructions, plate exchangers, graphite blocks, fluidised beds, etc. Instead, the main reason for describing some object as an heat exchanger is whether or not its primary function is to transfer heat between two notional streams, either by indirect or direct contact.
As regards the development of prototypes and taxonomic structures based on family resemblances, which does not necessarily involve the use of common characteristics, Johnson-Laird quotes the work of Smoke (1932) which makes the point well:

> As one learns more and more about dogs, his concept of "dog" becomes increasingly rich, not a closer approximation to some bare "element"... No learner of "dog" ever found a "common element" running through the stimulus patterns through which he learned.

This is exactly what is implicitly implied by the argument that categories have an internal structure with an intended range of application learnt by experience, the prototype(s) of a category representing a specific type of class norm. Wittgenstein's work on family resemblances and the nature of concepts holds, then, that many types of Concept definition are not specified by essential characteristics or sets of necessary and sufficient conditions.

5.4.3.5 The Nature of Intensions

Much of the philosophical argument on the nature of intensions and word meaning is made from two idealised or abstract points of view that fail to take into account the perceptual mechanisms involved in the identification of Concepts and the use of language. The purpose of this discussion is not to pursue these arguments but to identify the nature of Concept definitions in a knowledge representation system based on observations and research in cognitive science and whether or not this involves the use of definitional conditions.
The discussion above has suggested that in the case of natural kind objects, at least, the assumption of necessary conditions for category identification is not valid. Pulman (1983) provides an excellent introduction into a more detailed discussion of the subject and the characterisation of other word types. His arguments are summarised in Table 6.1 below

Table6.1: Basic Word Types

	Subject of some stable generalisation	Subject of some analytic specification	Understood mainly by stereotype
Natural	yes	no	yes
Primary	yes	can be	yes
Nominal	can be	yes	can be

This table requires some explanation. Some of the terms involve technical definitions which are explained at length in Pulman (p138-169, 1983).

The word stereotype in the third column is used to indicate a generalisation of Rosch's notion of prototype. A stereotype may be mainly perceptual information, e.g. "red", or primarily functional, e.g. "torch", or rely on connotation eg "spinster".

The term analytic specification refers to the type of definitional conditions used in attempts to define words like bachelor (male, single, etc). There is an important distinction between weakly and strongly analytic specifications. Weakly analytic specifications are conditional only whereas strongly analytic forms are bi-conditional.

The term stable generalisation is used to help distinguish between natural and nominal kinds, these terms being more easily introduced by examples. Most natural life forms, e.g. apple, orange, dog, tree, are for obvious reasons classified as "natural kinds". The research on prototypes and family resemblances indicates that these objects are primarily understood via a prototype and that their true intensions in terms of an analytic specification are undefined. The term stable generalisation involves two simple ideas shown by the following example from Pulman (p156, 1983).

Consider the sentence "dogs are animals". If the answer to the question "Could all dogs have turned out not to be animals?" is yes then the original statement is corrigible and therefore not analytic. If, however, the answer to the question "Given that we think dogs are animals could the 1000th dog not be an animal when all the previous examples were?" is no then the original statement passes the "counter-example test", i.e. it is not refutable by a single counter-example. Statements which are both corrigible and pass the counter-example test are stable generalisations.

If natural kind terms are primarily understood via prototypes and can be subjects of stable generalisations then it clarifies the status of "necessary conditions". For example, consider the natural kind "tiger". Most individuals' prototypes presumably consider tigers to be gold and black in markings. This is not a strict necessary condition. It is not even a weak, necessary condition, eg we can have a tiger without striped markings. It is just that the prototype is so familiar and rarely brought into doubt that a posteriori its markings take on the status of essential characteristics. Such a posteriori, i.e. after the event, truths are referred to by Kripke (1972) as

metaphysically necessary truths.
This account is in agreement with the account of conceptual categorisation as it recognises the inverted status of both categories and Attributes. If sufficient exemplars of non-striped tigers or striped non-tigers were discovered then it is likely that the term tiger would be replaced by two or more new terms.
The basic difference between natural kind terms and nominal ones (Locke, 1960) is that the latter are part analytic, e.g. bachelor, farmhouse, breakfast, etc. That is weakly analytic specifications can be useful in checking category membership of questionable exemplars. The use of stereotypes is still essential however for discriminating between examples such as glass and goblet which would have similar analytic specifications both in terms of form and function. It is important to note that most of these categories are purely based on social convention and membership is open to negotiation, particularly in dubious cases. Johnson-Laird refers to such terms as having a "constructive semantics" by which he means that the intensions of such words are mental constructions imposed upon the world.

Also included in this class of word are many man-made artefacts. The specification in this case is usually in terms of intended function rather than some internal structure or form. In particular the majority of functional plant equipment items, e.g. pumps, heat-exchangers, reactors, etc., are covered in this important class of word. It is worth noting that many such categories are labelled by the nominalisation of the verb involved, e.g. pump. In these cases it is likely that the Concepts will be closely bound up with related functional Concepts.

Other nouns stretch the notion of constructive semantics still further, e.g. home, chair, melody and often only have weak analytic specifications concerning function. For example, what can be considered to be a table, or a chair etc can vary considerably and may look quite different from an individual's prototypes of these items. Similarly, bricks, boulders, even the heel of a shoe can be substituted into a sentence about hammering nails, and easily understood despite the fact they bear very little resemblance to a prototypical hammer.

In summary then, nominal kinds make more use of analytic specifications but these specifications are only weakly analytic and merely reflect perceived social conventions. Considerable use of stereotypes is still required as a basic means of utilising such Concepts.

The third type of word in Table 6.1 above is termed primary, following Locke (1960). These words include examples such as hot, dry, wet, liquid, solid, red, blue, smooth

etc. These words often appear as adjectives or in noun form and can be thought of as "the basic stuff" of the world of which our senses are aware. It is difficult to be more precise because these words in common usage are only labels for perceptual level notions that are hard to verbalise other than through common examples, comparisons, etc. In most cases they are understood directly via a prototype, e.g. water, stone, etc. Scientific type classifications of primary kind words provide an alternative means of description, e.g. what constitutes a liquid, vapour, gas etc., but as with necessary conditions for natural kind words these are very much a posteriori specifications. The primary means of use and understanding of these words in "common sense problems" is again via stereotypes.
This is a fundamental observation as it is precisely this sort of common sense understanding that has long eluded developments in AI.

Finally it would be naive to think that the basic word types natural, nominal and primary are sufficient to deal with the range of Concepts and associated intensions needed in a process design environment. For example many functional Concepts do not have a simple one to one correspondence with objects. A simple example is the notion of how to solve an equation. How is the meaning of this functional concept represented?
Another extremely important class of words not covered here are preposition type words, e.g. at, on, with. This topic is well discussed in Miller and Johnson-Laird (1976). These words also have a constructive semantics which is important for the theory of mental models since they can be implemented as effective procedures, creating the appropriate number of entities in a mental model, their inter-relationship and so on.
The same applies to the basic set of sentential connectives, "and, or, if, then, else, because, so..." and the various wh-words, "where, what, which....", either used as interrogatives or in relative clauses.

5.4.3.6 Abstract Concepts, Formal Languages and Necessary Conditions

Given the discussion above the position taken here is that the status and use of analytic specifications of many types of word in natural language requires careful consideration. This contrasts with many other AI based developments which implicitly assume sets of necessary and sufficient conditions in frame or object definitions. In many cases this is seen to result from an oversimplistic view of the semantics of Concepts and their use in everyday thought. The question then remaining is "are there any concepts which require strongly analytic specifications?" The only good examples I can offer are ideas formulated in abstract or formal languages such as mathematics. For instance, precise geometric shapes fall into this

category. Statements regarding the definition of a circle, square, equilateral triangle, etc. are incorrigible since they are always "true" by definition. Interestingly enough, even these examples are rooted in perceptual level characteristics. As Bruner et al point out, however, (p6, 1956) once a formal equivalence concept has been established many more dependent Concepts can be generated with little regard for their original source. This is particularly well demonstrated in the development of many modern areas of mathematics. It is still the case however that most of these Concepts can still be treated in the same way as nominal word types. (As an interesting exercise try and describe 4-dimensional space without making reference to 2-D or 3-D Cartesian coordinate axes, the latter being generic instances of spatial representations that make useful stereotypes).

5.4.3.7 The Nature of Extensions and Vagueness

The treatment of the exact nature of intensions and theories of word meaning is a complex subject lying largely outside the scope of this thesis. A much simpler problem to analyse is the question of vagueness. This was considered earlier, section 4.3, in the context of uncertainty in simple knowledge based systems. It was argued there that the representation of uncertainty as point numerical measures was far from adequate in many situations. Further the semantics of fuzzy set theory in particular ran counter to basic intuitions about natural language in that self contradictions can have truth values greater than zero. Johnson-Laird also discusses several of these issues (p198-201,J-L, 1983), and arrives at much the same conclusion.

Johnson-Laird makes a major contribution, however, by pointing out that vagueness is in fact a desired solution rather than a problem in the context of formal equivalence categories. This claim requires clarification as follows. It is obvious that vagueness is very persuasive in natural language usage. Speakers or writers may be deliberately imprecise or vague when providing information, use misleading metaphors, figures of speech, and so on. There are two types of vagueness worth noting:

1. Extensional vagueness - as in "Is the exchanger expensive (safe, etc.)?", "Is John tall?".
2. Intensional vagueness - as in "Is a lamp an item of furniture?" due to the imprecise nature of the furniture Concept.

These situations contrast with extensionally precise aspects of language such as proper names which pick out or identify specific individuals.

Most concepts and relational predicates are not precise because if they were the category boundaries would have to be specified and remembered precisely. The cognitive demand this would place on people and the agreed social conventions that would have to be set up would make this a massive undertaking and one likely to defeat most people. The same would be true of any attempt to set up a knowledge

based system in a similar way.

The answer to this problem, John-Laird argues, is the way in which Concepts are represented and interrelated in a taxonomic structure. The boundaries of a word's extension are set by the nature of the particular taxonomy the Concept appears in. For example, what counts as a dog depends on the similarity of categories representing foxes, cats etc and the prototypes in use. As indicated earlier if the Concept involved is primarily described in terms of function, eg a table or chair, then many items can be treated in an equivalent way. This, of course, is exactly the basis underlying the notion of equivalence categories.
Some concept hierarchies are relatively well defined such as common animal classes. Other concepts are involved in more complex relationships. Johnson- Laird discusses a fragment of the semantics of "in, at, on and with" as a more typical example.

The advantage of having Concepts organised in some superordinate structure is precisely because it allows or yields vagueness in extensions without the need for any accompanying account of uncertainty. This means, for example, that speakers are not forced to be precise in cases where they are ignorant or they can communicate information at the level of "vagueness" appropriate to the situation. This mechanism allows Hayes (1974) to state that "the heap is small" without the need for numerical measures of uncertainty that he, and others, have argued against. Whether or not this is a relatively precise or vague statement will depend on the surrounding context of the conversation and the information the speaker intended to communicate to the listener.
These observations fit in with earlier ideas on conceptual categorisation and the use of equivalence categories. The whole emphasis there is to think or speak with maximum amount of economy appropriate to the situation. This implies that by default people will communicate at a relatively vague level by moving up the equivalence category structure until it is necessary to explain some point or other. This explanation may involve more precise details or it may simply make appeal to prototype connotations or refer to related Concepts in the taxonomic structure.
Resolving problems of intensional vagueness ultimately requires inventing new words or more precise predicates in order to cope with dubious concept exemplars or entirely new categories.

5.4.3.8 Summary: Word Meaning

The purpose of the discussion above on the nature of intensions and extensions in theories of word meaning was to clarify how several key aspects should affect the development of a knowledge representation system. As Johnson-Laird suggests questions such as "Are meanings in the mind or in the world? Is Realism or

Psychologism right? Do concepts have necessary and sufficient conditions or not?" appear to be false antitheses.

Closer analysis from a cognitive viewpoint reveals that the nature of intensions are not uniform across different word types. Some words require mental constructions to be imposed upon the world and are understood by both stereotype and part analytic specifications. Other words, e.g. natural kinds, refer to entities whose underlying structure is unknown or whose intensions are at best part describable by relatively few experts. Lack of definition of such concepts is no bar to their successful use in communication or reasoning, however, as they are principally understood via stereotypes anyway. There are also words that can be stipulated as necessary and sufficient conditions and support analytic type inferences. As Johnson-Laird puts it "Language embodies no particular metaphysics" it embraces both Realism and Psychologism".

The implication for knowledge representation systems is that the differences in interpretation of different Concept types have to be catered for. The representation must support the notion of a Concept, inter-related to other Concepts within a taxonomically organised structure, Concepts themselves having an internal structure and, more often than not, stereotypical meanings. The structure itself plays an instrumental role in determining the boundaries of word extensions and enhance the ability to readily use and interpret words in more or less vague ways for communication purposes.

5.4.3.9 Styles of Word Representation and the Effect of Context

In order to emphasise the cognitive aspects of word representation and interpretation Johnson-Laird discusses at length the inadequacies of three styles of word representation, namely lexical decomposition, semantic networks and meaning postulates. The problems he identifies are summarised here as they illustrate the important role that "context" has to play in interpretation mechanisms (Context is used here in the general sense of the context or background of a conversation).

Lexical decomposition, as a style of word representation, attempts to decompose words into sets of necessary and sufficient conditions. Consider the phrase "a man lifts a child". A simplified representation of this would be:

```
man   :  Human, Adult, Male
child :  Human, not (Adult)
lift  :  Cause(Activity X, Upwards (Move) Y),
         where X is the subject.

The phrase is represented as
Cause(Activity(Human, Adult, Male),
       Upwards (Move) (Human, not (Adult)))
```

This sort of representation is very similar to what is found in many natural language

or knowledge based systems.

If this was a valid representation it suggests that the decomposition process should be experimentally observable, more complex words taking longer to decompose than simpler words (e.g. using man instead of adult). Johnson-Laird points out that this effect, termed semantic complexity, has not been observed, (see chapter 10, J-L, 1983). What is observed is that complex sentences constructed using many connectives or relative pronouns are hard to follow. This point is returned to later.

Meaning postulates are assumptions which limit the set of model-theoretic interpretations of a language. They are used to stipulate required relations between words, ensure that proper names refer to the same individual in all possible worlds, etc. An example representation might be:

1. man MAN child CHILD lift LIFT Postulates: For Any X. If X is A MAN Then X is HUMAN and X is ADULT and X is MALE. For Any X. If X is A CHILD Then X is HUMAN and NOT (X is an Adult) For Any X and Y. If X LIFTS Y Then X CAUSES Y TO MOVE UPWARDS

Adherents of this approach assume that comprehension consists of translating sentences into a mental language which bears a very close resemblance to its surface form. Each morpheme in natural language has a corresponding unanalysed token in the mental language, e.g. man --> MAN The basic claim of the theory is that words cannot be adequately defined and are merely translated into tokens in the mental language.

Johnson-Laird argues that both approaches (as well as semantic networks, but these are more a representation style rather than denoting a theory of meaning) are inadequate to cope with the vagaries of comprehension. The fundamental flaw is that they fail to explain how expressions relate to the world, or models of it. Four examples are listed that make the point in different ways:

1. Ambiguity and Context - Johnson-Laird gives several examples of ambiguous phases, e.g. "They are handsome". Depending on the context, handsome might mean generous or good-looking. That is, the ambiguity is resolved once the referent of "they" is known. A selectional restriction, i.e. meaning postulate, is not placed on the meaning of "they" itself then. Rather, what needs to be constrained, in general are the referents of expressions.
   Predefined selectional restrictions also make it very difficult to cater for imagined contexts such as in nursery rhymes, e.g. "the dish ran away with the spoon". Yet people can comprehend such ideas easily. Imagined contexts are an important source of engineering models. A simple example is the use of "springs" to try and model fluid interactions in polymer solutions. Again the use of context is invaluable in resolving apparent anomalies in word comprehension.
2. Instantiation and Context - Context helps to narrow down the possible interpretations of words. This might be in the form of an assumption about a

more specific category type (e.g. "the exchanger is good for temperature overlap duties, assume a double pipe fin tube?") or the identity of a pronoun (e.g. "it is expensive", given the context assume the identity of "it"). What context provides then is a guide to the instantiation of specific referents.

3. Inferences and the Autonomy of Intensions - many approaches to word meaning have assumed the autonomy of intensions used in inferences, i.e. the assumption that meanings are independent of context or use. An extension of this approach is to assume that the occurrence of a word in a specific linguistic context somehow instantiates a more precise or specific sense of the term. Johnson-Laird argues that this basic assumption is false. Consider the simple sentence.
   "The turtle rested on a floating log and fish swam beneath it".
   Most people assume the fish swam beneath the log rather than between the turtle and the log. This effect is hard to account for on the grounds of instantiating more specific meanings - see below.
   Another simple example is the use of phrases such as "X is Y", "On X's right". Logicians or "meaning postulants" would have that the transitivity be stated in an autonomous axiom. However, sit a number of people round a circular table and the interpretation of "On X's right" becomes dubious. Requiring that higher order postulates be used that generate specific meaning postulates as a function of context dependent information clearly contradicts the assumption of the autonomy of intensions.
4. Deixis and Interpretation - The examples above indicate how the reference of some expressions play a role in determining the senses of other expressions, at least in any plausible theory of comprehension. Consider the sentence:
   "The bacon and egg is sitting at table 5 and getting impatient".
   This is a simple example of indirect reference as might be used in a restaurant. A reader makes sense of this by recovering the appropriate referent for "bacon and egg", ie not the plate of bacon and egg itself but rather the person who ordered it. A theory that assumed a set of literal meanings could first be assigned to a sentence and then context made use of to eliminate inappropriate meanings would not work in such situations. Similarly all conventional notions regarding selectional restrictions will fail in such examples. The obvious, and only solution, is to recognise that many acts of comprehension must be resolved by making context dependent inferences.

The examples above, hopefully, illustrate the vital effect of context on inferences. Such effects cannot be accounted for by theories of word meaning that fail to relate how expressions in a representation language relate to the world, or a model of it. The latter make the implicit assumption of the autonomy of intensions which turns out to be false in general.

Johnson-Laird states:

> Unless the retrieval of referents is taken into account, it is impossible to explain the resolution of lexical ambiguities, the instantiation of words in context, the vagaries in the logical properties of spatial relations, and a variety of deictic phenomena, (p242, J-L, 1983).

Many of the examples can be readily accounted for if it is assumed that a mental model of the relevant state of affairs was constructed that takes account of context dependent information. The procedures required to construct such a model are now

considered.

5.4.4 A Procedural Semantics for Mental Models

It was stated earlier that three types of mental representation were worth considering, namely propositions, images and mental models. The purpose of this section is to summarise the procedures needed to construct mental models based on proposition or image type representations. The treatment of image representations is not pursued further here but it is likely that much of what follows for the propositional case is true for images also in terms of the overall functionality of any knowledge representation system.

Based on considerable experimental evidence, Johnson-Laird suggests that a two stage theory of comprehension is required. The first stage involves a superficial understanding of a sentence or utterance (or what you see) that gives rise to a propositional (or image) representation. This propositional representation, constructed in some mental language using appropriate tokens, provides an economical way of representing discourse, especially for sentences involving indeterminate descriptions, e.g. "some of the exchangers are oversized". The richness of the tokens used in the representation will be comparable to that of natural language.

The second and optimal stage of comprehension is to make use of the propositional (image) representations as a partial basis for the construction of a mental model. The basis is only partial since the constructing processes must also make use of general knowledge (e.g. stereotypical information), contextual information, local constraints, etc.

In order to understand what is implied by this two stage process consider again the statement

"Some of the exchangers are oversized".

Any speaker of English will have no difficulty in grasping the meaning of the sentence at a superficial level, irrespective of whether they know what an exchanger is or not.

What, of course, is not grasped is the true significance of the statement, ie the significance of the proposition it expresses. The sentence is simply one that has been found in a thesis on representation. If the sentence had been read, however, in the context of a preliminary design report and it was inferred that the "exchangers" referred to one or more of the five proposed exchanger designs, the reader would be part way to grasping the "significance" of the statement. That is, recovering the

particular proposition the statement conveyed in its original context. The full significance of the statement, to the limited extent of what "full" can ever mean, also requires establishing the writer's communicative intention.

The two stage comprehension process is hopefully well illustrated by the example above. The first stage involves a superficial grasp of an utterance in some propositional representation. This representation will resemble the surface level form of the statement rather than a phonetic, say, transcription. The second stage, which is optional, brings in the theory of mental models, the significance of a statement being established by relating the propositional representation to a mental model. When the referents of a statement have been inferred the new information or descriptions conveyed by the statement can be added to the model, e.g. when "exchangers" was identified as referring to the proposed heat exchanger designs in a specific report.

There are a number of important points worth making about the whole process:

1. Is there any evidence that a surface level propositional representation is used or necessary? Johnson-Laird cites various researchers who have argued that this is the case. How else, for example, would sentences like the over-sized exchangers be handled out of context? Experimental evidence suggests that propositional representations can only be recalled over a relatively short time.
2. Recall seems, instead, to be very much more a case of active reconstruction based on what remains of a model of the original narrative or event.
   The whole treatment of euphemisms, for example, can also be understood if the distinction between surface level representations and more detailed models is maintained.
3. Constructing the propositional representation implies that some syntactic relationships between words must be recovered in order to represent the original statement.
4. Mapping of surface level representations into mental models requires that the basic intention of a statement must be at least established, e.g. is this a question, or a descriptive statement, or a request, etc. Establishing this intentional framework is essential for establishing context otherwise statements will merely perplex us or "pass us by", such as the first reading of the oversized exchangers.
5. Considerable evidence suggests that the interpretation of representations in terms of models will happen as early as possible. That is, it will happen at a constituent by constituent or clause by clause level rather than at the level of a complete sentence. Intelligent readers guess ahead as to what to expect next, continuously updating their model as they proceed.
6. Constructing a model from surface level representations clearly requires extra effort beyond handling the latter themselves. The process is not one of simple translation. It requires going beyond the surface level information since inferences are required that involve identifying or assuming referents, making use of generic Concepts and taxonomic information, use of prototypical ideas, etc. It is a constructive process, hence the term "constructive semantics" used in the earlier discussion on the nature of intensions. Constructive semantics is not a new idea. It is the whole essence of Conceptual categorisation, and has been developed by other workers, e.g. Bransford, Barclay and Franks (1972).

As regards the last point it is worth noting the basic difference between interpretive

theories and constructive theories. An interpretive theory is one that assumes that a semantic interpretation assigned to a sentence, often based on a full syntactic analysis of the sentence, provides a full analysis of its meaning. Constructive theories, on the other hand, hold that interpretations are made that infer beyond the linguistic information provided ("jumping to conclusions" is a commonly used phrase that is indicative of this approach).

Having briefly outlined the nature of the use of mental models on a trivial example it is now possible to state the basic assumptions of the theory and the types of procedure required for its implementation. For a full discussion of the following points see Johnson-Laird (Chapters 11-16, 1983).

5.4.4.1 Principle Assumptions

The theory of mental models is based on five simple assumptions:

1. "The processes by which fictitious discourse is understood are not essentially different from those that occur with true assertions". This assumption has important consequences for the role of surface level representations in the construction of hypothesised situations whether they be alternative process synthesis designs or fanciful stories. I object to the use of the word "true", however, and propose that "true assertions" be replaced by "assertions perceived to be non-fictitious". As shown later, notions of truth and falsehood have to be reconsidered in any cognitive theories of comprehension.
2. "In understanding a discourse, you construct a single mental model of it". This is a fundamental point. There are not lots of mental models constructed from which an appropriate model is chosen. A mental model is a single representative sample. The question of how to construct the "correct" mental model requires a non-deterministic mechanism, explained more fully later but also alluded to in the next assumption.
3. "The interpretation of discourse depends on both the model and the processes that construct, extend and evaluate it". The basic reason why a single model can be used as a representative sample of the infinite set of possible models is that it can be revised, or must require the capability of being so, in the light of subsequent discourse. Subsequent here may refer to a constituent by constituent level, or clause by clause level, or statement by statement level, etc. It is this revision process that simulates the non-determinism needed to satisfy the previous assumption. This assumption effectively rules out any assumptions regarding the anatomy of intensions and has major implications regarding the status and interpretation of any attempted "static definitions", e.g. meaning postulates, heuristics formulated as production rules, object definitions, etc.
4. "The functions that construct, extend, evaluate, and revise mental models, unlike the interpretation functions of model theoretic semantics, cannot be treated in an abstract way". This assumption, or requirement, implies that there must be explicit algorithms for computing the functions that map propositional representations into mental models. In general it is likely that these will be difficult to formulate. It is a fundamental point, however, on which the usefulness of the theory of mental models rests.
5. "A discourse is true if it has at least one mental model that satisfies its truth conditions that can be embedded in the model corresponding world". This long-winded formulation only applies to assertions which have definite truth conditions. It basically states that formal truth conditions, i.e. necessary and sufficient conditions, can be catered for when they arise. As argued earlier,

however, the whole question of what "necessary and sufficient" means in a theory of word meaning requires careful consideration.

The assumptions above all have major implications regarding the development and implementation of knowledge representation systems. Rather than try and discuss them all at once they will be introduced where relevant throughout the next chapter.

5.4.4.2 Basic Procedures

Johnson-Laird suggests that the essence of the theory of mental models can be boiled down to a, seemingly, simple set of procedures required for translating assertions into mental models or dealing with questions, requests, etc. The requirements listed here are modified from those given in the original text:

1. A procedure that can construct a new mental model whenever an assertion is introduced that neither makes a reference to any entity in the current model nor refers to the current context of the discourse.
2. A procedure which can recognise if at least one entity referred to in a statement is represented in the current model and, if so, add the other entities, properties or relationships referred to in the statement in an appropriate way.
3. A procedure that can integrate or combine information from two or more previously separate models if a statement is introduced that interrelates them (but see qualification later).
4. A procedure which attempts to verify whether the asserted or assumed properties or relations of the entities involved hold in the current model. This is referred to as the verification procedure.
5. A procedure to introduce or add an entity, property or relation in order to verify some statement for which there was previously insufficient information to determine whether or not it held, i.e. make an assumption, possibly only temporarily.
6. A procedure which can check whether the current model can be modified in such a way that renders an assertion "true" which is apparently "false" whilst maintaining the consistency of previous assertions in the model.
7. A procedure which can check whether an assertion is a valid deduction from previous assertions by testing whether the current model can be modified in a way that is consistent with previous assertions but renders an apparently true statement false.

A number of important qualifications must be made regarding the specific nature of these procedures.

The first procedure effectively implies that models and the entities comprising them can come and go as time or analysis of a problem proceeds. The existence of separate models is also recognised in the third procedure. The statement that information from models can be combined should not be taken to mean that there is some large repository of mental models akin to flowsheet case studies. Rather the context of the discourse is essential in establishing what is worthwhile creating as a "long term" model. Within that context it is then possible to cater for short term, "on the fly" hypotheses which may involve creating temporary, much reduced or simplified models, e.g. containing only one or two entities, to check out a particular idea. Any resultant information can then be acted upon and incorporated back into a main model

if necessary.
The verification procedure raises the tricky question of establishing whether or not a statement can be thought of as already being represented in a model or not. Unlike Johnson-Laird I am reluctant to refer to this as the "truth value of assertion" since what is understood by "true" or "false" in the context of mental models has yet to be discussed.

Whatever the answer, the last two procedures are intended to simulate the non-deterministic aspects of mental models by providing the mechanisms needed to revise a model to account for additional information. The latter procedures, if implemented exhaustively and without error, would constitute a decision procedure for any inference that could be used in a mental model. Obviously, they cannot be practically implemented as such since such decision procedures are, in general, undecidable. Further discussion is delayed until the nature of what is meant by truth is considered.

In general, the ability to grasp the significance of a propositional representation involves invoking one of these procedures as a function of its referents, the context represented by the current model, and "background knowledge" triggered by the tokens comprising the statement. (By background knowledge I mean stereotypical information drawn in by the association of a token with a Concept which itself resides in a superordinate structure with relationships to other Concepts). The verification and revision procedures are essential for inference purposes and depend wholly on the ability to interpret an existing model. This raises a fundamental point, which was pointed out earlier, that cannot be overemphasised. Namely, the way in which a model is interpreted is intimately bound to the intended function of the model: "a specification of structure and content must always be supplemented by an account of the processes using the model if one is to formulate what the model represents", (p158, J-L, 1983). This intended use is required precisely because many Concept definitions are inherently vague. An explicit idea of the intended purpose of a model provides a means by which background information can be qualified.
Consider the following problem fragment. A reference has been made to some triangle in a problem statement. Before hearing the details of the problem, e.g. triangle dimensions and angles, an entity is created in a model based on a prototype of this class, eg, it may be an equilateral or isosceles triangle but it does not matter as long as it has three sides joined up. Now, on further reading of the problem it is discovered that a right angled triangle, say, is involved. Verification at this stage does not involve a hard and fast check of the angles involved against our model entry resulting in logical falsehood, inconsistent information or a rejection of the information by the reader after much "thrashing around" as would occur in a simple

theorem prover. Instead, the status or intended purpose of the model entry is immediately recognised, i.e. to allow a start to be made on the problem, and easily modified to take account of the new and more specific information which, at this stage, there is no reason to question. If all the triangle dimensions were also provided and the reader was aware of the general constraints that govern triangle angles and dimensions he or she may well validate that information in a logical way if it was felt necessary to do so.

Although the example involves an abstract, formal concept involving necessary conditions it still illustrates in a limited way the intended nature of the revision processes involved in the interpretation of mental models. The word triangle need merely be substituted by separation column or feed preparation section to illustrate the relevance to process design problems. The important point is that successful interpretation of a partial model requires an understanding, either explicit or implicit, of the original context and purpose for which it was created.
Another more general point to note is that there will normally be a number of ways in which a statement can be interpreted with respect to a mental model. Consider, for example, the assertion

"Pump p101 is upstream of the reactor"

This information could be used to help establish which pump is called p101 on a process flow diagram, or which unit could be the reactor. Alternatively, it can be used to construct a description of the given state of affairs by introducing appropriate entities into a mental model. The particular interpretation or process involved will depend on the background context of the discourse in which the statement was made. This is an example of the point made earlier regarding the context dependent retrieval of referents needed to successfully construct models from propositional representations.

A practical, but completely fortuitous example of the need to take problem context into account and the vagueness of certain interpretations can be seen in the development of the rule based system for physical property method selection in chapter 3. In that system two mechanisms were described that attempted to take problem context into account, namely a "data compromise" and an "applicability compromise". It was clear at the time of development that the formulated rules should not be treated in a hard and fast manner. The mechanisms introduced to allow some flexibility into the interpretation of the rules can be seen as primitive attempts to cater for the types of inference required in revisable mental models.

The introduction to the procedures required for the construction of mental models, as described above, has been necessarily brief and simplified. For an extended discussion of the issues involved see Johnson-Laird (Chapter 11, 1983). In particular a programme is described that interprets spatial descriptions using procedures along the lines of the seven basic ones described earlier.

5.4.4.3 Other Aspects of Mental Models

Johnson-Laird's discussion on the nature of mental models touches upon many issues that are of long term relevance to the development of knowledge representation systems but cannot be discussed here for reasons of space, eg the treatment of grammar, parsing and the coherence of discourse in the context of mental models (see J-L, chapters 12-16, 1983). There are three basic topics however which must be mentioned:

1. Truth and Falsehood
2. Conceptual Primitives
3. Types of Mental Models

The notion of what true and false means in the context of mental models must be considered carefully since it impinges directly on any implementation of the verification or revision procedures required to construct, evaluate or extend a mental model. In propositional logic a formula is said to be contingent when it is consistent but not valid, ie true under some interpretations, false under others (see Appendix B). In cognitive terms a contingent assertion could be taken to be true if it "corresponds to reality". However, in the theory of mental models all that can be done is evaluate the consistency of an assertion in relation to some model of the world. Even here, the cognitive effort required to verify some assertion is likely to be very limited. The question of how to compare an assertion with reality is much more problematical. This is principally due to our fundamental incompleteness of knowledge of the world. A partial answer to the problem suggested by Johnson-Laird involves considering the model of discourse.

A mental model is simply a representative sample from an infinite set of possible models that at any one instant represents the currently believed state of affairs. The recursive procedures used to revise a mental model imply that any one of the infinite number of possible models could in theory be generated. It is this revisable aspect of the interpretation of mental models that, at first sight, seemingly causes considerable problems in providing any working definition of a truth value. The heart of the problem can be described in simplified terms as follows: a mental model can be taken to represent the extension of an assertion, ie the situation it describes, and the recursive revision mechanisms can be taken to represent the intension of the

assertion, ie the set of all possible situations it describes (see Johnson-Laird, p440 1983).

Now, earlier discussion indicated that for many types of word, eg natural kind words, the true intensions are unspecified, or only part specified, and that the Concept the word tokenises is primarily understood through stereotypical information. This in turn requires taking into account the internal structure of a category as well as its inter-relationships with other Concepts within a superordinate structure. The interpretation of a Concept therefore may constantly change depending on the context and purpose for which it is being used. If this is the case then in many cases it implies that the notion of some rigorous or objective truth value must be abandoned. Indeed the basic idea that the words true or false can be applied to inferences in a cognitive framework is an implicit assumption carried over from the assumption that some sort of mental logic exists. As discussed previously, the theory of mental models rejects such an assumption.

The absence of an analytic specification of intension does not, of course, prevent "intelligent" use or interpretation of a Concept. Indeed it is exactly this mechanism that is required for successful Concept attainment, ie simple learning. Johnson-Laird gives the example of a doctor observing some condition in a patient which he decides to call disease X, say. By introducing a new natural kind term to the language the doctor can communicate and convey to others the idea involved by describing stereotypical features of the patients involved. A mental model can still be constructed of the situation even though there may be universal ignorance of the intension involved. As investigation continues it may be decided that the condition be better classified as a number of diseases rather than a single one. In such a case neither the intensions nor extensions of terms are fixed.

In an attempt to summarise the situation Johnson-Laird states:

> If a discourse has complete truth conditions, it is true with respect to the world if and only if it has at least one mental model that can be mapped into the real world. If a discourse has only partial truth conditions, it is false with respect to the world if it has no mental model that can be mapped into the real world. If its truth conditions are not fixed or known then, to use Russell's aphorism about mathematics we never know what we are talking about, nor whether what we are saying is true. Indeed, we cannot know.

This is a very weak claim and says nothing about how to verify assertions. It does suggest that for successful communication of ideas represented in a model to others to take place requires an ability to compare a model with some other independent model, eg a textbook model of a column, say, that is widely agreed upon or a model

representing the perceived beliefs of another engineer. The ability to verify assertions is bound up in the implementation of procedures used to represent conceptual primitives. This was the second point in the list above and is now considered.

The need for Conceptual primitives, implemented as procedural primitives in mental model theory, is not surprising. The discussion on the nature of Attributes as being Concepts themselves in the context of Rosch's work indicated that such a mechanism was required to break out of the infinite regress that otherwise occurs. The earlier work of Bruner et al indicates that many concepts are treated at a perceptual level in real life and form the basis from which higher level, functional and formal equivalence categories can be constructed. Johnson-Laird (Chapter 15, 1983) also makes an appeal to the idea of innate Conceptual primitives. As in the case of Conceptual categorisation it is suggested that these primitives are perceptual in nature, cannot be easily verbalised and constitute the basic means from which mental models are constructed, eg create representations, imagine possibilities, etc. Conceptual analysis can therefore be treated at three levels: primitives, encoded as procedural primitives; simple Concepts, based directly on the use of primitives involving stereotypical information; and complex, part definable Concepts constructed in relation to other complex or simple Concepts.

In an earlier work, Miller and Johnson-Laird (1976) identified a basic set of Concepts which require a number of primitives to describe them. This set includes the Concepts of time, space, possibility, permissibility, causation and intention. Notions of time and space are closely tied to perceptual experiences of these Concepts. Procedural primitives representing these ideas are needed in order to reason about propositions involving temporal and spatial order, eg is earlier than, is right of, etc. In complex cases it may be useful to map spatial descriptions into idealised representations involving Cartesian coordinate systems but this is a constructive, formal process and should not be confused with the base level primitives involved.

Primitives based on possibility and permissibility are required to emulate the ability to construct models of imagined or hypothesised situations that are alternative to a current model. A permissible primitive is required to evaluate a model as is, ie without recursive modification, with respect to formal relational Concepts, eg basic principles or equations describing some subject area, in order to verify a model.

How to respond to attempted verifications will depend on the use of primitives based on intention, eg the intended purpose of a model, intended status or use of an entity in a model, etc. This in turn will subsequently affect the way in which a model is revised to take account of new or contradictory information. Primitives describing intention and causation will also be needed to implement the description of many functional

Concepts, eg to pump a fluid, since primitives are required to construct the assumed causal relationships between the entities involved, eg the entity used to pump the fluid and the fluid itself. They are also required to discriminate between the use of related groups of verbs eg to see, to watch, to view, to scrutinise, etc.
Many of the primitives will be directly concerned with the "constructive" aspect of semantics imposed upon the use of Concepts. Johnson-Laird discusses in detail one example of this, namely the concept of ownership, (see Johnson-Laird, p416-418, 1983). This Concept is purely based on social convention and illustrates nicely the notion of revisable truth conditions that depend on the amount of information that is known at the time of interpretation of some assertion involving the Concept.

The major gap in the theory of mental models, from the point of view of implementation of a knowledge representation system, is an indication of the precise nature of these primitives and their interaction with the seven basic procedures listed earlier for constructing, evaluating and extending mental models. The implicit assumption, however, is that such procedures can be implemented to enable the "truth conditions" of expressions to be evaluated in relation to a model of the world. If this cannot be done it will then not be possible to modify the model in a way deemed appropriate to the current context and model of discourse. The only "handle" available to determine how a model should be modified is the use of primitives used to indicate the intended purpose or function of model entities within the context of some verification procedure.

(An oversimplified analogy with predicate calculus might be that every formula in a model have associated with it a meta or higher level description indicating the status of the formula that is taken into account in any subsequent inferences. In that case default and non-monotonic logic would be special cases of such a logic). This mechanism is also required to cater for those Concepts which are part or fully defined in analytic terms as regards the way in which assertions involving such Concepts should be verified.

From personal reflection and the indirect evidence presented in Johnson-Laird it seems likely the logical rigour of this procedure will be very limited and probably restricted to simple forms of syllogistic reasoning and limited application of Modus Ponens.

The nature and use of such procedural primitives can perhaps be summed up in the following two statements. The first is that the notion of revisable truth conditions, which in turn is derived from the nature of Concepts in a taxonomic structure,

indicates that truth and falsify are not independent of the cognitive process involved. The truth of assertions and meaning of terms are, as it were, whatever you want them to be for a particular purpose. If a model is to be of any use in conveying information to others it must, of course, take account of recognised social conventions regarding the use of terms.

The second statement is that the theory of mental models requires three basic constituents: primitive procedures that are both ineffable and recursive in cognitive terms, propositional representations and models. The procedures are necessary for mapping propositional representations into models. They would also be used for relating images and models in a similar way. Primitives are also required to denote the construction of higher level Concepts or analytic specifications that may well be superficially represented in a meaning-postulate type form, eg the types of if-then heuristic used in rule based systems. The effective means of interpreting Concepts in a model indicates that in many ways it is more relevant to consider prototypes, for example, as procedures that specify prototypical information in a model rather than as some definite structure. In his discussion of Wittgenstein's use of schema and family resemblances, Johnson-Laird suggests:

> Hence, a schema is not an image, but a model that underlies the ability to form an image (p190, J-L, 1983)

In fact, in Rosch's later work (eg Rosch (1978)) the idea of a prototype is formulated as an operational definition based on judgements and reaction times so that categories do not have specific prototype members so much as a "more prototypical than" relation defined over the category members. Such procedural formulations make it clearer how the inherent vagueness of Concept descriptions arise.

The final aspect of mental models to be considered here is the typology of mental models that Johnson-Laird introduces. It is clear that mental models will contain tokens that correspond to entities in the world. The properties of these tokens and the inter-relationships between them correspond to the state of affairs represented by the model. Johnson-Laird introduces the notion of types of mental model in order to help specify the constraints that exist regarding what can be represented in a mental model. In particular, a distinction between "physical models" and "conceptual models" Physical models are described as corresponding directly to the physical world, they can represent perceptible situations but cannot represent either abstract relations or anything other than determinate physical descriptions. Six types of physical model are suggested:

1. A relational model - consists of finite sets of tokens representing physical entities, their physical properties and the physical relationships between the

entities.
2. A spatial model - a relational model in which the relations represent spatial descriptions.
3. A temporal model - a sequence of spatial "frames" occur in a temporal order corresponding to some sequence of events.
4. A kinematic model - a temporal model that is continuous in that it can represent changes and movements of entities in a continuous way.
5. A dynamic model - a kinematic model in which there are causal relations between "frames".
6. An image - a viewer based representation of the visible characteristics of an underlying three dimensional spatial or kinematic model, ie corresponds to a view of the state of affairs represented in an underlying model.

Although the model types seem intuitively obvious I find Johnson-Laird's claim that mental models are direct structural analogues of the world difficult to believe or understand in the context of temporal, kinematic and dynamic models. In addition, I would argue that the use of causal relationships is likely to be of a limited nature in practical situations. Many inferences in problem solving situations have to be made in the absence of well defined causal models, eg medical diagnosis, business predictions, etc. Even in more well defined situations, eg a chemical process description, an engineer's working definition of the causal relationships involved will only be of a simple or superficial nature. The notion of an image however as a projected view of an underlying model is likely to be one of significant long term interest to the development of a knowledge based system that wishes to convey or interpret information in a graphical form. Johnson-Laird introduces four types of "conceptual model" to represent more "abstract matters":

1. A monadic model - employing one place predicates to denote properties of tokens and binary relations to denote identity and non-identity between tokens. An extra notation is also required to denote the uncertainty of an entity of a particular type existing.
2. A relational model - introduces a general relational model involving the tokens in a monadic model. It also contains the mechanisms needed to cope with quantifiers, indefinite and definite expressions, proper names, etc.
3. A meta-linguistic model - this allows abstract relations between elements in a model, including a meta-linguistic model, and tokens corresponding to certain forms of linguistic expression to be represented (see below).
4. A set-theoretic model - introduces mechanisms needed to reason abut tokens denoting sets and the relationships between these tokens.

These model types merely cover the intuitive ideas implied in the rest of this discussion on the mapping of propositional representations into mental models. The meta-linguistic model does revise some interesting properties of mental models. The abstract relations involved include essential interpretative ones such as "refers to" and "means". For example, the linguistic structure

One of the X is Y

will have the following general model form:

```
          x
'Y' -->   x
          x
```

The quotation marks indicate a token representing a linguistic expression and the arrow denotes the relation of reference to entities existing in the model, assuming x refers to previously entered entities. The meta-linguistic model definition is recursive and means that one model can be thought of as being embedded in another. This is extremely important as it means that propositional attitudes, eg "I suspect that...", "I believe that he thinks...", etc, can be represented by nesting beliefs within mental models (see p430-438, J-L, 1983). It also provides a means to represent the notion of logical truth in an explicit way. For instance if P is some symbolic expression that is taken to be "true" can be represented as"

```
'P' --> model of world
```

where the arrow indicates the truth relation and the outer model relates P to another model corresponding to some required state of the world. It is then a simple task to represent an assertion of the form

"P is true" means that P corresponds to the world

as shown below, where the double arrow indicates the relation "means":

```
'P is true' ==> P  --> model of world
```

This example indicates the sort of meta-level "tricks" that can be used to implement some of the conceptual primitives referred to earlier.

Apart from helping to identify the types of relationships involved the developments described later will not attempt to follow or adhere to any strict distinctions between Johnson-Laird's model types. There are two basic reasons for this decision

1. A typology of models seems premature at this stage of analysis.
2. There is no apparent reason to suggest that a formal discrimination between model types is necessary for their interpretation. Rather the information in the different model types appears to be different facets or viewpoints that could easily be embodied in a single model.

5.4.4.4 Summary

The theory of mental models affects a large number of philosophical and cognitive research issues. The introduction here has only touched upon these problems in a necessarily brief and superficial way. Nevertheless, many of the important problems that will have to be directly addressed by a more sophisticated form of knowledge representation and inference system have been described. The following quotes summarise the main points of the preceding discussion:

> A mental model is a single representative sample from the set of models satisfying the assertion. .... Comprehension normally leads to just a single model, which is constructed by the procedural semantics from what is known about the truth conditions of an assertion. ... If a subsequent assertion shows that the particular model is incorrect, then recursive procedures attempt to reconstruct the model so as to satisfy the current set of assertions, (see p264, J-L, 1983).

Perhaps the most important part to be stressed regarding this entire discussion is as follows:

> ... the significance of an assertion depends on both the model and the procedures for manipulating and evaluating it, (see p264, J-L, 1983).

This is a fundamental statement that cannot be overemphasised. It implicitly rules out the assumption of autonomy of intensions, and hence simplistic knowledge representations based on that assumption, implies the use of revisable truth conditions and context dependent information, and requires that the intended function or purpose of models and their constituent entities be accounted for in subsequent inferences.

5.5 Overall Summary

This chapter has introduced many new ideas, some representing whole areas of cognitive science research, that are considered to be central to the development of a more sophisticated knowledge representation and reasoning system. The first topic covered was the initial attempts within AI research to establish the importance of the use of epistemological level relationships to more properly specify the internal structure of Concept representations, see section 5.1. The analysis, principally due to Brachman (1979), indicates that research on knowledge structuring primitives and their semantics is required in order to more precisely define the nature of higher level conceptual and linguistic constructs. In order to demonstrate the concept Brachman provided a simple set of primitives in the KL-ONE system that could be used in the definition of restricted frame-like objects in an inheritance hierarchy. Given the review in chapter 3 concerning the confused semantic status of many frame based systems, or rule based systems, and the likelihood that they could not be used in their current form to satisfy the demanding requirements of an integrated process design environment it was decided to investigate relevant research topics in the area of cognitive science. In particular the theories of Conceptual Categorisation, prototypes, and mental models were discussed. It is clear that such research has much to offer the study of AI. Of considerable importance is the contribution that can be made regarding the nature of concepts and their use in everyday thought. This work can therefore be used to provide guidance for the development of suitable knowledge

representation primitives at various levels, epistemological, conceptual, etc.

There are certain key features worth summarising from each of the areas of conceptual categorisation, the use of prototypes, and the theory of mental models.

Conceptual categorisation established an overall framework in which the process of concept attainment can be achieved. This framework emphasises the interactive, ie social aspect of such an endeavour and the importance of the effect of validation in various forms on the overall process. Conceptual Categorisation theory also makes various claims regarding the nature of Concepts. Equivalence categories and Identify categories are identified as basic forms. These range from highly perceptual, pre-verbal notions to abstract formalisations. The notion of an equivalence category is fundamental to many inference processes. That is, the natural tendency is to treat discriminably different things in an equivalent way in order to simplify the reasoning processes involved and reduce the cognitive effort required to cope with the large amounts of information available at any one time. Equivalence categories can be treated as one of two basic types, either functional or formal. Formal equivalence categories can be further analysed according to their basic mode of attribute combination, conjunctive, disjunctive or relational. The difference between criterial and non-criterial attributes and the flexibility of choice of which Attributes come under the criterial heading under different circumstances is recognised. The use of equivalence categories strongly suggests that Concepts are organised in some superordinate structure. Further it seems clear that there are many links between low level, "common sense" notions of Concepts and events, eg force, and higher level abstractions of the same Concept. The nature of the two levels is likely to be considerably different. Within the hierarchical Concept structure the use of typical, ie averaged, instances and generic, ie idealised, instances appear to be useful inference mechanisms. In general processes involving attribute reduction are important for reasoning with equivalence categories.

The study of prototypes and family resemblances has most to offer in the area of investigating how hierarchical Concept structures are constructed. The findings are compatible with that of Conceptual categorisation but emphasise the importance of the internal structure within a category. The idea of structuring categories within a hierarchy based on simple common equivalences is superceded by an analysis based on the use of family resemblances in which there may be no common elements between sub-categories in terms of surface level form. In many cases the common or binding element is one of intended function. Research in this area has also indicated the importance of the use of prototypes, more generally stereotypes, in understanding

Concepts existing at the basic kind level. Many natural kind words occur at this level. Further linguistic research has shown several basic word types, eg primary, natural and nominal can best be treated by recognising the difference in contribution to word meaning arising from the use of stereotypes and part analytic specifications, either weakly or strongly analytic. As with Conceptual categorisation, then, it appears there is no one simple way in which Concepts can be interpreted or said to be understood.

Finally, the theory of mental models puts forward a theory of comprehension that implicitly caters for most of these ideas although they are not explicitly referred to in the original text. By claiming that mental models are constructed by a mapping process making use of propositional representations, the theory of mental models also encompasses all five of Brachman's levels of knowledge representation, from implementation through to linguistic. The theory of mental models offers many ideas of potential importance to AI not least of which is the analysis of the nature of both intensions and extensions. This emphasises the importance of the use of context and a model of discourse in the retrieval of referents. The theory suggests that a procedural semantics be adopted for the interpretation of mental models that obviates the need to assume the existence of a mental logic in order to make valid inferences. Mental models are recursively modifiable which once again emphasises the need for "revisable truth conditions" when considering the interpretation of a Concept. In order to properly understand the significance of assertions it is necessary to account for both the intended function and purpose of constituent entities in a model and the procedures used to manipulate and evaluate it. As a consequence the autonomy of intensions can, in general, no longer be assumed.

The most obvious conclusion from this study is that for many Concepts knowledge cannot be represented in a context independent or "static" form. This, of course, is exactly what many implementations of rule and frame-based systems effectively purport to be, eg heuristic rules defined in isolation, frames using fixed Attributes and relying on common element construction via inheritance, etc. Since the actual process of attempting to write such rules or frames is simply an attempted linguistic expression of mental Concepts, which by their very nature are inherently vague, it seems unlikely that reasoning systems based on context independent and fixed interpretation algorithms will be capable of making the sorts of inferences required in an integrated process design environment. Johnson-Laird (p262, 1983) makes this point in an indirect way when he suggests that meanings of words in some object language will have no simple, let alone one to one, relation with that object language.

The conclusion, then, is a controversial but unavoidable one if the ideas presented in

this chapter are taken seriously.

Chapter 6
Development of Knowledge Representation Ideas

This chapter describes the preliminary development of knowledge representation techniques based on a cognitive or mentalist approach as alluded to in the previous chapter. At the time of development the potential size and difficulty of the task was not underestimated. As stated earlier, the research was undertaken with the view that if at all successful then the basic approach will have much more to offer process engineering developments in the long term than current knowledge based system.

6.1 Outline of Representation Requirements

In order to allow a start to be made on the problem it was decided to select a restricted set of features taken from the areas of Conceptual categorisation, prototypes, and mental models and investigate how these might be modelled in a system, albeit in oversimplified ways in the first instance. The features studied fall into two main areas. The first concerns the use of Concepts and their describing Attributes within some sort of taxonomic structure. The second area concerns the use of what I have termed a functional model (the word "functional" is used given its dictionary definition - "designed with special regard to purpose and practical use: serving a function"). Functional model is used in preference to mental model in order to emphasise the limited approximation of the former to the theoretical requirements of the latter. The specific items of interest regarding Concepts are as follows:

1. The difference between functional and formal equivalence categories should be handled.
2. The internal structure of a Concept should be studied, including different modes of Attribute combination in formal equivalence categories.
3. The use of stereotypical information should be catered for.
4. The distinction between criterial and non-criterial attributes should be maintained in concepts where part analytic specifications can be used.
5. The notion that a Concept definition is variable in nature dependent upon the context in which it is interpreted must be modelled.
6. The idea of inter-related Concepts within a taxonomic structure based on family resemblances should be modelled.

In many ways the most important of these requirements is point five since it implicitly assumes the need for a revisable functional model. All of points one to six require that suitable representation primitives be developed to implement these ideas.

The specific item of interest in the area of functional models is the investigation of how to implement the seven basic procedures needed to construct, evaluate and extend mental models. These procedures will be wholly constrained by the sophistication with which Concepts can be modelled. Indeed the implementation of

the two areas are intimately bound up in one another.

There are two final points to be made that will be seen to greatly enhance the potential use of such a knowledge based system. The first point concerns the use of formal algorithms to solve idealised problems when implemented in some "external" way. It is clear that in process engineering design many problems involve by modelling unit operations as sets of equations that can then be "solved" by some iterative technique. The technique itself, eg, Newton's method, the secant method, flash calculation methods, etc., has the status of a formal, abstract, relational Concept with part-analytical specifications. It is these formal aspects that are described in texts on numerical analysis and learned by process engineers. The intended use of the Concept is often described in terms of some generic function relating to notional variables "y" and "x". Having grasped the stereotypical behaviour or connotations involved most engineers are ready to accept that the technique can be implemented "externally", ie on a computer, but still be used to reason about entities in a mental model, ie unit operations in a process model. Concepts such as Newton's method can then be linked to functional concepts such as "things to solve equations with" and reasoned about as task variables. This means that efficient implementations of numerical algorithms can easily be accommodated within the overall system structure. It is also worth noting that this description seems to describe well the actual, or at least perceived, situation that exists in solving real problems.

The second point is similar to the first in that it concerns the explicit use of logic, and in particular various types of theorem prover, to help solve problems. In exactly the same way as engineers can think of entities in a mental model being represented by a set of equations in a particular context and using algorithmic procedures to solve a given problem, so too can entities be mapped to a set of logical formula of various types, eg propositions, predicates etc, and be used within an "external" theorem prover to verify a particular hypothesis. This is because the status of logic, in a cognitive framework, is reduced to that of a task variable like any other formal abstraction one wishes to make use of in a problem solving environment. This situation undoubtedly appears to be the case in real life. People can read and grasp the ideas involved in the formalisations of many types of logic (eg propositional, first order, non-monotonic etc) but few, if any, could claim to be able to make rapid inferences using the actual algorithms proposed for theorem provers. As in the case of numerical algorithms, the use of logic when implemented as an explicit abstraction is not ruled out in the proposed system.

The use of formal abstractions in a problem solving environment is an important

general principle that can be accounted for in a theory of mental models. This is well discussed in Levy (1987). It generally involves the following steps:

1. A problem is recognised to exist. This will be evident from the model of discourse. There then exists the question of whether there is any motivation to solve the problem.
2. By some recognition procedure, an inference technique is selected that may involve some abstract formalism.
3. The problem is then "mapped in" to some appropriate form.
4. This form is then manipulated to attempt to achieve some intended effect.
5. The form is then "mapped out" into a mental model to infer what this may mean in real world terms. This will often involve communication to update the model of discourse
6. If the consequences are not liked or believed it may be necessary to return to the original expression of the problem and try some other approach.

The whole mapping approach is exactly what was referred to at the start of section 5.4 regarding Johnson-Laird's use of Craik's model of thinking. The problem with logic, or any other like formalism, is that it is distanced from language and comprehension in a way that ignores the mapping in and out processes, the selection of possible mappings and the use of non-linguistic formalisms.

In summary then the primitive$ procedures developed for interpreting functional models should recognise the need to be able to map a Concept into a required abstract formalism.

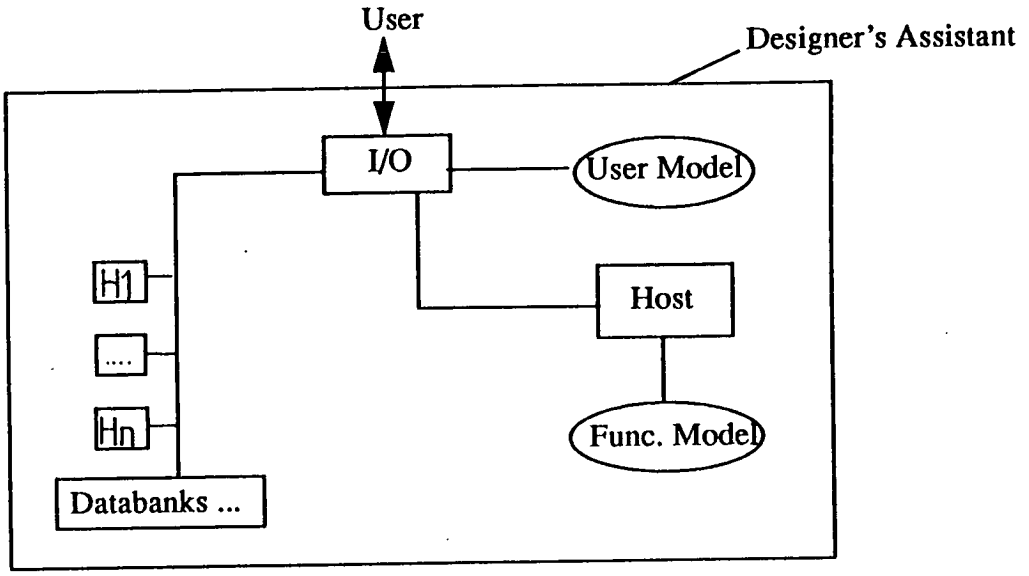
6.2 System Structure

The following sections describe the development of Designer's Assistant - version II. Due to the novelty of many of the ideas much of the research proceeded in an iterative manner, most ideas being refined or abandoned in the process. Rather than try and describe these ideas immediately, it is easier to describe the overall structure of the system that emerged from a programming point of view. A number of terms will be used however, eg Topic, whose meaning will not be fully discussed until later.

Given the probable size of the system that would emerge, it was decided to organise Designer's Assistant in a modular fashion, that is, closely related concepts would be partitioned into modules pertaining to specific subject areas or Topics. It was not originally envisaged that such a partitioning should have any assertional import or effect on reasoning and was purely intended for programming convenience, ie one module being transparently available to another. As it turns out, the proposed system structure and use of Topics can be used to good effect in primitive reasoning mechanisms.

An overview of the system structure from a user's point of view is shown in Figure 6.1.

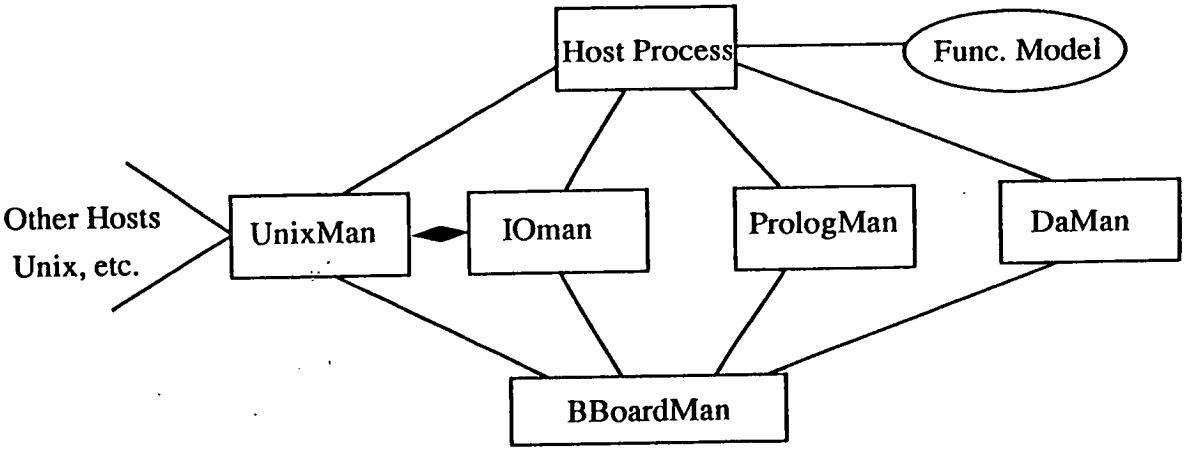
Figure 6.1: User's View of Designer's Assistant



The items on the left of the figure correspond to modules used to implement specific Topics. Also indicated on the bottom left of the figure are "external" utilities and databanks. The former is used to indicate any external, executable code, eg the NAG library of numerical algorithms. The latter is used to indicate external data storage, eg physical property data, that can be interpreted in a suitably appropriate way within a module. The item labelled "Host" represents a host module which, from the users point of view, can be thought of as the part of the system dealing with the overall problem conveyed by the user through the "input/output" system. The two other aspects of Designer's Assistant that the user will be aware of are the functional model and the model of discourse. The functional model can be thought of as the process model which is interpreted by whichever module happens to be acting as host at the time. The model of discourse shall later be shown to be essential for the system to be of practical use. It is interpreted in the same way as a functional model but is primarily concerned with modelling conveyed intent and purpose. This involves a form of user model (how the host models a user), a "self" model (how the host models itself) and a representation of the discourse between the two. These ideas will be explained in more detail later.

The actual implementation details of any module are better described by Figure 6.2.

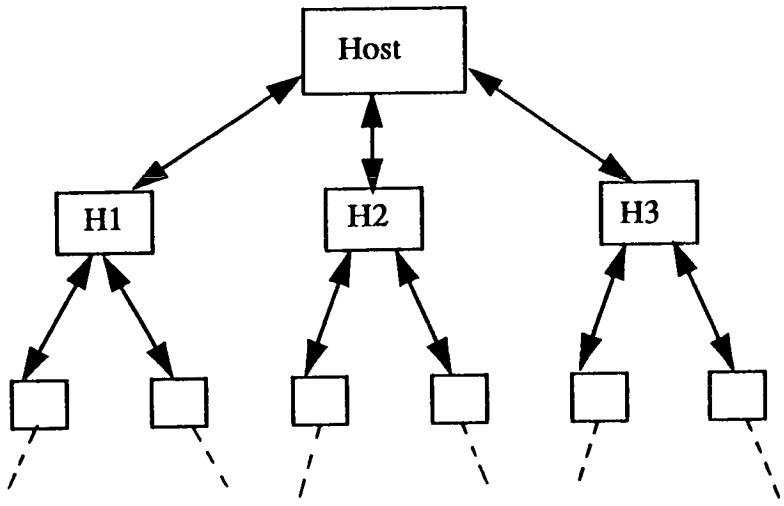
Figure 6.2: A Module Within Designer's Assistant



There is no one inference algorithm, as such, to interpret a functional model but rather a large set of utilities that attempt to implement the required functionality of the primitives used in the model representation. Some of these primitives are functional in the sense that they perform some intended function, eg $set, $modify (system primitives are indicated by a preceding dollar sign $). Several of these primitives make use internally of a system of blackboards as shown in the diagram. The blackboards are used purely for convenience of data storage and are not involved in inferences other than for the storage and retrieval of "data", both external and internal. Their operation is described in section 6.2.1 below.

From the system point of view and the interpretation of a single functional model the structure of Designer's Assistant is represented by Figure 6.3.

Figure 6.3: System View of Designer's Assistant



Each module is simply regarded as a set of procedures capable of creating evaluating or extending a given functional model. From a system point of view it is irrelevant that these procedures have been partitioned into modules. The figure indicates that any one module may invoke another if required. The system is written to ensure that infinite interpretation loops are trapped and terminated. The implementation is such that individual modules can be invoked in a distributed manner if desired, ie invoked in parallel for suitable reasoning tasks.

6.2.1 Use of Blackboards

The use of a blackboard system for internal storage and retrieval mechanisms was chosen simply to provide a clean, modular and consistent way of implementing system level primitives, eg accessing input typed by a user. In the current version of Designer's Assistant there are four specific blackboard managers and one generic manager. The generic manager, labelled Board-man in Figure 6.2 is responsible for initialising and managing the other four managers, eg initialising work space etc. It also looks after the work space required for a host's local copy of a functional model, or any embedded models, as well as handling "raw" input/output data. The other four managers are concerned with different aspects or limitations of the system that have to be represented. As indicated on Figure 6.2 these are:

1. DA-man - Designer's Assistant internal manager that is used to store data relating to the internal status of the functional model interpretation procedures, eg noting when interpretation "events" start and finish. Such low level information is not normally explicitly represented in the model of discourse as it would become extremely cluttered and difficult to reason with. This information is still explicitly available to the interpretation routines using this mechanism.

2. ID-man - All input and output between a module and the rest of the system, meaning another module or the user, is passed via the ID manager. This manager uses input/output channels allocated by the Unix Manager.
3. Unix-man - the Unix manager is used to provide a standard way of accessing or allocating operating system dependent facilities. This includes providing communication sockets, accessing files, loading external executable code, etc. Most of this is simply achieved by using the facilities of the base implementation language Prolog.
4. Prolog-man - the Prolog manager is used to keep track of memory limitations and perform base level Prolog operations "outside" the model procedures. This feature was intended to help isolate the implementation specific details from the more generic parts of procedures.

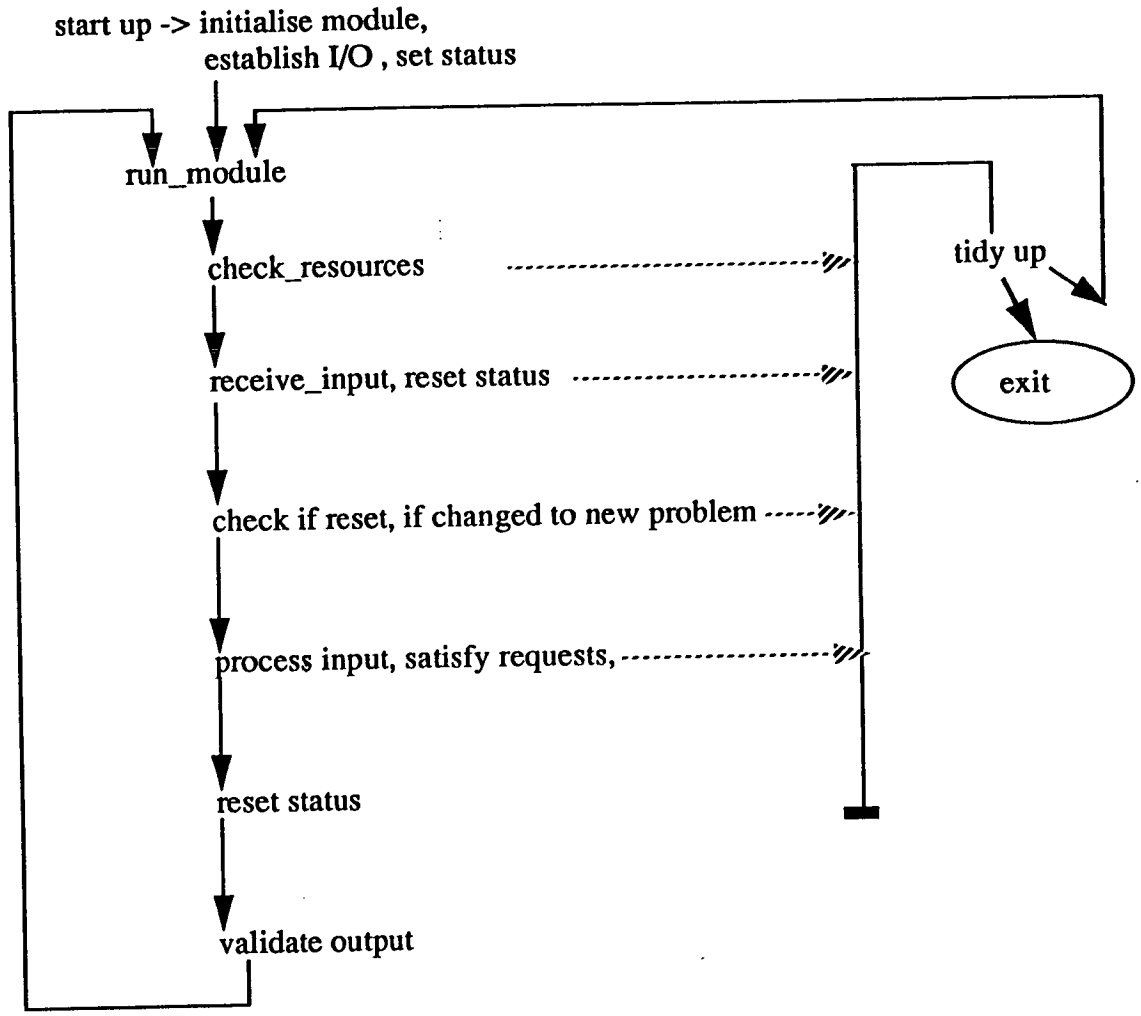
All of the blackboard managers operate in exactly the same way. In the case of data storage a command argument is supplied along with Topic and Module information and the relevant data. The command used is one of the functional primitives of the representation language eg $add, $modify, etc. The same applies to data retrieval except that the data argument can be a variable (a simple retrieve operation) or a Prolog structure either part or fully instantiated (to provide a checking mechanism).

In the context of Conceptual categorisation the blackboard facilities are considered to represent perceptual level responses that are outwith the reasoning capabilities of the system. This could easily be modified if necessary by describing these operations in suitable functional concepts (see section 6.3.2 later). As it stands the use of blackboards is upwardly compatible with the interpretation of the functional model since the same functional primitives are used in each case to achieve some intended effect.

6.2.2 Module Operation

This section briefly outlines the top level implementation of any module within Designer's Assistant. Since each module corresponds to one or more Topics, a Topic representing a structure of related Concepts, the steps involved simply mimic the iterative loop involved in the Concept attainment process, see Figure 5.2. These steps are shown in Figure 6.4 below.

Figure 6.4: Top Level Loop in Designer's Assistant



The initial step outside the loop is required when a module is loaded into Designer's Assistant. A number of calls are made to the Bboard manager and the DA Manager. The Bboard manager initialises the other blackboard managers and notes the name of the functional model it has been given to work with. The name is a simple string identifier used to distinguish between temporary functional models. A module can be passed a copy of an existing model, presumably to investigate some functional Concept or establish an embedded model, or it can start from scratch. Copying functional models is done merely to simplify the task of keeping track of temporary model revisions. If some intended effect is not achieved then it is simpler to delete the revised model than undo changes to a starting model. The main procedure, run-module, iterates over a small set of high level calls. The first two steps involve low level checks with the Prolog manager and DA manager regarding resources and

the internal status of the interpretation procedures. The third call accesses any "input" and interprets this with respect to the functional model. This is an important step and corresponds directly to the mapping of surface level propositional representations into a mental model.

If this input phase succeeds and any requests are pending then an attempt is made to process these requests, updating the functional model as necessary. If any of these requests succeed then an attempt is made to convey the information back to the user in a suitable form, propositional assertions in the current version. If anything untoward happens in this process then the system exits to a common point where it can be decided how to proceed. It is at this point the feedback loop required for concept attainment can be implemented. The other alternative shown involves halting the module and returning to the calling process. If the intended effect has been achieved then a revised functional model can either be merged with or replace an existing one.

6.3 The Development of Concepts

The previous section described the overall framework of Designer's Assistant. The framework and the intended mode of operation provide the context in which a more detailed discussion of specific aspects of the system can take place. The purpose of this section is to introduce the techniques that were developed to represent Concepts. The original intention was to satisfy the requirements for Concept representation as given in section 5.5.

There are two important points to keep in mind concerning the remainder of this discussion :

1. By viewing Concepts within a cognitive theory of comprehension it is impossible to divorce the representation of a Concept from its interpretation. This is due to the basic nature of Concepts, the inherent vagueness involved, alternative means of interpretation, context dependencies, etc.
2. The notion of a Concept should not be confused with any intuitive ideas about A.I. type frames or objects. In particular there is a recognition of the different types of Concept involved and the fundamental differences in the ways which they can be interpreted. There is no assumption of a fixed set of describing slots. Rather Concept definitions are taken to be variable in nature in a way that allows for context dependencies. In addition there is no implicit use of slot or attribute inheritance within a taxonomic structure of Concepts. In Rosch's terminology Concepts at the superordinate and basic levels are best treated from a family resemblance approach. Inheritance is more suited to describe the relation between subordinate and basic level Concepts.

With these points in mind the remainder of the section is structured as follows :

1. Outline description of formal Concepts in a declarative form. The representation

presented is an attempt to provide the necessary hooks or links needed to reason about formal equivalence categories.

2. Outline description of the representation format and the use of functional Concepts.
3. Description of the development of the primitive procedures and utilities needed to interpret such Concepts in the context of a functional model.

6.3.1 System Level Concepts

The treatment of formal equivalence categories is most easily introduced by describing the set of base level Concepts that can be used to describe any other Concept. Rather than give recursive definitions of these meta-type terms, since there are no lower level or more primitive Concepts, the definitions involve system primitives that indicate how the Concept is to be interpreted, (e.g. "$type(internal)"). This is an important point as it turns out the process of Concept attainment requires the ability to reason about such ideas in an introspective or self-reflective way.

The simplified definitions of several Concepts are shown below in Figure 6.5 These include the notion of an object, an attribute, adjectives, quantities, values and names (i.e. proper names).

Figure 6.5: Base Level Concepts

```
concept(object,object,formal,conj,any) :-
  [
     $ref O,
     $context       ($given object-O),
     $nature        [abstract],
     $structure     [it $can_be defined],
     $def_atts      ('$BASE' - ( $type(internal) $of O $is object ) ),
     $non_def_atts  [attribute],
     $sub_types     [physical,mathematical,abstract]
  ].


concept(attribute,attribute,formal,conj,any) :-
  [
     $ref [Att,O],
     $context    ($given object-O),
     $nature     [abstract],
     $fnctnl     [Att $describes O],
     $structure  [it $cant_be defined],
     $def        ( Att $has_a value ),
     $sub_types  [adjective,quantity]
  ].


concept(adjective,attribute,formal,disj(context),any) :-
  [
     $ref [Adj,Choice-(O $or Q)],
     $context ($given object-O $or quantity-Q),
     $nature [abstract],
     $fnctnl     [Adj $describes Choice],
     $def        (none)
  ].

concept(quantity,attribute,formal,conj,any) :-
  [
    $ref [Q,O],
    $atts_from  [attribute,attribute,formal,conj,any]
                - [[Q,O],[]],
    $sub_types  [physical]
  ].


concept(value,attribute,formal,conj,any) :-
```

```
  [
    $ref [V,Att],
    $context    ($given attribute-Att),
    $nature     [abstract],
    $fnctnl     [V $represents Att],
    $def        (none),
    $adjs       [known,unknown]
  ].

concept(name,attribute,formal,conj,any) :-
  [
    $ref [N,Obj],
    $context    ($given object-Obj),
    $nature     [abstract-attribute
                   $but treated $as_a linguistic-object ],
    $fnctnl     [N $represents Obj],
    $def        (none),
    $adjs       [known,unknown]
  ].

concept(variable,attribute,formal,disj(context),any) :-
  [
     $ref [V,Choice-(O $or Q)],
     $context    ($given object(mathematical)-O $or quantity-Q),
     $nature     [abstract-attribute
                    $but treated $as_a linguistic-object],
     $fnctnl     [V $represents Choice],
     $def        ( $equiv_to name $of Choice ),
     $sub_types  [$any - object(mathematical)],
     $adjs       [known,unknown]
  ].


/* ================================================ */

concept(known,adjective,formal,disj(context),any) :-
  [
     $ref [K,Choice-(A $or V)],
     $context    ($given attribute-A $or variable-V),
     $nature     [abstract],
     $fnctnl     [K $describes Choice],
     $def        ( $if Choice $has_a value
                       $then $note Choice $is known
                 )
  ].

concept(unknown,adjective,formal,disj(context),any) :-
  [
    $ref  [Uk,Choice-(A $or V)],
    $context    ($given attribute-A $or variable-V),
    $nature     [abstract],
    $fnctnl     [Uk $describes Choice],
    $def        ( $if Choice $has_not_a value
                      $then $note Choice $is unknown
                )
  ].


concept(system,object,formal,conj,any) :-
  [
     $ref  S,
     $nature     [abstract],
     $structure [S $consists_of collection $of objects - [O|Rest] ],
     $inter_rels [ relationships $between objects-[O|Rest] $in S
                     $can_be defined ],
     $def_atts   ( see-structure $and see-relationships )
  ].
```

The base level constructs have straightforward interpretations. An overview of these definitions is now given. Further details on the use of specific terms in a Concept description are given later.

An object, i.e. some entity, represents any formal Concept that can be reasoned about. Instead of using one generic type the category subtypes physical, mathematical and abstract are used. Physical entities refer to those which have direct physical correlates and include both natural kinds and nominal kinds. Other entities are considered to be abstract in nature, i.e. abstract, mental constructions imposed upon the world. For example, the notion of an object itself and its describing attributes are abstractions. The idea of a set, i.e. a grouping of entities, is an abstract Concept. So too are important Concepts in process design such as a control volume, a system and its surroundings, etc. Abstractions often correspond to idealised representations that can be used to simplify the analysis of problems.

A distinction is made, somewhat arbitrarily at this stage, between abstract Concepts that are mathematical in nature, e.g. equations, numbers, etc., and other abstract forms. The principal reason for this distinction was to recognise the importance of analytic specifications in these sorts of Concepts. A similar case could be made for terms used in the field of logic. Mathematical descriptions also involve special notational forms that can be interpreted in a more precise way than other linguistic forms and are somewhat unique in this respect compared to other Concepts.

Objects are part described by attributes which are either taken as criterial or non-criterial (corresponding to $def and $non_def_atts in Figure 6.5 above). Criterial attributes cannot, in general, be taken to mean necessary and sufficient conditions. Depending on the context and intended purpose at the time they will be interpreted in different ways. Attributes themselves are further specified by "adjectives" and "quantities".

Adjectives are a catch all means of object description and refer, often implicitly, to the value of some attribute, e.g the adjective "red" and the attribute "colour" of some object. Quantities are reserved for measurable attributes and are of particular importance are physical quantities, i.e. quantities involving physical phenomena.

The distinction between adjectives and quantities is important in cases where the same base attribute can be described in different ways. Consider the problem of handling assertions such as "X is small". In a functional model this is directly represented by the adjective "small" along with context information. There is no need to create some fictitious uncertainty factor. If a situation arises that requires some comparison of values regarding the implicit attribute "height", say, the the

stereotypical information can be used to relate the adjective to the quantity involved from which a direct (numerical) comparison can be made. The use of stereotypes and the intended purpose of in a given context has then to be noted if any "sense" of the comparison is to be made at a later date.

The application to process design situations is obvious if "height" is replaced by "value", say, with an associated "cost" quantity.

Adjectives can, in general, be used for any sort of descriptive attribute irrespective of whether it can be described in terms of a measurable quantity. It is important for both forms to co-exist to allow vagueness to be expressed in suitable forms. Even when precise values are known (e.g. the cost of a reactor) the adjective form of description will still tend to be used in object descriptions for communicative purposes (e.g. it is expensive, cheap, etc.). Quantity type descriptions tend to be used only in those situations that require the measured values to be enumerated.

The "value" of an attribute can be used to refer to either the adjective used or the measured quantity.

The use of "name" as a general attribute refers to the association between a proper name and the object or entity involved, e.g. person - John, reactor - r101. In a similar way a "variable" is treated as an attribute that names some quantity or mathematical object.

From the reading of Figure 6.5 it is seen that attributes can be sub-classified. This is more important for measured quantities as it allows discrimination between different formulations of the quantity involved, e.g. relative and absolute scales of measurement, empirical correlations, net, gross or compounded attribute values etc.

The use of the attribute "value" Concept provides a means of defining the adjectives "known" and "unknown". These terms are treated elsewhere as primitives.

Finally, the base level definition of a system as a collection of objects with inter-relationships between them can be used to construct an entry in a functional model that is applicable to virtually any type of process design problem.

6.3.1.1 Concept Layout

Having given a brief summary of the base level Concepts in Designer's Assistant it is now appropriate to describe in more detail the layout of information within a Concept. In a standard frame based system this would be covered in a straightforward way using a set of well defined syntax rules. The task is much more problematic here

however since Concept definitions cannot be sensibly separated from the non-deterministic primitives that interpret them. Nevertheless an attempt to summarise the main ideas is given below. More complicated features of Concepts will be introduced at later points in this chapter.

6.3.1.2 Concept Header

The header of a Concept is of the general form

concept(Name,SystemType,ConceptType,ConceptMode,Topic)

where

Name is the word or symbol used to represent the object, e.g. pump, reactor, equation, set, etc.

SystemType is one of "object,attribute, adjective or quantity" for non-functional entities.

ConceptType is the type of a Concept depending on its attained status. In the current version this is simply a choice of either formal or heuristic. Formal types have part analytic specifications.

ConceptMode is the mode of attribute combination and is one of "conj" (conjunctive), "disj" (disjunctive), or "relational". Simple disjunctive definitions tend to arise from imprecise or ambiguous analytic specifications and, as Bruner et al observed, are often replaced by terms defined in a more precise, conjunctive manner. A relational, formal Concept allows arbitrary relations or constraints to be made between constituent attributes or entities.

Topic refers to the name of the Topic structure of which a Concept is a part. In many cases this is give the same name as the module involved, e.g. thermodynamics. If the word "any" is used for Topic then the Concept is likely to be some sort of low level abstraction applicable across many Topics.

The information in the header is primarily used as a convenient means of implementation and for easy access of Concepts within the Prolog language. Most of the information is in fact redundant as it could be inferred from the body of the Concept description. The latter approach is not practical in the current system.

6.3.1.3 Concept Referent Bindings

Following a concept header is a term of the form

$ref ReferentVariable

or

$ref [ReferentVariable,RelatedReferent]

These terms are used to provide a binding with entities existing in a functional model. The functional model entries are referred to as "referents". This is a technical term used in theories of word meaning that requires some explanation.

In the discussion relating to Realism, Psychologism and the nature of intentions, see section 5.4.3, the work of de Saussure was referred to. de Saussure took a Concept to be the mental entity signified by some linguistic sign, i.e. word form. That is, some words, "signifiers", are mentally associated with a Concept, the "signified". A good example is the use of a pronoun, e.g. "he" or "it". This linguistic form refers to some mental entity that may or may not correspond to some object in the real world, e.g. a specific plant item or some mythical animal. The word that is used to refer to this mental entity is "referent". This usage of the word referent follows the linguistic theory of Hudson(1984). It should not be confused with other approaches which use referent to indicate the entity in the real world that is being referred to. The former usage is suited to theories comprehension that deal with models of the world, as is the case with mental model theory.

For the moment, it is sufficient to note that an "instance" of a Concept in a functional model is accessed by some unique token to indicate the name of the referent. Further Concepts are then interpreted with respect to these referents. The "$ref" term provides the basic mechanism to temporarily bind a Concept to a referent to allow the former to be interpreted in the context of the latter. In the case of attributes, adjectives, etc., the instance of the attribute Concept is itself indicated by a referent that is then related to some other referent, the RelatedReferent, that the attribute purports to describe. This is indicated by the pair of referents in the second version of the $ref term above.

It will be seen later that it is important to be able to reason about an attribute independently of any entity it is used to describe.

6.3.1.4 Context Dependencies

Concepts in Designer's Assistant must be interpreted within a given context. All interpretations are stored along with Context information, either in a direct or indirect form, e.g. via another referent. All Concepts, therefore, provide a term of the form

$context ($given ... context bindings ...).

A typical example is

$context ($given object - O).

This means that the Concept will be interpreted in the context of some other Concept, in this case the referent O of some other object. This context can then be applied to

any referent that can be treated as being equivalent to the object Concept.

The context binding mechanism is one of the basic ways in which the central idea embodied in Conceptual Categorisation of treating discriminably different things (e.g. referents) in an equivalent way can be implemented.

A special case worth noting is that of disjunctive contexts. When describing Concepts in the format presented here it was noticed that what was disjunctive about a Concept was not its description but rather its binding context. A simple example in Figure 6.5 above is the "variable" Concept which can refer to either some mathematical object O or some quantity Q. In such cases the ConceptMode is "disj(context)" and the referent binding term is written as

$ref [VarReferent,Choice - (O $or Q)].

In this example both O and Q are possible referents and Choice is the selected referent under a particular interpretation.

6.3.1.5 Concept Nature

The nature of non-functional Concepts is invariably abstract or physical and is indicated by a term such as

$nature [abstract].

The special case of Concepts of a mathematical nature was highlighted above. The words "event" and "situation" are also use to indicate special forms of Concept.

6.3.1.6 Functional Aspects of Concepts

In section 5.2.1 it was indicated that formal equivalence categories should be described in a way that stopped short of describing their use. This principle is generally adhered to but for ease of implementation it was found simpler to place a cross reference to the functional Concepts involved in the non-functional Concept descriptions. For example, the base level Concepts describing adjectives and values use the terms

$fnctnl [Adj $describes Object]

and

$fnctnl [Value $represents Attribute]

The implicit reading is that there are functional Concepts defined elsewhere that refer to "things to describe objects with", "things to represent attributes", etc. This general link mechanism extends to higher level physical or abstract Concepts, e.g. "things to transfer heat with", "things to solve algebraic equations".

6.3.1.7 Criterial and Non-Criterial Attributes

The criterial part of a Concept description is given by a "$def" term. Depending on the type of Concept this can take a number of forms. In the case of entity type Concepts it is likely to be a sequence, either conjunctive or disjunctive, of plausible assertions interpreted in a weakly analytic way. These assertions are always interpreted relative to some context. For example

$context ContextDescription

$def Assertion1 $and Assertion2 ...

In the case of adjective type Concepts the $def term is often a

$if ... $then ...

statement which either tests for the presence of other attributes or makes use of typical values for related quantities depending on the context. (The notation for referring to stereotypes is given later). As indicated earlier, the intensional use of "vague" adjectives is often preferrable for description purposes, especially in those case where one is ignorant of precise details.

The $def terms for physical quantities are the most straightforward and will normally be the defining algebraic relationships involved. The context of physical quantity concepts will normally refer to dependent relationships or quantities to help simplify the definition. For example,

```
$context ($given unit-U $and pressure-P $of U
                        $and temperature-T $of U
         )
```

Even at this stage it is worth emphasising that $def terms can be interpreted in a number of ways. For instance, there is no guarantee that a $if ... $then statement in a adjective concept will be interpreted in a left to right, test and act fashion as normally occurs in procedural type languages. A procedure may choose just to look at the $if parts or the $then parts in an attempt to make some sort of inference.

Non-criterial attributes for entity type Concepts are usually just listed by name with optional references to simple forms of typical, i.e. averaged, values. For example, non-criterial attributes for an adult male might include

```
$non_def_atts [ weight $typically "12 stone",
                age $typically between - ["20 years","70 years"]
              ]
```

If the typical information is at all complicated it is better placed in the attribute Concept itself. In the initial implementation it was intended that only the more

commonly used attributes be placed in this list in order to limit any search or effort in situations that require going beyond the use of criterial attributes. Infrequently applied attributes can be described in isolation and picked up in an exhaustive search of a Topic via the entities referred to in the $context term.

6.3.1.8 Concept Structure

The theory of mental models places a strong emphasis on the claim that the models constructed are structurally analogous to the state of affairs they represent. On first reading this seems a dubious claim, especially in cases of complex three dimensional shapes or complex sequences of events.Nevertheless the simplified structures used to emulate simple forms of syllogistic reasoning do appear to enable significant inferences to be made.

In the initial development of procedures to handle criterial and non-criterial attributes it emerged that the structural form of a Concept did appear to merit being treated in a separate way. The most notable feature of the interpretation of structural forms is the need to create referents in a functional model. This is particularly so in the case of aggregate forms of Concept, e.g. groupings of objects, a system, etc., because of the imprecise nature of some of the descriptions. For instance the term

it $consists _of collection $of objects - [ObjectlOthers]

is an allowable form that represents some indefinite grouping of objects. Interpreting this term calls for special entries to be made in a functional model to represent this situation. The approach needed is very similar to that described in section 5.4.2 in the introduction to syllogistic reasoning.

Until the significance of the structural forms of Concept descriptions become more clearly established it was decided to separate structural descriptions from other describing attributes. Many Concepts therefore make use of a "$structure term

$structure StructureDescription

There are a number of basic forms that can be used in structure descriptions :

```
X $consists_of Y
X $is_a member $of Y
X $is_a collection $of Y
X $is_a sequence $of Y
```

There are also a number of special terms useful for describing low level Concepts :

```
[undefined, regarded $as ... ]
[it $can_be defined]
[it $cant_be defined]
```

The definition of a set for example involves the first "undefined ..." form. Basic

notions of grouping are therefore treated as primitives at a perceptual level in Designer's Assistant.

If a Concept has a $structure term it will be referred to in the $def term by

see - structure.

The definition of a system in Figure 6.5 above uses this feature.

N.B. The use of the word structure here should not be confused with its use in section 5.3 concerning the internal structure of a category. Structure there refers to ordering of class members according to a "more prototypical than" relation.

6.3.1.9 Subtypes, Special Cases and Use of Inherited Forms

The final features of a Concept to consider at this point are those involving taxonomic aspects of descriptions.

To indicate that a link to a subclass, or a set of subclasses, exists a "$sub_types" term can be used. The definition of object, for example, uses the term

sub_types [physical, mathematical, abstract].

There are two points to note about this :

1. Indicating that such a link exists says nothing about whether or not attributes are inherited. What it does indicate is that there must be at least some sort of family resemblance between the subtypes involved. With nominal type words, e.g. pumps, an important common element will be the intended function. Inheritance of attributes in these cases will be less important since there are often relatively few to consider, e.g. the fluid involved.
2. Another useful term to help characterise Concepts is "$sp_cases". This is used to introduce a list of special cases for a given category. Special cases are not prototypical in any sense. Indeed, they are atypical and are listed precisely because they have special characteristics unique to a class. This term was introduced primarily for dealing with mathematical Concepts such as an empty set, identity forms, and so on.

Finally, attributes can be "inherited" from another Concept by using a "$atts_from" term. The general form used is

$atts_from [[Concept, HeaderInfo] - [LocalContext,InheritedContext]]

The Concept name and header information are used to locate the appropriate Concept. LocalContext is a list of variables used to bind to the local context while InheritedContext is the same but for the inherited context. These binding lists provide a simple but powerful way of imposing the epistemological , i.e. knowledge structuring, constraints suggested by Brachman, see section 5.1. This is because the variables in the inherited context will in fact be referents that can further constrained or specified locally within a Concept, using the $def term, to indicate how the

inherited information is to be used. Further, this can be done without the need for introducing an extra set of primitives at the epistemological level.

There is a subtle but very important point to note here. Inheritance is not viewed as a mechanism for providing common sets of slots or attributes to subclasses, as is the case in simple frame based systems. In Designer's Assistant the mechanism of interpreting a referent within an "inherited" context is precisely that of treating the referent in an equivalence based way compared to the "parent" referent. That is, attributes do not just "appear" at the lower level. The distinction between the two referents is maintained. The lower level referent is treated temporarily as being equivalent to the accompanying referent. Interpretations based on this equivalence need to be recorded along with this contextual information.

6.3.2 Functional/Procedural Concepts

The Concepts described so far have been mainly concerned with providing primitive tokens, e.g. '$consists_of", to describe and recognise entities in a functional model. That is the classes of tokens have been either one of description or recognition. The third class of primitive token is concerned with the ability to express intended forms of action or purpose. In formal Concepts these action or procedural type primitives should allow various types of algorithm to be expressed. In other cases the primitives will be used to express more general lines of reasoning such as the systems engineering methodology described in chapter 1. Both formal algorithms and "vague" forms of guidance are captured within functional Concepts in Designer's Assistant.

In the work of Bruner et al(1956) and Johnson-Laird(1983) the question of explicitly reasoning about procedural knowledge is not addressed. In those cases the "actions" involved are closely bound to physical responses and perceptual level Concepts. There is, however, a considerable body of knowledge relating to formal equivalence categories that is expressed in functional terms in process engineering. Common solution techniques for specific forms of unit models are typical examples, e.g. short cut flash calculations, as are more general solution techniques, e.g. Newton's method for solving a set of algebraic equations.
In Designer's Assistant solution techniques, methods, etc., are all viewed as functional Concepts. What distinguishes these types of Concept from those previously described is that some of the primitives used indicate specific actions or sequences of events that may be carried out or reasoned about.

There are two aspects of functional Concepts. The first involves procedural type descriptions as noted above. The second aspect, and the one that Bruner et al emphasise, is the grouping of entities that can perform a specific function. For

example, "things to solve equations with" refers to entities that are methods or algorithms. In the general case such functional groupings can only refer to stereotypes since many forms could be used to fulfil a particular purpose if need be, e.g. things to sit on. Nominal kind words, e.g. table forms, pumps, etc., will often be closely related to these functional descriptions.

Functional Concepts are useful for a number of tasks over and above the obvious need to express procedural information. These include :

1. Provide a means of guiding search activity when trying to fulfil some request.
2. Provide a means to perform analogies by treating other Concepts as equivalent to known ones for some given task. This is a fundamental form of creative reasoning.
3. Provide a means to label "intended purpose" in a functional model. This may be at a "physical" level, e.g. the use of an heat exchanger, or at an abstract level, e.g. the use of shortcut solution method.

6.3.2.1 Examples in Set Theory

To illustrate the form of functional Concepts using procedural type primitives some examples of Concepts in basic set theory are presented below. These Concepts form part of the maths module which is described later, see section 6.4.

The reason for choosing an abstract, formal topic, i.e. set theory, was to avoid introducing to many perceptual level primitives into the system, e.g. primitives for motion, spatial layout etc. As it turned out the extensive use of special notations in mathematics created many problems of its own. Concepts such as "set" or the "member of" relation are also so basic that primitives had to created to represent these ideas anyway. The development of Concepts in set theory therefore illustrates many of the problems that would be encountered in other areas.

The Concepts in abstract set theory are set up within a subtopic of the mathematics topic. They make use of some of the generic Concepts such as set expressions which are defined at this higher level. Fragments of these Concepts are listed in Figure 6.6 below.

Figure 6.6: Some Generic Concepts in the Mathematics Topic

```
concept(equation,object,formal,conj,mathematics) :-
 [
   $ref        E,
   $nature     [ object(mathematical) $but
                 it $represents relation - equals(Lhs,Rhs) ],
   $structure  [ it $consists_of relation - '='
                    $and arguments - [Lhs,Rhs] ],
   $def_atts   ( see - structure ),
   $sub_types  [equation(algebraic),equation(set)]
 ].
```

```
concept(equation(set),object,formal,conj,mathematics) :-
 [
   $ref        E,
   $nature     [mathematical],
   $structure  [ ( it $consists_of relation - '='
                      $and arguments - [Lhs,Rhs] )
                  $where
                 ( Lhs $and Rhs) $are expressions(set)
         ],
   $def_atts   ( see - structure )
 ].
```

Examples of the non-functional set theory Concepts are given in Figure 6.7 below. These range from simple definitions for a set to more complex descriptions of relations and functions.

Figure 6.7 : Some Functional Concepts in the Set Theory Topic

concept(set,object,formal,conj,theory(set)) :-

```
  [
    $ref S,
    $nature [mathematical],
    $fnctnl [ it $is_a tool(basic) $in theory(set) ],
    $structure [undefined,
                regarded $as_a collection $of objects
               ],
    $def_atts  ( see - structure ),
    $non_def_atts [cardinality]
  ].
```

concept(tuple,object,formal,conj,theory(set)) :-

```
  [
    $ref T,
    $nature [mathematical],
    $fnctnl [it $is_a tool(basic) $in theory(set) ],
    $structure [undefined,
                regarded $as_a sequence $of objects ],
    $def_atts  (see-structure),
    $non_def_atts [size],
    $sub_types [pair,triple,N-tuple]
  ].
```

concept(N-tuple,object,formal,conj,theory(set)) :-

```
  [
    $atts_from [tuple,object,formal,conj,theory(set)],
    $def_atts  ( $given tuple-T ;
                 T $has N-elements )
  ].
```

concept(pair,object,formal,conj,theory(set)) :-

```
  [
    $atts_from [tuple,object,formal,conj,theory(set)],
    $def_atts  ( $given tuple-T ;
                 T $has 2-elements )
  ].

/* ======================================================== */
/* ======================================================== */
```

concept(member,relational,formal,conj,theory(set)) :-

```
  [
```

```
    $nature   [mathematical],
    $context  ($given object-A $and set-B),
    $fnctnl   [it $relates object-A $and set-B],
    $def        (undefined $but test(internal))
  ].

concept(intersection,relational,formal,conj,theory(set)) :-
  [
    $nature     [mathematical],
    $context    ($given sets-[A,B]),
    $fnctnl     [f(A,B) ->> C $where (A $and B)
                              $are expressions(set)
                            $and C $is_a set
                ],
    $def        ( C $is set_of_all(X, X $is_a member $of A
                                    $and X $is_a member $of B)
                )
  ].

concept(difference,relational,formal,conj,theory(set)) :-
  [
     $nature     [mathematical],
     $context    ($given sets-[A,B]),
     $fnctnl     [f(A,B) ->> C $where (A $and B)
                               $are expressions(set)
                             $and C $is_a set
                 ],
     $def        ( C $is set_of_all(X, X $is_a member $of A $and
                                       X $is_not_a member $of B)
                 )
  ].


concept(N - relation(place),relational,formal,conj,theory(set)) :-
  [
    $nature   [mathematical],
    $context  ( $given sets - (a(1)...a(N)),

    $fnctnl   [ it $is_a relation(generic)-R
                 $and it $relates N - (sets -(a(1) ... a(N)))
                 $but treated $as_an object ],
    $def   (R $is_a subset $of
                    expression(set) - (a(1)'$X' ... '$X'a(N))
           ),
    $specific_def ( R $can_be defined
                              $by ( $a rule $or $a sentence ) ),
    $sub_types [relation(binary)]
  ].


concept(function,relational,formal,conj,theory(set)) :-
  [
    $nature   [mathematical],
    $context  ($given relation(binary)-F $between sets-[A,B]),
    $fnctnl   [ it $is_a relation(binary,generic)-F
                 $and it $relates sets-[A,B]
                 $but treated $as_an object  ],
    $def      ( $if for_all(X, X $is_a member $of A,
                            ( $exists_one pair(X,Y) $and
                              pair(X,Y) $is_a member $of F
                              $and Y $is_a member $of B
                            )
                            $or pair(X,Y) $is_not_a member $of F
                          )
                $then F $is_a function
              ),
    $specific_def (F $can_be defined $by
                     ($a rule-Rule $or $a sentence)),
```

```
    $non_def_atts [inverse,domain,range],
    $sub_types [mapping,function(partial)],
    $notation  [ F $is denoted $by [f,':',A,'->',B] ,
                 Rule $is denoted $by ([f,':',X,'->',Y] $or
                                       equation - (Y=f(X)) )
               ],
    $common_use [ $treat Rule $as F $and
                  (A $and B) $as one $of ['R','N','C','Z','Q'] ]
  ].
```

The descriptions of sets, tuples, n-tuples, pairs etc. are hopeful self explanatory.

The relational type Concepts are used to describe standard set theoretic relations, e.g. intersection, union, etc., as well as generic N-place relations, functions, etc. Relational Concepts make use of some additional terms that are worth describing.
The notation

f(A,...) ->> Z $where Qualifications

is simply a shorthand for function type relations, i.e. those relations which are normally used to evaluate some product entity. The intersection Concept is a simple example that uses this form. The more general way of indicating relationships is to use a term of the form

it $relates A $and B ... $where Qualifications

where A and B are referents picked up from the context bindings. The body of the specification for a formal relation is placed in the $def term as for attributes and adjectives.

In order to handle the mapping between surface level propositional representations using special symbolic notations, e.g. "f : R -> R" to indicate a real function, a "$notation" term can be used to denote the syntax. The entry for the function Concept is a good example of this. A formal description of a function states the domain and range involved as well as the rule used to define a specific function, e.g. "f : x -> sin(x)". The $notation entry for function indicates that either of the forms "f : x -> sin(x)" or "y = sin(x) { = f(x) }" can be used to denote the specific form of a rule.

There are two other useful features used in the description of the function Concept. The first is the form

$but treated as_a Whatever

in the $fnctnl term. This indicates that the relational Concept will be referred to in an object like way in surface level representations, e.g. in the subject position of a sentence.
The second feature, again related to surface level representations, is the use of a "$specific_def" term. The entry for a function is

F $can_be defined $by ($a rule-Rule $or $a sentence).

This term indicates that an entry for a specific function in a functional model can be picked up from either of these two forms.

The purpose of all these notational extras is to provide a convenient way of relating special surface level forms to entities, i.e. referents, in a functional model. This information can be acted upon by procedures for both the interpretation and generation of surface level descriptions.

Having described the object and relational Concepts involved it is now possible to describe the functional Concepts that can be used to manipulate them. Figure 6.8 below shows some of the Concepts needed to rearrange and solve simple equations involving sets.

Figure 6.8: Functional Concepts to Manipulate Set Expressions

```
concept( $to solve - equation , method(general),
         heuristic,functional,_) :-
 [
  $plan (
          $if equation $can_be classified  $then
               (
                $assign T $to type $of E ;
                $do solve - T
               )
         )
  ].

concept( $to solve - equation(set), evaluation(simple),
     heuristic,functional,abstract_set_theory) :-
  [
    $plan (
              $given equation(set) - E ;
              $if E $can_be solved $then
                   (
                     $do rearrange - E $into form(normal) ;
                     $check E $has arguments - [Lhs,Rhs]  ;
                     $do evaluate - Rhs ;
                     $assign Lhs $to Rhs
                   )
            )
  ].

concept( $to evaluate - expression(set),
      definition $of operators,
      formal,functional,abstract_set_theory) :-
  [
    $plan (
              $given expression(set) - E ;
              $if E $consists_of operator - Op
                    $and arguments - [L,R]
                 $then
                  (
                     $do evaluate - L ;
                     $do evaluate - R ;
                     $eval_term term(E,Op,L,R)
```

```
            )
         $else
            $if E $consists_of operator-Op $and argument-L
               $then (
                        $do evaluate - L ;
                        $eval_term term(E,Op,L)
                      )
      )
].
```

The sequence of intended events in a functional Concept is given by the $plan term. In order to represent intended actions three new primitives are introduced, namely $do, $assign, $eval_term. The first two are quite general and can be used to refer to non-mathematical tasks and objects. The $eval_term is a shorthand form that is only valid for evaluating mathematical type expressions (set, algebraic, etc.). The form

$do evaluate - Thing

is the more general form to use.

If these Concepts were actually used to solve equations or evaluate expressions the procedures would have to rearrange and unwind expressions down to the variable level "by hand" as it were, making use of the explicit definitions of set membership, intersection, union etc. In practice, this would be an extremely lengthy procedure. The following functional Concept provides an alternative definition that make use of an external, efficient implementation of a set manipulation tool.

Figure 6.9: An Alternative Functional Concept to Solve Set Expressions

```
concept($to solve - equation(set),evaluation(simple),
     heuristic,functional,abstract_set_theory) :-
 [
  $plan (
         $if equation(set) $can_be solved $then
         (
          $do rearrange - equation(set) $into form(normal) ;
          $do solve - equation(set) $using internal(set_eval)
         )
        )
 ].
```

Section 6.4.2.3 explains how encoded algorithms can be accessed and used to speed up certain tasks. The important point to note is that the interpretation procedures can be used to evaluate set expressions if need be and will interpret the explicit descriptions of Concepts in the set theory topic to do so. This sort of capability is the first step in what is commonly referred to as reasoning from first principles. The system can either make use of efficient deterministic implementations of algorithms that are coded up externally or it can use the original Concepts involved when situations arise that require their reinterpretation. An example of the latter is in learning or Concept attainment situations where previous interpretations have to be reevaluated to take account of new information. This sort of task requires that

procedural forms be represented in an explicit way that can then be used as the subject or data for other interpretation routines. Functional Concept descriptions provide this sort of functionality.

6.3.2.2 Using External Code to Solve Simple Equations

The discussion above indicated that it is important to be able to reason about and use formal functional Concepts both in an explicit declarative form as well as in a specific, efficiently implemented procedural form. In order to provide the necessary links between "external" code, functional Concepts and referents in a functional model the notion of an internal Concept is introduced.

Consider for example the alternative functional Concept for solving set equations above. This uses the line

$do solve-equation(set) $using internal(set_eval).

This says that the equation is to be solved using an internal Concept called set_eval. The definition of this internal Concept is given below.

```
internal( set_eval, functional, abstract_set_theory) :-
 [
  $context    (solving - (equation(set) - E) ),

  $guard [ E $consists_of relation-(=) $and arguments-[Lhs,Rhs],
           E $can_be solved $for Lhs ],
  $inputs     [Rhs],
  $outputs    [Lhs],
  $action     ( $call set_eval(Rhs,Lhs) )
 ].
```

The terms involved are quite simple :

- $context : provides context bindings with the functional model as usual.
- $guard : Either a conjunctive or disjunctive term that specifies constraints on the referents involved that must be satisfied before the rest of the Concept can be interpreted.
- $inputs : A list of variables, bound via the context, which are used as input arguments for the external code.
- $output : A list of variables that can be used to return values from the external code. These values have then to be associated with referents in the functional model once the code has been executed.
- $action : This term specifies the sequence of intended actions that are needed to map referent descriptions into variables, pass these variables to a call that invokes the external code, then map any output variables back to referents in the functional model once the code has completed.

Internal Concepts are just a special form of functional Concept that allows efficient implementations of algorithms to be used. There is the same flexibility in interpretation of internal Concepts as there is for other functional Concepts. For example, the specified sequence of of actions can be reasoned about in a declaritive way. The default interpretation is to evaluate the terms in a top down, left to right

manner giving the appearance of a procedural interpretation.

In order to demonstrate how the manipulation of Concepts based on formal equivalence categories can be easily represented, the functional Concepts required for solving a linear algebraic equation are now described.

The top level functional Concept to solve any type of single equation is

```
concept( $to solve - equation , method(general),
                  heuristic,functional,_) :-
 [
  $ref E,
  $plan (
         $if equation $can_be classified  $then
              (
               $assign T $to type $of E ;
               $do solve - T
              )
        )
  ].
```

If the referent E can be subclassified as algebraic then the "$do solve.." term will cause the following functional Concept to be interpreted :

```
concept( $to solve - equation(algebraic), evaluation,
                  heuristic,functional,algebra) :-
[
  $plan (
          $given equation(algebraic) - E ;
          $check solving - (equation - E $for variable-V ) ;
          $if E $is linear $in V
              $then $do solve - E $for V
                        $using internal(solve_lin)
        )
].
```

The plan involves testing whether the adjective "linear" can be applied to the variable referent involved, related in the functional model to the equation referent, and if so solves the equation for the required variable using the following internal Concept :

```
internal( solve_lin,functional, algebra) :-
  [
    $context   ( solving - (equation-E $for variable-X) ),
    $guard     [none],
    $inputs    [E,X],
    $outputs   [Val],
    $action    ( $call solve_lin(&E,&X,Val) ;
                 $assign value $of X $to Val
               )
  ].
```

In special cases where adjectives do have analytic specifications it seems acceptable to use some efficient implementation of code to test the truth value involved. The adjective linear when applied to an algebraic equation for a given variable is one such case. The Concept used for linear is as follows :

```
concept(linear,adjective,formal,conj,algebra) :-
  [
```

```
    $nature   [ abstract ],
    $fnctnl   [ it $describes $an equation(algebraic) ],
    $i_test     lin_test,
    $def      (see - $internal_test)
  ].
```

The definition makes use of an internal Concept lin_test which is defined as :

```
internal( lin_test, functional, algebra) :-
  [
    $context  (testing - (equation - E
                              $is linear $in variable-X) ),
    $guard    [ none ],
    $inputs   [E,X],
    $outputs  [Ans],
    $action   ( $call lin_test(&E,&X,Ans) ;
                $if Ans == true
                    $then $note ( E $is linear $in X )
                $else
                    $note ( E $is_not linear $in X )
              )
  ].
```

The "$action" term for this Concept finally calls some code, also called lin_test, to test the validity of the use of the adjective.

6.4 Initial Attempts to Implement a Maths Module

The discussion of Designer's Assistant has so far covered two areas :

1. The overall system structure, section 6.2.
2. The basic types of Concept and features available for representing Concept descriptions.

This section gives an overview of the initial attempt to implement a set of procedures and associated utilities to construct and interpret functional models using the available Concept descriptions. The development of these low level utilities represents a significant proportion of the research effort spent on the latter part of this thesis. Any individual utility is in itself, however, rather uninteresting (e.g. parsing and manipulating specific Prolog structures) and to describe them all in any detail would serve no particular purpose. All that is given here therefore is a summary of the main utilities that were developed for implementing a simple mathematics module based on the Concepts presented earlier. Appendix E gives a listing of the main utility files in the system and the types of operation involved.

6.4.1 Simple Example Problems

The initial development of utilities focussed on the interpretation of functional Concepts that could be used to fulfil particular requests made in the model of conversation. This in turn involved development of many of the utilities needed to create entries in a functional model and interpret the referents in a particular context.

To demonstrate the feasibility of the overall approach a simplified set of input/output routines were used to read in fixed forms of problem statement. Two extremely simple problems and answers are shown below taken from a script of a session.

```
problem area : abstract set theory.
consider :
   A = {1, 2, 3}
   B = {3, 4, 5}.
solve :
   C = A int B.
proceed.

done that ... ask a question for more detail
|: value of C?

C has value {3}


problem area : algebra.

solve :
   x*(2+a)-4 = y
   for x.
proceed.

done that ... ask a question for more detail
|:value of x ?

x has value (4+y)/(2+a)
|: how ?

I solved the algebraic equation using evaluation.
|: end.

yes
?-
```

The input syntax is quite restricted consisting of the following forms :

1. problem area : indicates the topic of interest.
2. consider : introduces a series of statements describing a problem.
3. solve : introduces a number of specific requests.

The constrained form of syntax greatly reduces the problem of providing a general means of mapping surface level representations into appropriate constructions in a functional model.

Despite the trivial nature of the problems considered the output does show that the Concepts in set theory and algebra can be interpreted as described earlier. The set example makes no use of internal Concepts and relies on the definition of the intersection relation given earlier. The following structures are parts of the internal terms used to describe the referents in the set equation. The first one is part of the description of variable C and the second one is part of the referent description for the entire equation.

```
recorded(functional_model,Term,_).
```

```
Term = object(variable(set), var3,
       [ [3] - ($by solving - equation(set) - var4) ],
       [ is_a([variable(set)]), status([solved])])

Term = object(equation(set), eqtn3,
       [ (var3 = [1,2,3] int [3,4,5]) -
                 ($by rearranging - equation(set) - eqtn3)),
         (var3 = var2 int var1) - given ],
       [other( [eqtn3 $can_be solved $for var3,
                eqtn3 $consists_of relation - (=)
                $and arguments - [var3, [1,2,3] int [3,4,5]]])])

where var3 is cross referenced to 'C'
      var2 is cross referenced to 'A'
      var1 is cross referenced to 'B'
```

The format of referent entries in a functional model is discussed below.

The linear equation is solved, or rearranged in this case, using a subset of the tools available in the symbolic equation solving system PRESS, see Bundy and Sterling(1982). PRESS itself is a large system that attempts to solve equations within a theorem proving environment and was used in the MECHO system referred to in Chapter 2. The specific tools needed to solve single algebraic equations were used in isolation and accessed via an internal Concept, solve_lin, as listed in the previous section.

6.4.2 Main Utility Areas

In order to interpret and manipulate Concepts and functional models a large set of utility procedures had to be developed (- several thousand lines of Prolog code). These utilities are still far from complete in that Concepts can only be interpreted in a limited number of ways. Despite these limitations the main areas of functionality considered are now given.

6.4.2.1 Expression Parser

In order to map the simplified problem descriptions into referents in a functional model a special parser was written that interpreted expressions and created entries in the appropriate way. The parser makes use of any special notational forms referred to in a Concept as well as common synonyms for the Concepts involved, e.g. intersection is written as "int", union as "vv", difference as "-". The same is true for for relational operators in algebra, (times, divides, etc.).

On reflection, it was realised that the parser should have been written as a set of generic procedures that handle special notations for any sort of Concept. These procedures should then be embedded within higher level functional Concepts for consistency.

6.4.2.2 Temporary Model Interpretations

Evaluating any term in a functional Concepts, e.g. "$do Thing", potentially involves interpreting a number of other functional Concepts in a recursive manner. Any call may involve treating one referent as being equivalent to another for some particular purpose, e.g. to "inherit" an attribute description, to make a simple form of analogy, etc. A set of procedures were developed to allow referents to be interpreted in different ways that catered for the recursive interpretation of Concepts. When functional Concepts "unwind" the referents representing temporary interpretations can be marked as "garbage" to be removed when convenient. Each recursive interpretation is summarised internally in a way that would allow the reasoning process and contextual information to be retrieved later if necessary. This is similar in spirit to the approach taken in truth maintenance systems, see section 4.3, with respect to storage of justifications. Due to the need to recall context, intended purpose, etc. the approach here has to record several extra items of information other than the nodes directly justifying some proposition, see section 6.4.3 below.

6.4.2.3 Using Internal Concepts

An important aspect of Designer's Assistant that could let it be developed into a practical tool is the ability to reason about and interface to external code that performs very specific tasks. This is done through the use of internal Concepts as explained earlier. A set of utilities were developed to map descriptions of referents in a functional model into variable bindings that could be passed to and from calls to external procedures. For examples, an external procedure may require an equation that is to be solved in a particular form. This is the case for the earlier example of a procedure to solve an algebraic equation. The following construct can be used to manipulate the referent description, i.e. the equation E :

```
$if $required form of E is weak(normal)
    $then  $do rearrange - E $using  ... Whatever
```

This can be used to rearrange E into an appropriate form.

The interpretation of actions before and after a call to an external procedure involves the same utilities described for functional Concepts. This potentially provides the ability to reason about the content and use of external code in as sophisticated a way as other functional Concepts.

6.4.2.4 Concept, Context, and Attribute Utilities

Most of the utilities all require access to the Concept and Attribute descriptions. A common set of procedures are therefore provided to access the various parts of Concepts. Other useful Concept procedures include those to test whether one referent can be treated as a subclass or equivalent to another. This is usually done within a

given context in which extra constraints are placed upon the "incoming" referent, i.e. the referent being matched with the Concept type referred to in the context. Matching referents up in this way requires matching relevant parts of their attribute descriptions. Procedures have been written to perform simple forms of attribute matching. This usually involves interpreting the $def term of the attribute involved which in turn can involve any of the utilities previously referred to.
The referent entries in a functional model have to record this contextual information in a suitable form. This is briefly described in section 6.4.3 below.

6.4.2.5 Topic Utilities

The development of Topics is far from complete in the current version of Designer's Assistant. A Topic is notionally meant to provide the necessary links to place Concepts within a taxonomic structure based on family resemblances.
At present there are only utilities for testing Topics in simple ways and no procedures yet exist for restructuring a Topic. Such a capability is required for properly implementing Concept attainment processes. The Topic itself is simply a set of Prolog structures that provide a number of alternative descriptions of how Concepts cross reference each other. This includes subcategory references, functional use of concepts, and references to Concepts, and references to Concepts in $structure descriptions. The functional groupings of Concepts are lists of Concepts that refer to other common Concepts in the $fnctnl term.

Stereotypical information is stored in the Concept descriptions themselves rather than within the Topic structure. The information simply consists of lists of named Concepts that will be used in preference to others by some of the interpreter utilities. There are three terms used to indicate different types of stereotypical information :

1. $protos List : A list of preferred prototypes for a given Concept.
2. $common_adjs List : A list of common adjectives used to describe a Concept.
3. $obs List : A list of common forms or observations used in surface level representations involving a given Concept.

Finally, the relational links between adjectives and quantities are currently implicitly specified by the $def entries used to describe these Concepts. This cross reference information should however be made explicit within the Topic structures.

The internal structure and use of Topics still requires considerable development before a set of guidelines can be given as to how best stereotypical information can be interpreted. The current structures merely provide the basic information that would be be expected in any cross referenced taxonomy.

6.4.3 Object Entries in the Functional Model

Entries for referents of Concepts in a functional model are implemented as pairs of Prolog terms recorded in Prolog's internal database. The functional model itself is not implemented as an explicit data structure. Rather it is a term used to describe the set of recorded referents that comprise it. This includes referents for attribute descriptions, surface level expressions, and internally generated referents resulting from context matching. To indicate the information used the terms used to record referents of objects are now briefly described.

There are two recorded terms for each referent, both recorded under the same database key. The key indicates the name of the functional model in use. Temporary keys can be generated when necessary to represent the idea of embedded functional models. The main items of information stored within a recorded term are :

1. ConceptWord : The word or token used to label the Concept, e.g. "pump".
2. ConceptType : Type information from the Concept header.
3. AttributeRefList : A list of attribute referents used to describe this referent.
4. ValueList : A list of "values" for this referent. For example, a referent for a mathematical variable may have one or more values. For other Concepts "value" often refers to the specific form of the $structure description.
5. DescList : A list of summarised descriptive statements in propositional form.
6. Cgtoi : If a generic referent is matched against an individual then the pair Iref-Behaviour is stored in the Cgtoi list, where Iref is the matched referent according the interpretation Behaviour. Behaviour is just the primitive token responsible for the top level match, e.g. $is_a, $has.
7. Citogrefs : This is a list of Ref-Behaviour pairs where Ref is an individual that is treated as equivalent to this referent under some Behaviour. There can obviously be many such equivalence based interpretations.
8. Cconrefs : When a generic context is matched against another context a number of constraints are generated. The referents for the things in the generic context that are constrained by the match are stored in this list. This list is therefore a subset of the variable listed in the original context bindings.
9. Cconstbinds : This list is updated when Cconrefs is updated. It is a list of pairs of the form IconstraintTerms - Ibinds. That is, the constraints on the referent under a context match along with the matching instance bindings. These are needed for unification purposes when updates are made due to changes in the matching instance referents.

This information is used and updated by many of the utilities that work with referents in the functional model.

6.4.4 Overview of Maths Module

Although the mathematics module was only developed to a limited extent it is worth outlining its overall implementation in the context of the wider operation of Designer's Assistant described in section 6.2.2.

The top level procedure that is invoked from the system level has a simple calling sequence :

```
maths_module(InputChannels ... OutputChannels) :-
      set_environment( ... ),
      set_up_data( ... ),
      plan_requests( ... ),
      run_requests( ... ),
      gather_report_terms( ... ).
```

The set_environment call picks up various pieces of information from the blackboard managers, accesses existing functional models if present, etc. The set_up_data is then used to perform the simplified mapping of surface level representations of data and requests into functional model entries. If any requests were made then an attempt is made to plan their evaluation using any appropriate functional Concepts. An attempt is then made to process these requests using whichever Concepts are necessary. The gather_report_terms is not properly implemented but it is supposed to tidy up extraneous referents in the functional model and generate appropriate responses to the original requests.

The general structure of this module could be used for many other Topic areas. The calling sequence is deceptively simple. The recursive revision mechanisms that are used in their implementation can result in extremely involved interpretations of Concepts. The level of complexity that results is not surprising given the reasoning tasks that are trying to be emulated.

6.4.5 Discussion of Designer's Assistant and Further Research

The development of Designer's Assistant is still at a very early stage. The descriptions of Concepts and Attributes in the current implementation provide the minimum sorts of information required to allow primitive procedures to be developed for their interpretation. The development of such primitives is by no means a trivial problem. This is especially true given that research into the nature and use of stereotypical information and how it can be expressed in general theories of word meaning is still in its early stages. There are very few firm guidelines therefore on what features reasoning with stereotypes should exhibit.

Despite this current lack of understanding of theories of comprehension a positive start has been made on the problem of implementing a reasoning system that captures the ideas put forward by Bruner et al on Conceptual Categorisation, Rosch and coworkers on the use of prototypes, and Johnson-Laird on the theory of mental models.

Two basic forms of Concept have been represented, namely functional and non-functional Concepts. The distinction between the two is fundamental as functional Concepts provide the basis and instrumental direction needed for implementing a variety of observed problem solving behaviours, e.g. solving by

analogy, incremental or gradual forms of learning, etc. The combination of functional Concepts and internal Concepts provides the possibility of being able to reason about complex solution techniques before and after their use, implemented in some efficient, external way. This, of course, is exactly how engineer's currently work with, say, libraries of numerical analysis routines. The important point to realise is that since the theory of mental models reduces the role of logic to a task variable as it were there is no reason why theorem provers cannot be treated in exactly the same way as, say, equation solvers. Just as a unit operation or plant can be modelled or mapped down into a set of algebraic equations and then handed over to an equation solver, so too could propositional or predicate representations of unit operations be generated and used by a theorem prover to derive some specific theorem.
Non-functional Concepts are usefully split into object like entities and the Attributes used to describe them. By representing Attributes as Concepts themselves it is relatively straightforward to create dynamic interpretations of object type referents by attaching or detaching Attribute referents in different contexts when necessary. It is then possible to formulate more general reasoning processes that discriminate between the use of criterial and non-criterial Attributes. It is also relatively clear how complex problems involving the interpretation of surface level representations, e.g. ambiguous statements, can be reasoned about using referents within a functional model framework.

There is a large amount of work that needs to be done in order to further the progress of developing reasoning mechanisms for Designer's Assistant. The most immediate problem is that of developing primitives that can map between surface level representations and functional model descriptions. Work was started on this problem but was not completed to any satisfactory extent. The importance of this aspect of the system cannot be overemphasised. A good deal of "everyday" reasoning relies heavily on making simple inferences from surface level type descriptions of problems, situations etc. The context of the model of the conversation provides instrumental direction for guiding any subsequent reasoning. The work is also essential for the fairly basic task of developing a working interface to Designer's Assistant.

Work of a more fundamental nature that needs to be done includes :

1. A reassessment of the fundamental word types that should be handled in Designer's Assistant and to what extent they rely on stereotypical interpretations and analytic specifications. For example, in formal topics such as set theory it would be useful to introduce what might be called an "abstract kind" that corresponds in many ways to natural kind words. Abstract kinds appear to make appeal to stereotype interpretations but have a strong analytic component as regards their abstract "structure".

2. Development of equivalence category primitives that allow Topic structures to be interpreted and revised in a comprehensive way. It is my opinion that these procedures cannot be sensibly implemented without detailed regard for the use of Concepts in surface level descriptions, e.g. written problem statements, simple two dimensional graphics images, etc.
3. A reassessment, and standardisation, of the various forms that can be currently used to formulate the different types of Concept descriptions.

Despite the significant research and programming effort that is still required it is my opinion that a knowledge based system designed on ideas taken from cognitive science research is indeed feasible. If this is the case then the benefits to be gained from further development along these lines could be substantial in terms of the reasoning capabilities of the system.

Chapter 7
Final Conclusions

The majority of the research described in this thesis was completed in the period 1984 - 1988. The final conclusions below therefore postdate the rest of this work by some two years. The comments made take into account relevant research that has been published in the intervening period.

The three main research goals of the work are laid out in chapter 1, section 1.9.

The first goal was to investigate and further develop representation techniques that would be suitable for use in a process engineering design environment. The investigation was to include :

1. The representation of objects, their attributes and inter-relationships with one another.
2. The representation of problem solving techniques that took into account the consistent use of data and knowledge in a design.
3. The explicit representation of an overall design methodology

The second and third goals were concerned with identifying what constraints a design methodology would place on a representation system.

In view of these goals an extensive study of both conventional integrated design environments using database technology and state of the art representation techniques from the field of Artificial Intelligence was made. It was concluded that attempting to implement the complex forms of representation required using conventional database systems was incompatible with the stated goal of maintaining the integrity of data and design knowledge throughout a design process, unless significant additions were made to the capabilities of these systems. Personal experience with the systems currently in use in I.C.I. has since confirmed this conclusion.

The study of A.I. representation techniques reflected state of the art developments in the field at that time. Three main areas were studied and prototype systems developed in Prolog were used to demonstrate the issues involved regarding their use for simple design problems :

1. Rule based production systems - a simple rule interpreter to select heat exchangers. The flexibility of combining Prolog within a production system was illustrated.
2. Blackboard systems - a framework for a modular design environment was described based on an overall blackboard structure. The system made use of inherited information within a taxonomy of classes and implemented a more sophisticated production rule system. Tokens used in rules were defined in object like structures that allowed the constraints that existed between them to be specified. Procedural attachment of code was similarly handled in a standard way. A mechanism was provided to allow "compromises" in the use of data and

application of rules. A simple physical property method selection selection system was implemented to illustrate the ideas.

3. Frame based systems - the use of object based representations using inheritance was investigated. A flexible representation environment, CLAP, was implemented on top of Prolog. The environment includes many of the advanced features found in current commercial systems. The environment combines different programming styles and forms of representation useful for many sorts of design problems : the use of objects, inheritance and message passing, procedural style method descriptions, and aspects of logic programming.

Considerable effort was made to show how much of this work, as well as standard heuristic and algorithmic approaches to process synthesis, can be all be understood and described using basic ideas from graph theory and first order predicate logic. An understanding of all these approaches and their formulation in a logic framework is important if sensible comparisons are to be made of the relative merits of the different programming styles involved.

The combination of all three studies mentioned went some way to addressing the three aims. The CLAP system best represents this part of the research. The idea embodied in CLAP was that a variety of programming styles and representation techniques will be required for a practical programming tool. The use of production systems can be easily embedded within CLAP as can standard procedural code. CLAP itself could be used as the implementation interpreter for a module in a blackboard system. Alternatively it could be used to reimplement the overall blackboard structure itself by making use of objects to represent blackboards and the message passing facilities provided.

There are two extensions in CLAP, not available in other systems reported in the literature, that are of particular relevance. First, CLAP provides a built in way of explicitly describing relationships between objects. CLAP differs from, say, Knowledge Craft in that it provides mechanisms not only to work with symbolic relationships, e.g. "is upstream of", but also to handle the generation of algebraic forms of constraints and their potential solution. The model building capabilities of CLAP have been subsequently put to use in the work of Hutton(1990) in the context of process unit modelling.

The second notable feature of CLAP is the use of what are called extended methods. Extended methods are intended to allow flexible design methodologies to be investigated by providing suitable high level constructs in which to describe general, non-committal forms of process design analysis, i.e. general guidelines on how to go about solving a design problem. A parallel construct is provided that would allow the investigation of an A.I.P. approach to process systems engineering design to be investigated. Such an investigation was not made in this thesis but the idea has been followed up by Hutton(1990).

During this period of research an appreciation had been gained of some of the more advanced areas of study in both A.I. and cognitive science. After careful consideration it was concluded that an overall modular structure for a design environment based on a blackboard architecture may well be appropriate for a process design environment but that the representation techniques needed were far from adequate. Chapter 4 provides an extensive discussion of the shortcomings of both rule and frame based representations. Alternative logic formulations were introduced to illustrate some of the problems involved. Unfortunately, many of these logics are still in an early stage of research and are very difficult to implement in a general and efficient way.

It was stated, section 4.1, that two fundamental objectives have yet to be achieved :

1. A unified and systematic representation of fundamental, heuristic and procedural knowledge and a consistent means for its interpretation must be devised.
2. A means of updating and revising representations of process models in a consistent way that tackles the problem of arbitrary tokenism must also be devised.

The remainder of the thesis was devoted to studying these problems. It fell into two main areas :

1. A study of the fundamental nature of Concepts and reasoning within a cognitive framework.
2. An evaluation of whether such ideas could be captured in a knowledge based system.

The study of Concepts and simple forms of reasoning about them was deemed necessary to shed light on the confusion that seems to have arisen from the intuitive representation and reasoning approaches used in A.I. The study concentrated on early work on Conceptual Categorisation in the field of psychology, Bruner, Goodnow and Austin(1956), as well the psychological study of Prototypes by Rosch and her coworkers(1970's). It also drew heavily on the recent contribution of Johnson-Laird(1983) to cognitive science in the form of mental models as a basis for a theory of comprehension.

The study raised some fundamental points concerning the nature of Concepts and "thought" that have many implications for representation systems. Perhaps the most important of these is that most Concepts in a theory of comprehension are inherently vague and are rarely understood in terms of necessary and sufficient conditions. The theory of mental models well exhibits this viewpoint and makes the fundamental assumption that the interpretation of Concepts will invariably have to be revised as reasoning proceeds.

Another extremely important point is the need to emphasise the role of equivalencing

in reasoning, i.e. treating discriminably different things in an equivalent way, in order to simplify and provide direction for reasoning tasks.

The final research effort was a preliminary attempt to investigate whether such ideas could ever be implemented in a system in order to address the two fundamental research objectives stated above. The tentative conclusion drawn from the very limited success of this work was that such a system could indeed be feasibly implemented. This optimism is based purely on the understanding of the problems involved gained during the attempted implementation of Designer's Assistant. The main, and perhaps somewhat obvious conclusion of this final part of the thesis is that considerable theoretical research effort is still required. The two most pressing areas are :

1. The nature of stereotypical information and its relation to different word types in the wider context of a theory of word meaning.
2. The instrumental role of models of conversation for communication purposes and their use in guiding instrumental cognitive activity.

This research is hardly within the traditional scope of chemical engineering. It is my opinion however that the main contribution that A.I. can make to process engineering in the future is not in the form of improved solutions to specific design problems. Rather, it will be in the form of systems that can handle design information in an "intelligent" manner and allow process engineers to communicate their design intentions in a way that has a direct effect on the reasoning and tasks performed by such systems. This is why such emphasis was placed on the role of context and intended purpose in the discussions in chapter 5.

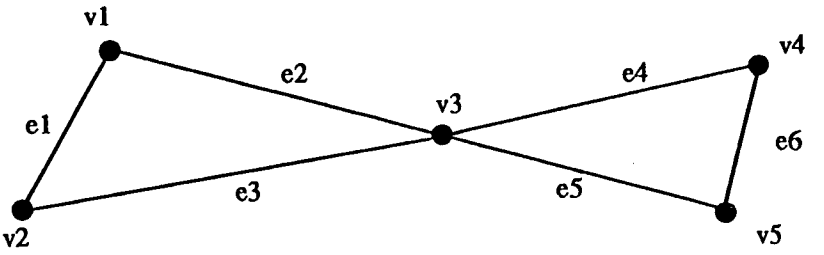
Perhaps the main research contribution of this work has been to bring together a number of related ideas from psychology and cognitive science and propose an overall framework for a reasoning system that could be potentially developed to meet these long term aims.

Appendix A
Basic Graph Theory Terminology

A graph is simply defined to be a set of points in space (also called "nodes" or "vertices") which are interconnected by a set of lines. In the case of an undirected graph the lines are referred to as edges and do not have an associated direction between the nodes which they interconnect. Thus, an undirected graph G is the pair of sets (V,E) where
a) V is the non-empty vertex set, and
b) E is the edge set of undirected pairs of not necessarily distinct elements of V.
For the graph in Figure A.1 :

$G = (V,E)$,
$V = \{v_1, v_2, \ldots v_5\}$,
$E = \{(v_1,v_2),(v_1,v_3),(v_2,v_3),(v_3,v_4),(v_3,v_5),(v_4,v_5)\}$

Figure A.1: A Simple Graph



The vertices, v_i and v_j, connected by an edge, e, are called the end points of e. An edge e is said to be incident with a node v if v is an end point of e. A node v_i is said to be adjacent to node v_j if (v_i,v_j) e E. Two edges are also said to be adjacent if they have a common end point.
The degree, or branching degree, of a vertex v, written d(v), is the number of edges incident with v.

Directed graphs are more common in A.I. applications than undirected graphs. A directed graph or digraph consists of a pair of sets (V,A), where
V is the non-empty vertex set, and
A is the set of directed pairs of, not necessarily distinct, elements of V.

The directed pairs (x,y) e A are called arcs or links. If an arc is directed from node n to n', n' is said to be a successor node of n(or child or offspring) and node n is termed a parent node of n'.
The number of successors emanating from a given node is called the out-degree $d^+(n)$. A directed edge (n,n') is said to incident *from* n and incident *to* n'. Similarly, the in-degree $d^-(n)$ is the number of directed edges incident to n. A digraph is balanced if for every vertex v e A, $d^+(v) = d^-(v)$, and a symmetric digraph is one in which for every edge (v_i,v_j) there is a complimentary edge $((v_j,v_i)$.

In the case of directed graphs there is a successor mapping
$S : V \rightarrow V$, where the image of n under S, S(n), is defined by
$S(n) - \{n' | (n,n') \in A\}$
Directed graphs are therefore sometimes alternatively represented as a pair (V,S).

An important subclass of graphs are trees. A tree is a graph in which each node, except the root node, has only one parent node. The root node in the tree has no parent node. A node which has no successors is called a leaf, tip or terminal node. The depth or level of a node n in a rooted tree is the number of links between the root node and node n. A uniform tree of depth N is one in which each node at a depth less than N has the same branching degree and all nodes at depth N are leaf nodes.
An ancestor of node n is any lying on the path from the root node to node n. If node n is an ancestor of node n' then n' is a descendant of n, or is accessible from node n. The path between nodes n_1 and n_k is the sequence of nodes $n_1,n_2,...,n_k$, where each n_i is a successor of n_{i-1}, and the path has length k. A circuit is a closed path (that is its endopoints coincide) and a loop is a circuit containing only one arc. In the case of undirected graphs the terms chain and cycle are used rather than path and circuit.

A graph G is said to be connected if there is a chain between every pair of vertices of G. The more precise definition of a tree is that it is a connected graph containing no circuits.

A.1 AND/OR Graphs

Many A.I. problems are concerned with AND/OR graphs rather than the special case of trees. In general, a node n can be an OR node with respect to one parent and and AND node with respect to another. Since multiple links can eminate from a node it is usual to call an AND/OR graph a hypergraph. Instead of arcs connecting pairs of nodes, hyperarcs connect a parent node with a set of successor nodes. The hyperarcs are called connectors. A k-connector is directed from a parent node to a set of successor nodes. If all the connectors are 1-connectors the AND/OR graph is the special case of an ordinary graph described previously.

Appendix B
Basic Terminology of Formal Logic

The following appendix provides a basic, somewhat informal, definition of the concepts involved in the simpler formal languages of logic. More detailed and rigorous definitions can be found in texts such as Marciszewski(1981), Pospesel(1976), Frost(1986), and Thayse(1988). The propositional calculus is outlined first and subsequently expanded to cover first order predicate calculus.

B.1 Propositional Calculus Terminology

Syntax : The syntax of propositional calculus defines the alphabet of symbols used in the language and how these symbols can be combined to form legal constructions. The propositional vocabulary consists of a set of propositions (or propositional constants) and of five logical connectives. A logical proposition, P say, can stand for a variety of statements such as "he101 is a pump", "the reactor is overheating", etc. The convention used here is to use upper case letters for arbitrary propositions, lower case for well defined propositions. The logical connectives are used to combine propositions into well-formed formulas (wffs) and are shown in Table 2.a.1 below.

Table 2.a.1: Logic Symbols

Name	Symbol	Arity\Alternative Symbols	
negation	$\neg$	monadic	~, not
conjunction	$\wedge$	dyadic	&, ., and
disjunction	$\vee$	dyadic	|, or
implication	$\supset$	dyadic	$\Rightarrow$,$\rightarrow$
equivalence	$\equiv$	dyadic	$\leftrightarrow$

The meanings of $\neg$, $\wedge$, and $\vee$ correspond to everyday usage of these terms. The definition of logical (or material) implication differs, however, from the normal use of "implies". Consider the formula $H \supset C$, where H is the hypothesis (also called the antecedent) and C is the conclusion (or consequent) of the implication. Implication $H \supset C$ is usually defined as $\neg H \vee C$, which says that it is true if the consequent is true (regardless of the antecedent) or if the hypothesis is false (regardless of the antecedent) or if the hypothesis is false (regardless of the consequent). Similarly, the equivalence connective is defined as $[H \supset C] \wedge [C \supset H]$.
It is easily shown that any n-adic connective can be represented using these basic connectives. A simple example is that of representing "if X then Y else Z", which is modelled by the formula $(X \supset Y) \wedge (\neg X \supset Z)$.

A literal is just a proposition or the negation of a proposition. Formulas built using the $\wedge$ connective are called conjunctions, each component formula termed a conjunct. Likewise, a disjunction is built up with the $\vee$ connective, with disjunct component formulas. Any conjunction or disjunction composed of wffs is also a wff. The negation of a wff is also a wff.

The semantics of any formal language, including propositional calculus, defines how meaning is assigned to well formed sentences. Propositional logic is concerned only with sentences which are true or false. More formally, the semantic domain {T,F} is introduced, sometimes represented by {1,0}. An interpretation function, or interpretation of a formula, is a function which assigns a truth value to each

proposition. The domain of the function is the set of propositions, and an interpretation of a formula which is true is called a model of the formula. The semantics of propositional logic are compositional, or denotational, in that the meaning of a formula must be a function of the meaning of its components. This means that given the truth values of the components of a wff containing logical connectives, the truth value of the overall formula is uniquely determined. The standard connectives above are, therefore, described as truth-functional in nature. The semantics of each connective are conveniently described by truth tables as shown below for the dyadic connectives, Table 2.a.2.

Table 2.a.2: Truth Table for Dyadic Connectives

X	Y	X∧Y	X∨Y	X⊃Y	X≡Y
T	T	T	T	T	T
T	F	F	T	F	F
F	T	F	T	T	F
F	F	F	F	T	T

It is a trivial matter to combine such tables to assign a truth value to an arbtrarily complex formula. The use of truth tables to determine properties of formulas such as satisfiability, validity, equivalence and logical consequence, is called semantic reasoning.

A formula or set of formulas is satisfiable if there is at least one truth assignment which satisfies it, ie if it has a model and can be interpreted with value T, e.g. $p \vee q$ is satisfied when p is true.

A formula or set of formulas is universally valid when it is always interpreted as true for any interpretation of the constituent propositions, e.g. $\neg\neg p \equiv p$ or $p \supset p$.

Two formulas are equivalent if they are assigned the same truth value by every truth assignment which is appropriate to both formulas. This, of course, is just the definition of the logical connective ≡.

Valid formulas are often called tautologies and the notation $\models A$ means that A is a tautology. If S is a set of formulas then $S \models A$ denotes that all interpretations which satisfy S also satisfy A. A is then said to be a logical consequence of S. A tautology, therefore, is a logical consequence of the empty set. If a formula can be satisfied, i.e. it is consistent, but is not valid then it is said to be contingent. If a formula cannot be satisfied under any interpretation it is inconsistent, or unsatisfiable e.g. $P \wedge \neg p$ is inconsistent.

Finally, given the above notation it is clear that

$$B \models A \equiv \models (B \supset A),$$

which says that A is a logical consequence of B if and only if the material implication, or conditional, $B \supset A$ is a tautology. This can be extended to

$$\{H_1,...,H_n\} \models C \equiv \models (H_1 \wedge ... \wedge H_n) \supset C,$$

where H_i are hypotheses and C is a conclusion. A fundamental problem in logic is to determine whether a formula C is a logical consequence of a set of formulas S. This is called the deduction problem. It reduces therefore to showing that the relevant conditional is a tautology. Now, a formula C is a logical consequence of S if and only if $S \cup \neg C$ is inconsistent, or alternatively a set is inconsistent if and only if F is a logical consequence of it. This is called the deduction principle and is expressed as

$$\{H_1,...,H_n\} \models C \equiv \{H_1, ... , H_n, \neg C\} \models F .$$

The use of the deduction principle is central to many logical theorem proving techniques.

Clauses and Normal Forms Many theorem proving techniques require that a wff be first transformed into an equivalent formula expressed in "normal" or "canonical" form. A basic form referred to is a "clause". A clause is simply a disjunction of a finite number of literals, eg the wff $(l_1 \vee l_2 \vee ... \vee l_n)$ is a clause. The empty clause is the only inconsistent clause and is usually represented by F.

There are two normal forms used in propositional logic, namely conjunctive normal form and disjunctive normal form. A formula is in conjunctive normal form if it is a conjunction of disjunction of literals, ie the conjunction of a finite number of clauses. It can be shown that any wff in propositional logic can be transformed into conjunctive normal form (eg see Frost(1986), p194). The dual concept of a clause is a cube. A cube is the conjunction of a finite number of literals. Likewise, disjunctive normal form is the disjunction of a finite number of cubes. Any wff can also be reduced to an equivalent disjunctive normal form.

An important subclass of clausal form is the Horn clause. A Horn clause is a clause containing at most one positive literal such as $\{\neg p, \neg q, \neg r, s\}$ which is equivalent to the conditional $(p \wedge q \wedge r) \supset s$.

A definite Horn clause contains a positive literal otherwise it is termed a negative Horn clause. A positive unit clause consists solely of a single positive literal. This can be used to represent an unconditional fact.

The use of Horn clauses are important in practice and are the basic representation form used in the programming language Prolog (Clocksin and Mellish,1984). The Prolog form of the conditional $(p \wedge q \wedge r) \supset s$ is

```
s :- p,q,r.
```

B.2 Predicate Calculus Terminology

Syntax: The basic components of predicate calculus are variables, individual constants, predicate constants, the connectives ($\neg$, $\wedge$, $\vee$, $\supset$, $\equiv$), the universal quantifier $\forall$ (for all), and the existential quantifier $\exists$ (there exists).

Variables are introduced to represent general statements about "classes" of unspecified individuals rather than fully specified individuals, e.g. in the following,

```
Every pump has a shut off-head.
P101 is a pump.
Therefore, P101 has a shut off-head,
```

general denotations like the word pump are called variables while particular denotations, like P101, are called individual constants.

Predicate constants, often just referred to as predicates, are used to make statements about individuals, e.g in 'P101 is faulty', 'is faulty' is a predicate constant. The expression is 'faulty' has no truth value itself. A truth value can only be assigned to a predicate constant if it is applied to an adequate number of arguments, called terms. The number of arguments is called the arity, and the application of the predicate constant is called the predicate form. For a predicate constant P of arity m, and terms

$t_1 \ldots t_m$, the predicate form is denoted by

$$P(t_1, \ldots, t_m), \text{ e.g. is_faulty(P101)}$$

A term is either a variable or an individual constant. It is worth noting that propositions are merely predicate constants of arity O.

Predicates of arity one can be used to define "sets" or "sorts", e.g. man(x), animal(x), vessel(y), reactor(Y). Some sorts may well include subsets of other sorts in their definition such as animal and human.

Quantifiers, Scope and Lambda Notation The principal use of the existential and universal qualifiers is to make useful statements which contain variables in their formulas, e.g. all tanks are types of vessels. The formula

$$(\forall\ x)\ P(x)$$

has value T as an interpretation when the value of P(x) under this, interpretation is true for all assignments of x to entities in the domain. The formula

$$(\exists\ x)\ P(x)$$

has value T for an interpretation when the value for P(x) under this interpretation is T for at least one assignment of x to an entity in the domain. The scope of a quantification refers to the formula to which quantification is being applied. An occurrence of a variable x in a quantification, $\forall x$ or $\exists x$, is said to be quantified and all other occurrences of x in the scope of the quantifiication are said to be bound. Variables which are neither bound or quantified are said to be free. The notion of bound variables as "place holders" is more clearly expressed by Lambda notation both in mathematical formulas and logic formulas. Consider the simple non-linear function.

$$f(x) = \sin(x) + 5$$

This is expressed in Lambda notation as

$$f = \lambda x.\sin(x) + 5$$

In this notation x does not denote any object but the "place" of an object which will be occupied by other objects when the function f object is used. The object status of f is also better emphasised in this notation. Similarly in logical expressions, if L is a logic formula and if x is a variable of L then λxL is called a λ-expression. The lambda operator is a useful extension to predicate calculus and results in the Lambda Calculus. This is briefly described below.

First Order Predicate Calculus

A basic extension to the predicate calculus described above is the replacement of the concept of individual constant by the more general concept of functional constant. Functional constants have well defined arities like predicate constants but differ from predicates in that they do not have the values T or F assigned to them. Instead, when a functional constant is applied to an adequate number of terms it returns in "object" in the entity domain related to its arguments. e.g. the functional form, father(JOHN), can be used to denote the mapping between the entities JOHN and his male parent.

Functions can be recusively applied as in father(father(JOHN)). The concept of a

term is now extended to cover either a variable or a functional form. Predicate forms can, therefore, use functional forms in their argument list,

e.g. married(father(JOHN), mother(JOHN)),
is_controlled(outlet_stream(R101)).

Individual constants are just functional constants of arity 0.

Given this definition of functional constants and the earlier definition of quantification, a logic is said to be first order if it permits quantification over individuals but not over predicate or functional constants. This implies that formulas such as $(\forall P)\, P(x)$ are not wffs in first order predicate calculus. Higher order languages permit such formulas.

B.3 Substitution and Unification

A substitution is a mapping from the a set V of variables with a set T of terms. A substitution s is represented by a set of ordered pairs $\{t_1/v_1, t_2/v_2, ..., t_n/v_n\}$. The pair t_i/v_i means that term t_i is substituted for variable v_i throughout a formula. The process of finding substitutions of terms for variables to make expressions identical is called unification. This is an extremely important technique in theorem proving and more generally in symbolic computation.

Consider the literal suitable_estimation_method(p,t,fluid,METHODA), where p,t, and fluid are variables, METHODA is a constant. The substitution instance {z/p, w/t} giving suitable_estimation_method(t,w,fluid,METHODA), is called an alphabetic variant of the original literal since the variables have merely been renamed. The substitution {10/p, 273/t, METHANE/fluid} results in a ground instance since none of the terms in the literal contains a variable. A term t_2 is said to be an instance of term t_1 if there exists in substitution s such that $t_2 = s(t_1)$. If a term contains no variables it is said to be fully instantiated. Each occurrence of a variable in an expression must be substituted by the same term throughout the expression such that distinct variable names remain distinct after any substitution. This is called the uniform substitution rule.

If a substitution s is applied to a set of expressions $\{E_i\}$ the substitution instances are denoted by $E_i s$. A set of expressions is said to be unifiable if there exists a substitution s such that $E_1 s = E_2 s = ... = E_n s$. s is said to be a unifier since the set $\{E_i\}$ collapses to a singleton. e.g. The substitution s = {CHLORINE/x, 298/t} unifies

{EXPENSIVE(x,COST(x,t),298),
EXPENSIVE(x,COST(x,298),298)}

to give {EXPENSIVE(CHLORINE,COST(CHLORINE,298),298)}.

Most unification algorithms are concerned with finding the most general unifier of a set of expressions. In the last example, for instance, there was no need to substitute CHLORINE for x. This unified literal can be expressed by

$$l_u = s_u(l_1) = s_u(l_2) = ... \; s_u(l_n), \text{ where } s_u = s_1 \cup s_2 \cup ... \cup s_n.$$

s_u is also referred to us the unifying composition. Table 2.a.3 below shows some examples of unifying compositions.

Table 2.a.3: Example Unifications

S_1	S_2	S_u
{A/y}	{y/x}	{A/y, A/z}
{f(z)/x}	{f(A)/x}	{f(A)/x, A/z}
{A/x}	{B/x}	inconsistent
{g(y)/x}	{f(y)/y}	inconsistent

The last entry is important as it illustrates that no variable x can be replaced by a term containing that same variable which does not reduce to x. In such cases unification is impossible. Such a verification, called the occurs-check, is often omitted when efficiency is important as is the case of the Prolog language.

Unification is a naturally recursive process of linear complexity in the size of the terms to be unified. Figure 2.a.1 below uniformally outlines the UNIFY algorithm (Paterson and Wegman(1976), Nilsson(1982)) which will find the most general unifier of a set of unifiable expressions or return failure. The algorithm assumes that a literal will be expressed in list-structured form, e.g. P(x,y,f(z))is written as (P x y (f z)) in list structured form.

Figure 2.a.1: A recursive UNIFY algorithm

```
UNIFY(expressions E1, E2)

STEPS

1)   If E1 or E2 is an atom then interchange so that
     E1 is an atom and DO block SUBSTITUTE-CHECK
     (an atom is a predicate or functional or
      individual constant, or a variable or a negation symbol).

   BLOCK SUBSTITUTE - CHECK

   (1)   if E1 and E2 are identical return EMPTY SET.

   (2)   else if E1 is a variable START

         2.1)  if E1 occurs in E2 return FAIL

         2.2)  else return {E2/E1}

         FINISH

   (3)   else if E2 is a variable return {E1/E2}.

   (4)   else return FAIL.

   END BLOCK SUBSTITUTE CHECK

2)       Assign F1 to first element of E1, T1 to rest of E1.

3)       Assign F2 to first element of E2, T2 to rest of E2.

4)       Assign S1 to result of UNIFY (F1, F2).

5)       If S1 = FAIL return FAIL.

6)       Assign G1 to result of applying S1 throughout T1.

7)       Assign G2 to result of applying S2 throughout T2.

8)       Assign S2 to result of UNIFY (G1, G2).
```

9) If S2 = FAIL, return FAIL.

10) Return the composition of S1 and S2.

B.4 Prenex, Normal, Skolem and Clausal Forms

In propositional logic conjunctive and disjunctive normal forms, as described earlier, were introduced to simplify testing for consistency or validity. These forms can be directly extended to predicate logic but are restated in terms of a more general form, namely prenex form. The definition of a prenex form is simply a matrix preceded by a prefix which is a finite sequence of quantifications of distinct variables, e.g. $Q_1x_1Q_2x_2...Q_nx_nM$ is in prenex form where Q_i denotes either $\forall$ or $\exists$ and M is a quantifier free formula, the matrix. It can be proved that every logical formula can be stated in an equivalent prenex form (see Thayse, 1988). The following algorithm steps outline the process.

1) Eliminate equivalence and implication connectives by use of rewriting rules.

2) Reduce the scope of all negation symbols to apply at most to one atomic formula by use of rewriting rules, eg $\neg\neg A \rightarrow A$, $\neg\forall xA \rightarrow \exists x\neg A$
$\neg\exists A \rightarrow \forall x\neg A$, $\neg(A \wedge B) \rightarrow (\neg A \vee \neg B)$

3) Bound variables are renamed if necessary such that free and bound variables do not share common names. This is required within the scope of all quantifiers. (referred to as standardising variables).

4) All quantifictions are moved to the front of the formula by use of rewrite rules, e.g.
$(\forall xA \wedge \forall xB) \rightarrow \forall x(A \wedge B)$

In predicate logic a clause is the disjunction of a finite number of literals; a cube is a finite conjunction of literals; a conjunctive normal form is a prenex form whose matrix is a finite conjunction of clauses; and a disjunctive normal form is a prenex form the matrix of which is a finite disjunction of cubes.

In order to further simplify handling quantifiers Skolem forms of formulas are normally used. Any formula A can be expressed in a form S_A, of simpler structure, such that A and S_A are either both consistent or inconsistent. This form is called the Skolem form and is derived as follows.

1) Ensure the formula is closed, i.e. does not contain free variables. If formula A has free variables $x_1 \ldots x_n$ then the formula $\exists x_1 \ldots \exists x_n$ A is closed and is consistent if and only if A is consistent.

2) Translate the formula into an equivalent prenex form.

3) Transform the matrix of this form into equivalent conjunctive normal form. This results in a closed prenex form.

Steps 1 to 3 are preliminary in nature. The skolem procedure itself now follows:

4) For each existentially quantified variable associate a functional form whose functor is a new, distinct functional constant and whose arguments are any universally

quantified variables which occur before the existential quantifier, e.g. for
$(\forall y)[(\exists x)\ P(x,y)]$
associate g(y), say, with x.
The function g(y) is called the Skolem function and maps each value of y into "the x that exists".

5) In the matrix of the formula replace all occurrences of existentially quantified variables by their associated functional forms.

6) Remove all existential quantification from the prefix. This results in the Skolem form.
e.g. $(\forall y)[(\exists x)\ P(x,y)]$ becomes $(\forall y)\ P(g(y),y)$

A skolem form, therefore, is a closed prenex form such that the prefix only contains universal quantifiers. It is usual to drop the prefix and implicitly assume universal quantification.

A clausal form is simply a Skolem form whose matrix is transformed into conjunctive normal form. Nilsson(1980) describes an equivalent process in some detail with examples. Clocksin and Mellish(1984) present a Prolog program to transform formulas to clausal form. See also Bundy(1983).

B.5 Formal Deduction Systems, Rules of Inference, Theorems and Proofs

The use of formal languages such as propositional logic and predicate calculus for theorem proving purposes often involves the use of a formal deduction, or more generally production, system. The purpose of such a system is, given a set of wffs S, to deduce logical consequences of S. In formal deduction systems the mechanisms by which this is achieved are purely syntactic in nature and can proceed, if necessary, without reference to truth values.

The type of systems most studied in logic are called axiomatic systems. An axiomatic system consists of a formal language, a set of logical axiom schema, and a set of inference rules. A logical axiom is a universally valid wff of the formal language e.g. $P \rightarrow [P \vee Q]$.
An inference rule defines the syntactic operation, or mechanism, by which new valid statements can be derived from existing ones, i.e. they allow deductions to be made. A well known inference rule is Modus Ponens which states that given $P \rightarrow Q$ and P then infer Q.
In predicate calculus the derived wffs are called theorems. A derivation or proof of a theorem is simply the ordered sequence of axioms and inference rules that were invoked to produce that theorem.

There are two important properties of a formal deduction system: soundness and completeness. A system is said to be sound if all of the formulas which can be derived from a set of wffs S are also logical consequences of S, i.e. every theorem is a valid formula. If a formula A can be derived it is written as $\vdash A$.
A sound system has, therefore, the property that

$$\vdash A \rightarrow \models A.$$

The converse is true for completeness,

$$\models A \rightarrow \vdash A.$$

That is every logical consequence of a set of wffs can be derived from that set. A sound set of inference rules is an essential property of useful formal systems but a complete set of rules is not always required, although usually desired.

Finally it is worth noting that there is nothing unique as regards the set of axioms and inference rules that are adopted. For example, Lukasiewicz's (1929) axiomatisation of prepositional logic consists of the axioms

$$[\ \neg P \rightarrow P\ ] \rightarrow P$$
$$P \rightarrow \neg P \rightarrow Q]$$
$$[P \rightarrow Q] \rightarrow [[Q \rightarrow R] \rightarrow [P \rightarrow R],]$$

together with the inference rules Modus Ponens and Uniform Substitution regarding any propositional variable in a derived theorem. For example from $P \rightarrow [\ \neg P \rightarrow Q\ ]$ infer $A \rightarrow [\ \neg A \rightarrow Q\ ]$.
Hilbert and Ackerman (1928), however, chose a less obscure but less concise set of abbreviations and axioms :

Let $P \rightarrow Q$ stands for $\neg P \vee Q$
$P \wedge Q$ stands for $\neg[\neg P \vee \neg Q]$
$P \leftrightarrow Q$ stand for $[P \rightarrow Q] \wedge [Q \rightarrow P]$,

with axioms,
1) $[P \vee P] \rightarrow P$
2) $P \rightarrow [P \vee Q]$
3) $[P \vee Q] \rightarrow [Q \vee P]$
4) $[P \rightarrow Q] \rightarrow [[R \vee P] \rightarrow [P \vee Q]]$

Proofs which use a set of axioms and a single inference rule, such as Modus ponens, are usually referred to as Hilbert-Like" Proofs. An alternative approach is not to use axioms at all but only inference rules resulting in "Gentzen-Like" proofs after Gentzen (1934). The latter type of system is called a natural deduction system. Inference rules are of the form

$$\frac{S_1 \ldots S_n}{S}$$

which indicates that S can be deduced from $S_1 \ldots S_n$.

A set of introduction and elimination rules are defined for connectives and quantifiers e.g. the introduction rule for A is

$$\frac{S \Rightarrow A,\ S \Rightarrow B}{S \Rightarrow A \wedge B}$$

The symbol $\Rightarrow$ is used in this context to indicate an inference or a derivation and as such is a formal object (or metasymbol) representing the reasoning itself.

Finally, a theory T consists of a formal system together with a set of wffs, collectively referred to as the "proper axioms" of T, which are known to be true in some set of intended interpretations. In particular it is those interpretations which satisfy all the proper axioms of the theory. A theory can, therefore, be used to represent a "given situation" or "state of affairs" from which further conclusions can be drawn.

B.6 Lambda Calculus

Lambda calculus is an extension of predicate logic used in modal or intensional logics. Modal logics are concerned with "possible" worlds, i.e. hypothesised situations, and require a mechanism for expressing properties of, as yet, unknown objects. At some later point in the reasoning process these unamed objects may be named or identified with a given referent. The lamba operator, λ, is used to express the required properties as follows.

Set notation is familiar in mathematics. To express, for instance, the set of process units in a plant one could write

{u|u is a process unit}

The set of conditions on any variable u can be extended as necessary,

e.g. { u| (u is a process unit) and (throughput of u > 10 te)...},

and multiple variables can be used,

e.g. {u,p: (p is a plant) and (u is a process unit in p)}.

The set expression merely specifies, or defines, the set.

Similarly, a lambda expression denotes some set of objects, possibly singleton. For example,

λx[design_team_task(alistair, project1, x)] ,

denotes the set of tasks that alistair has to perform in project1. The lambda operator is similar to the existential and universal quantifiers in that it serves to bind the occurrences of variables.

e.g. λx,y [design_team_task(x, project1, y)]

denotes the set of pairs (x,y) that result in the predicate design_team_task being satisfied.

If F is a logic formula then λxF is a Lambda expression denoting some set. An n-ary Lambda expression

$$\lambda x_1, \lambda x_2, \ldots, \lambda x_n F$$

consists of a body F together with a list of n variables called the formal parameters. The values of the lambda expression $\lambda x_1, \ldots, \lambda x_n F$ are the values of the n-ary predicate obtained by assigning instances to the formal parameters, e.g.

λx,y(x + y) applied to 4 results in λy(4 + y)

F can, in fact, be a functional specification, i.e. one returning an object rather than a predicate. This is how it is normally used in modal logics.

Consider, for example, a separation synthesis problem. The problem is to select a separation unit, as yet unknown, that must satisfy some required properties $p_1 \ldots p_n$, e.g. achieve 99% separation. This can be written as

$$\lambda x[p_1 \wedge p_2 \wedge \ldots p_n]$$

This functional specification can be created as an object in its own right and can be "applied" to other objects, i.e. arguments, like any functional constant in predicate logic. This is a subtle point for the novice. If the synthesis problem is represented, say, by the predicate

design_item(separation_unit ??var)

then ??var has to be some function which evaluates to an object in a possible world. A logical conjunction can't be used directly as no such object might exist. What is required is to be able to reason about a hypothetical object which stands for ??var and has a specification of its properties.

B.7 Validity checking using the Tableau Proof Method

The top-down tableau proof method is most easily described in the context of checking the validity of a formula by means of search of a counter example. A simple example from Frost(1986) is given below.

A tableau consists of two columns, a left column and a right column. Formulas in the left column have a truth assignment of T, while those in the right column have a truth assignment F. A tableau can be "split" when there are two, or more, ways in which a sub-formula can be assigned the required truth value.

Consider the formula $[[Q \vee \neg P] \rightarrow P] \rightarrow P$. To check if the formula is valid an assignment of false is made, i.e. place the formula in right column.

| [[Q ∨ ¬P] → P] → P

A formula of the form $A \rightarrow B$ is false iff A is T and B is F. So the tableau expands to

[Q ∨ ¬P] → P | P

The sub-formula in the left column can be assigned with value T in two ways; either $[Q \vee \neg P]$ is false or P is true. So the tableau is split, and shown in full, as follows:

| [[Q ∨ ¬P] → P]
[Q ∨ ¬P] → P | P
Q ∨ ¬P
tableau 10 *tableau 11*

using the L → rule.

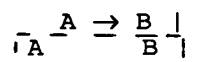
The sub-proof represented by Tableau 11 "closes" since P occurs in both columns indicating a contradiction. The proof continues with tableau 10 which requires that $(Q \vee \neg P)$ be false. This can only happen if both Q and $\neg P$ are false giving

| Q ∨ ¬P
| Q
| ¬P
P |

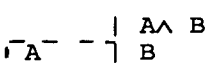
Finally, $\neg P$ can only be false if P is true, also shown above. Again this sub-proof closes as a contradiction in the truth assignment of P occurs.

The following eight simple rules are used to construct tableau proofs.

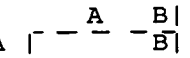
(R →) : If A → B occurs in the right column, put A in the left column and B in the right.
(L →) : If A → B occurs in the left column, split the tableau. Write A on the right of the new tableau and B on the left of the old one, i.e.



(R ∧) : if A ∧ B occurs on the right, split the tableau. Write A on the right of the new one and B on the right of the old one.



(L ∧) : if A ∧ B occurs on the left, put both A and B on the left.
(R ∨) : if A ∨ B occurs on the right, put both A and B on the right.
(L ∨) : if A ∨ B occurs on the left, split the tableau. Write A on the left of the new one and B on the left of the old one.



(R ¬) : If ¬A occurs on the right, write A on the left.
(L ¬) : If ¬A occurs on the left, write A on the right.

The binary labelling scheme can be conveniently used to reflect the structure of the proof.

Full details of the method can be found in Zeman (1973)

B.8 Identifying Paths in Matrix Connection

The matrix connection method requires an efficient way of eliminating paths containing a subpath Pa which has a connection, i.e. a complimentary pair of literals.

The following simple example from Frost (198??) outlines the approach. Consider the set of clauses

C1: {P,¬Q} C2: {¬P,R} C3: {¬R}
C4: {P,S} C5: {Q}

The steps are-

1. Choose any clause, C1 say, and any literal in that clause, P say.
   (i.e. a left to right depth first search choice will be used here but any strategy can be used).
2. Put all other literals in the chosen clause, C1, on a stack,
   i.e stack = {¬Q}
3. Identify a clause with a complimentary literal, ¬P, to the chosen one, eg C2 will do. All paths containing this connection can now be discarded. In this example, all subsequent paths through C3 C4 and C5.
4. Combine by selecting another literal in the last chosen clause, C2 here, and stack the rest. R is the only literal left to choose and there are none to stack.
5. Repeat the process form step 3 but avoid previously chosen clauses, In this example the pair R, ¬R connects the clauses C2 and C3.
6. When no literals are left in a clause, eg. C3, then backtrack to a literal on the stack, ¬Q, and repeat from step 3.

In this example ¬Q in C1 can connect with Q in C5. On backtracking this time there

are no more literals on the stack and the procedure terminates.

For a more detailed description of the alogrithm see Bibel (1982).

Appendix C
Uncertainty Factors and Probabilities in Production Systems

C.1 Confidence Factors in Mycin

Consider a set of diseases D1, D2... which are exhaustive and mutually exclusive. Baye's theorem results in the following formulae for the probability of a given disease given some evidence e:

$$P(D_i|e) = \frac{P(D_i)P(e|D_i)}{\sum_j (P(D_j)P(e|D_j)}$$

The left hand side is a useful piece of information. The right hand side involves the probabilities of diseases occurring and the probability that evidence e is observed in the presence of a given disease. Since evidence is usually combined in an incremental manner the equation is usually modified to the form

$$P(D_i|E2) = \frac{P(e|D_i\&E1)\ P(D_i|E1)}{\sum_j P(e|D_j\&E1)\ P(D_j|E1)}$$

where E1 is the set of all items of evidence so far and E2 is E1 plus a new item of evidence e.

This is not particularly useful in a production system since there will typically be a large number of diseases and a large number of items of evidence to consider. Computation of probability updates would be expensive. It is unlikely, however, that good prior probabilities for all items of evidence could be gathered from medical records to use in this formula.

Mycin trys to get round this problem by introducing a certainty factor to represent the belief in a hypothesis h given some evidence e. It is defined as Cf[h,e] = MB[h,e] - MD[h,e], where MB[h,e] is the measure of belief in hypothesis h and MD[h,e] is the measure of disbelief. They are both numbers which represent the degree to which evidence e increases the belief or disbelief in hypotheses h.

Their definition is borrowed from subjective probability theory. If P(h) represents someone's degree of belief in h, and P(h|e) is greater than P(h) then evidence e has increased their belief and decreased their disbelief in hypothesis h. The proportionate decrease in disbelief is given by

$$\frac{P(h|e) - P(h)}{1 - P(h)}$$

Conversely if P(h|e) is less than P(h) then e has lowered the belief in h, the proportionate decrease in belief being

$$\frac{P(h) - P(h|e)}{P(h)}$$

The first of these is taken to be MB[h,e] when p(h|e) > p(h). MD[h,e] is taken to be the latter when p(h|e) < p(h). The definitions used are written as

$$MB[h,e] = \begin{cases} 1 & \text{, if } P(h) = 1 \\ \dfrac{\max(P(h|e),\ P(h)) - P(h)}{\max(1,0) - P(h)} & \text{, otherwise} \end{cases}$$

$$MD[h,e] = 1 \quad \text{, if } P(h) = 0$$

$$\frac{\min(P(h|e), P(h)) - P(h)}{\min(1,0) - P(h)} \quad \text{, otherwise}$$

If one measure is non-zero the other must be zero for that hypothesis and evidence since an item of evidence cannot change the degree of belief in both directions at once. Also, if P(h|e) = P(h) then both measures are zero since the evidence has not changed the degree of belief. It is worth noting that

1. If h is certain then Cf[h,e] = 1
2. If not(h) is certain then Cf[h,e] = -1
3. If h is independent of e then Cf[h,e] = 0
4. In any case Cf[h,e] + Cf[not(h),e] = 0

The problem with these definitions is that there is no direct relation that can be used to calculate $Cf[h,e_1\&e_2]$ given $Cf[h,e_1]$ and $Cf[h,e_2]$. See Adams(1976) for a full discussion, also Buchanan and Shortcliffe(1984, Chapter 12).

C.2 Propagating Probabilities in Prospector

The odds of a hypothesis H with probability P(H) is given by

$$O(H) = P(H)/(1 - P(H)),$$
$$\text{i.e. } P(H) = O(H)/(1 + O(H))$$

The odds of H given evidence E are

$$O(H|E) = \frac{P(H|E)}{P(\text{not}(H)|E)} = \frac{P(E|H).P(H)}{P(E|\text{not}(H)).P(\text{not}(H))} = LS * O(H)$$

LS, the level of sufficiency, is estimated prior to calculation and used to label the inference links between nodes in an inference net. Similarly,

$$O(H|\text{not}(E)) = LN^* O(H)$$

LS and LN are just multiplicative factors, i.e. likelihood ratios, indicating the effect of evidence E, or lack of it, on the odds of hypothesis H.

Prospector has prior probability estimates of each assertion node, P(E). User replies to questions result in modified probabilities, (P(E|E'). A linear function is used to approximate the effect of evidence E' on hypotheses H.

$$P(H|E') = \begin{cases} P(H|\text{not}(E)) + \dfrac{P(H) - P(H|\text{not}(E))}{P(E)}.P(E|E') & \text{if } P(E|E') < P(E) \\ P(H) + \dfrac{P(H|E) - P(H).(P(E|E') - P(E))}{1 - P(E)} & \text{if } P(E) < P(E|E') \end{cases}$$

The first form is used when P(E|E') falls below the prior probability,negative evidence, and the second form is used when P(E|E') is estimated above the prior probability.

The problem is how to calculate the effect of several pieces of evidence E_i on a hypothesis H. If LS_i is the likelihood ratio for each piece of evidence E_i and it is assumed the E_i are independent under hypothesis H and not (H) such that

$$P(E_1,.....E_n|H) = \prod_{i=1}^{n} P(E_i|H) \text{ ,and}$$

$$P(E_1,....E_n|\text{not}(H)) = \prod_{i=1}^{n} P(E_i|\text{not}(H))$$

then the posterior odds are given by

$$O(H|E_1,.....E_n) = (\prod_{i=1}^{n} LS_i) * O(H)$$

If any E_i is false then LN is used instead of LS. The problem is that the E_i are only known to be true with probability $P(E_i|E')$ and the resulting calculation becomes somewhat complex. To avoid this a heuristic estimate is made:

$$O(H|E') = (\prod_{i=1}^{n} L_i') * O(H)$$

where $L_i' = O(H|E_i')/O(H)$, the effective likelihood ratio.

The valves for $O(H|E_i')$ can be obtained given the earlier formula for calculating P(H|E)

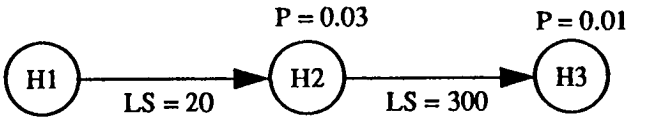
Ross(1987) gives the following example to illustrate the method. Consider the simple fragment of inference net in Figure?? Suppose the user has indicated that H1 is definitely true. Then the odds for H2 are

$$O(H2|H1) = 20*(0.03/(1-0.03)) = 0.618557$$
so $$P(H2|H1) = 0.618557/(1+0.618577) = 0.382166$$

The odds for H3 now depend on H2 and H1. The value for LS is modified from 300 to 300*((0.382166 - 0.03)/(1-0.03)) = 108.917 So, O(H3|H2) = 108.817*(0.01/(1-0.01)) = 1.10017 and P(H3|H2) = 1.10017/(1+1.10017) = 0.523849

These calculations assume that each E_i is conditionally independent of H or not(H). Probabilities at AND and OR nodes are taken to be the minimum and maximum values of the component nodes rather than attempt to perform the more rigorous calculations.

Figure C.1 : Fragment of Inference Net



Appendix D
Formalisation of Possible World Semantics

A non-empty set of possible worlds connected by a binary accessibility relation R is called a universe W. If a and b are two worlds in W and if world b is accessible from world a then aRb holds.

The pair (W,R) is called a structure. The formal properties of R, eg reflexivity, transitivity, determine the formulas that are deemed to be valid.

Let L be the language, ie form of modal logic, and P be the set of propositional constants of L. A valuation υ is a mapping WxP into {T,F}. This associates a truth value with each propositional constant in each world W. A "model" is a 3-tuple (W,R,υ). A model indicates in each world of W the set of propositional constants of L for which the valuation υ has assigned the value T.

The semantics of a modal logic are defined in terms of a consequence relation $\models$ between models and modal formulas.
The notation $(W,R,\upsilon) \models_w f$ is used to indicate that a formula $f \varepsilon L$ is true in world in w with respect to the model (W,R,υ).

The relation $\models_w$ is defined as:

1. $(W,R,\upsilon) \models_w T$
2. $(W,R,\upsilon) \models_w F$
3. $(W,R,\upsilon) \models_w f$ if $\upsilon(w,f) \models T$ and f is a propositional constant of L
4. $(W,R,\upsilon) \models_w f \supset g$ if $(W,R,\upsilon) \models_w f$ holds only when $(W,R,\upsilon) \models_w g$
5. $(W,R,\upsilon) \models_w Lf$ if for any world x eW such that wRx, one has $(W,R,\upsilon) \models_x f$.

The last rule is important in that it says a formula Lf is true in a world w of a model (W,R,υ) if the formula f is true in all the worlds of the universe W which are accessible from the world w.

The above description is necessarily brief. For a full introduction see Chellas (1980). It is hopefully clear, however, how possible world semantics relate to multiple contexts in a truth maintenance system, see Section 4.3.1.

Appendix E
Utility Files in Designer's Assistant

The main utility files developed for Designer's Assistant are briefly mentioned summarised here. All that is described is the main area of functionality of the clauses in each file. The utilities are all related to the notion of primitive tokens used to describe Concepts and their use in interpreting and manipulating functional models.

The system is implemented in Edinburgh Prolog. The utilities make heavy use of the pattern matching and backtracking facilities provided. The files are listed below in alphabetic order.

addfilts : utilities to add a referent to the list of "promoted" referents that are favoured for resolving surface level descriptions.

algebra.con (.fnc, .pro) : the basic concepts in algebra used to solve linear equations.

compatcode : utilities to decide if referents can be treated as equivalent based on Topic structure.

concept : the utilities needed to access the various parts of Concept descriptions.

constraints : an important set of clauses that attempt to match constraints between two contexts. This involves setting up setting up "intensional" type referents in the functional model with associated constraints on their interpretation.

context : The top level goals used to match contexts and their constituent referents against one another. Subtle use of Prolog backtracking is made to emulate the recursive revision of interpretations.

dautils : a wide variety of low level Prolog utilities, e.g. list scanning and manipulation functions, Prolog structure matching utilities.

filterdown, filterup : These clauses when combined together provide a sophisticated bidirectional search mechanism used to try and resolve the use of referents. Filter_down is used to take Concept descriptions, set up potential referents and then start a top down search to resolve those referents. Filter_up uses stereotypical inforamtion to propagate existing referents "up the search tree". Both sets of clauses make use of the notion of a promoted referent set to help reduce the search effort.

filters : A set of utilities used in the filtering search mechanism, e.g. promoting referents, removing them, resolving them, handling hanging referents, etc.

fmutils : the low level utilities for accessing referents in the functional model, e.g. adding referents, creating them, system level tests.

input : various input utilities including those to add assumptions to a referent description.

interp.pro : the top level set of clauses used to implement the interpretation of the various primitive tokens in Designer Assistant's Concept description language.

loosefilters : Utilities used in the filtering mechanism. These are used to delete referents that are thought to be inappropriate to an interpretation.

maths.con : some simple Concepts to describe equations.

newpatchoice : utilities to select parts of stereotypical descriptions to further resolve some intepretation.

parser : a parser used to handle special formats of surface level representations, e.g. equations that might be mixed in with sentences.

pending : The utilties used to get pending input of the I/O manager.

plan : the clauses used to set up and interpret functional plans in a procedural like way, i.e. left to right, top down order. This involves mapping referents into and out of nested interpretations. **process** : the top level clause for processing" a context when handling input.

runinput : the top level clauses used to get input and drive the subsequent interpretation.

rearrange : utilities used to scan or reset the pending input. This can involve adding partially processed input back on to the pending input so that it may be used to match up other stereotypes not yet considered.

runmodule : the top level clauses used to interface a module to the rest of Designer's Assistant.

sets : the various Concepts used to describe simple ideas in set theory.

structure.utl : various utilities to analyse the $structure parts of Concepts. These have an important effect on the referent filtering utilities.

wordtools : a set of clauses for handling "word" Concepts, that is the treatment of words in a language as Concepts in their own right. THese Concepts can be used to interpret input or generate output as required.

D.1 Example of a Word Concept

It was stated in chapter six that the interpretation of surface level representations had been started but had not been successfully completed. This is not quite true as many of the utilities needed to handle surface level input in the form of natural language sentences have been worked on. In particular the referent filtering mechanisms were developed, for the most part, to handle interpreting Concepts in the context of representing surface level forms in the functional model. Following the theoretical work of Hudson(1984) a novel form of natural language interpretation was worked on that avoided a strict syntactic analysis as is usually found in natural language front ends. Instead it is recognised that words in English are Concepts themselves that can be interpreted in much the same way as any other Concept. Hudson realised the importance of relating these Word Concepts to the other mental Concepts that they denote in the context of a conversation. Chapter 6 makes the point that this connection between Words and Concepts must be made in order to take part in conversations in an "intelligent" way, e.g. resolving ambiguity in a given context.

The filtering utilities in Designer's Assistant are, therefore, an attempt to provide a **context dependent** interpretation mechanism that can handle both the "syntactic" and "semantic" aspects of Word interpretation. This differs markedly from many natural language front ends which attempt to perform a rigorous syntax analysis followed by

a "semantic" interpretation. It takes little reflection to realise that this cannot be a plausible approach in a cognitive framework given the very limited grasp of grammar that many competent speakers of English possess (present author included !).

As a matter of interest, some Word Concepts are presented below that indicate the sorts of representations that were worked on. The context in these Concepts refers to referents in the model of discourse that the utilities make use of.

```
concept(sentence,object,heuristic,conj,any) :-
  [
    $ref S,
    $context  ($given conversation-C),
    $nature   [linguistic],
    $structure ( it $is_a
                 ( sequence - (A1...AN) $of words) $in C ),
    $def_atts (see-structure),
    $sub_types [sentence(declaritive)]
  ].

concept(sentence(declaritive),object,heuristic,conj,any) :-
[
  $ref SD,
  $vars [X,V],

  $nature [linguistic],
  $fnctnl ( SD $introduces
            ( ($a description $of object-X) $or
              ( $the type $of X) )
          ),
  $def_atts (see-functional $and see-structure),
  $structure ( [reference $to X, is, description $of X]
               $or
               [reference $to X,is,(a $or the),
                optional(description $of X),type $of X]
             ),
].


concept(reference,object,heuristic,conj,any) :-
[
   $ref    R,
   $vars   [W,Ws,X,C,T,A,S],
   $nature   [linguistic],
   $context ($given conversation-C $at time-T
                    $and addressee-A $of C
                    $and speaker-S $of C ),
   $fnctnl ( R $represents $an object-X ),
   $structure ( R $is_a ( word-W $or sequence-Ws $of words) ),
   $def_atts ( see-structure),
   $patterns ( ( [a,optional(description $of X),X]
                 $where X $is known $to S $at T
                 $but unknown $to A $at T
               ) $or
               ( [the,optional(description $of X),X]
                 $where X $is known $to (S $and A) $at T
               ) $or
               ( [name $of X] $where X $is known $to S $at T )
             )
].


concept(a,word,heuristic,conj,any) :-
  [
    $ref        A,
```

```
  $vars        [C,T,X,V],
  $context     ( $given conversation-C $at time-T ),
  $nature      [linguistic],
  $def_atts    ( see - functional),
  $fnctnl      ( it $introduces $an object-X
                 $where X $is unknown $to addressee $of C
                          $but known $to speaker $of C
               ),
  $patterns    [A,optional(description $of X),X,$rest],
  $mistakes    ( [A,verb-V,$rest] $then $assume A $is_a variable )
].
```

REFERENCES

Adams, J.B., "A probability model of medical reasoning and the MYCIN model", Mathematical Biosciences, 32, pp 177-186, 1976.

Adler, S.B., Spencer, C.F., Ozkardersh,H., and Kuo,C.M., "Industrial Uses of Equations of State : A state of the are review", in Phase Equilibria and Fluid Properties in the Chemical Industry, Ed. Storvick,T.S., and Sandler,S.I., ACM Symposium Series, 60, Washington, 1977.

Allen J.F., "Maintaining Knowledge about Temporal Intervals", Communications of the ACM, 26(11), pp 832-843, 1983.

Andrews, P.B., "Theorem Proving via general matings", JACM 28(2), pp 193-214, 1981.

ANSI, "Interim Report of the ANSI/X3/SPARC Study Group on Data Base Management Systems", ACM SIGFIDET, 7(2), pp3-139, 1975.

ART, Programming Primer, Inference Corp., 1985.

Baldwin,J.F.," FRIL - an inference language based on Fuzzy Logic", in Proceedings of Expert Systems, 1983.

Balza, R.M., Bernhardt, D.L., and Dube, R.P., "Data base technology Applied to Engineering Data", Proceedings of the Second International Conference on Foundations of Computer Aided Process Design, Snowmass, Colorado, CACHE Publications,1983.

Banares-Alcantara, R., "DECADE: A Hybrid Kowledge Based System for Catalyst Selection", PhD Thesis, Chem. Emg. Dept., Carnegie-Mellon, 1986.

Banares-Alcantara R., Westerberg,A.W., and Rychener,M.D., "Development of an Expert System for Physical Property Predictions", Comp. and Chem. Eng.,9(2)pp127-142, 1985.

Barr,A., and Feigenbaum, E.A., Eds.,"Handbook of Artificial Intelligence", Volume 1, Los Altos, California, William Kaufman, 1981.

Bellman, R., and Dreyfus, S.," Applied Dynamic Programming", Princeton, NJ, Princeton University Press, 1962.

Benayoune, M., and Preece, P.E., "An Engineering Data Model for Computer Aided design Databases", PSE 85, Institution of Chemical Engineers, Sysmposium Series No 92, 1985.

Benayoune, M., and Preece, P.E., " Review of Information Management in Computer Aided Engineering", Computers and Chem. Eng., 11(1),pp1-6, 1987.

Berlin,B.," Ethnobiological Classification in Cognition and Categorisation", Hillsdale NJ, Lawrence Erlbaum Associates, 1978.

Bibel, W.," A syntactic connection between proof procedures and refutation procedures", 2nd Conf. on Automated Deduction, Oberwolfach, Germany, 1976.

Bibel, W.," A comparative study of several proof procedures", Artificial Intelligence

18, pp 269-293, 1982.

Bibel, W.," Matings in Matrices", CACM 26(11), pp844-852, 1983.

Bingzhen,C., Jingzhu,S., Qing,S., Shanying,H.," Development of an Expert System for Synthesis of Heat Exchanger Networks", Proc. of PSE 88, Inst. of Eng. Australia, Aug., 1988.

Blass,E., "Methodical Dvelopment of Chemical Engineering Processes", Ger. Chem. Eng., 9, pp127-135, 1986.

Bobrow,D.G., and Stefik,M, "The LOOPS Manual",Xerox Corperation, 1983.

Bobrow, D.G., and Winograd, T., "An Overview of KRL, a Knowledge Representation Language", Cognitive Science, 1, pp 3-46, 1977.

Boffey, T.B.,"Graph Theory in Operations Research", MacMillan Press Ltd., 1982.

Boyer, R.S., and Moore, J.S., "The sharing of structure in theorem proving programs", in Melzer, B., and Michie, D., Machine Intelligence 7, New York, Edinburgh University Press, 1972.

Brachman, R.J.," A Structural Paradigm for Representing Knowledge", BBN Report, No. 3605, Bolt Beranek & Newman, Cambridge, Mass., 1978.

Brachman, R.J., "On the Epistemological Status of Semantic Networks", Associative Networks: Representation and Use of Knowledge by Computers, pp 3-50, edited by N.V. Findler, Academic Press, New York, 1979.

Brachman, R.J., Fikes, R.E. and Levesque, H.J., "KRYPTON: A Functional Approach to Knowledge Representation", FLAIR Technical Report No. 16, Fairchild Laboratory for Artificial Intelligence Research, Palo Alto, CA, May 1983.

Brachman,R.J.," I Lied about the trees, Or Defaults and Definitions in Knowledge Representation", The AI Magazine, pp80-93, Autumn, 1985.

Bransford,J.D., Barclay,J.R., Franks,J.J, "Sentence Memory: a constructive versus interpretive approach", Cognitive Psychology, 3, pp193-209, 1972.

Brownston, L, Farrell,R., Kant,E., and Martin, N.,"Programming Expert Systems in OPS5", Addison-Wesley, 1985.

Bruner, J.S., Goodnow,J.J., Austin,G.A.," A Study of Thinking", John Wiley and Sons Inc., New York, 1956.

Buchanan,B.G., and Shortliffe,E.H., Eds., "Rule based Expert Systems", Reading, Mass. Addison-Wesley, 1984.

Buchmann, A.P., "A methodology for the logical design of project engineering databases", PhD Thesis, Chemical Engineering Department, University of Austin, Texas, 1980.

Bundy, A.," The computer modelling of mathematical reasoning", Academic Press, London, 1983.

Bundy,A., and Sterling,L.S., "Meta-Level Inference in Algebra", Research Paper 164, Dept. of Artificial Intelligence, University of Edinburgh, 1981.

Bundy,A., Byrd,L.,Luger,G.,Mellish,C.,Milne,R., and Palmer,M., "Solving Mechanics Problems Using Meta Level Inference", DAI Research Paper 112, University of Edinburgh, 1979.

Bundy,A., Byrd,L.,Luger,G.,Mellish,C.,Milne,R., and Palmer,M., "A program to solve Mechanics problems", Working Paper 50, Dept. of Artificial Intelligence, Edinburgh, 1979.

Carbonell,J.R.," AI in CAI: An Artificial Intelligence approach to computer aided instruction", IEEE Transactions on Man-Machine Systems MMS-11, 4, p190-202, 1970.

Chang, C.L., Lee, R.C.T., "Symbolic Logic and Mechanical Theorem Proving", New York, Academic Press, 1973.

Charniak,E., "Ms. Malaprop, a Language comprehension program", Proc. 5th Int. Joint Conf on AI., MIT, vol 1, pp1-8, 1977.

Charniak, E., McDermott, D., "Introductio to Artificial Intelligence", Addison Wesley Pub., Reading, Mass., 1985.

Chellas,B.F.," Modal Logic: an Introduction", Cambridge University Press, 1980.

Chen, P.P., "The Entity Relationship Model- towards a unified view of data", ACM TODS, 1, pp9-36, 1976.

Cherry, D.H., "Data Organisation in chemical plant design", PhD Thesis, Cambridge University, 1975.

Cherry, D.H., Grogan, J.C., Knapp,G.L., and Perris, F.A., "Use of databases in engineering design", Chem. Eng. Progress, 78(5), pp 59-67, 1982. Clancey, W.J., The epistemology of a rule based expert system. A framework for explanation, Artificial Intelligence, 20, pp215-251, 1983.

Clancey,W.J. and Letsinger, R.,"Neomycin: Reconfiguring a rule based expert system for application to teaching", Proc. of the 7th Int. Joint Conference on A.I., p829-836, 1981.

Clancey,W.J.,(1983), "The epistemology of a rule based expert system: a framework for explanation", Artificial Intelligence, 20, pp 215-251, 1984.

Clocksin,W.F., and Mellish, C.S., "Programming in Prolog", Springer Verlag, Berlin, 1981, 1984 (2nd Edtn).

CODASYL, CODASYL Data Base Task Group Report,ACM, New York, April, 1971.

CODASYL Data Description Language Committee, DDL Journal of Development, Materiel Data Management Center, Quebec,1978.

Codd, E.F., "A Relation Model of Data for Large Shared Data Banks", Comm. ACM, 13, pp377-387, 1970.

Codd, E.F.,"Extending the database relational model to capture more meaning", ACM TODS, 4, pp397-434, 1979.

Cohen, P.R., "Heuristic Reasoning about Uncertainty: An AI approach", Pitman

Publishing, 1985.

Cohn,A.G.,"A more expressive formulation of many sorted logic", J. Autom. Reasoning, 3, pp113-200, 1987.

Craik,K.,"The Nature of Explanation", Cambridge University Press, 1943.

Craft, J., "The Impact of CAD and Database Techniques in Process Engineering", PSE 85, Institution of Chemical Engineers, Sysmposium Series No 92, 1985.

Date, C.J.," An Introduction to Database Systems", 3rd Edition, Addison Wesley,, 1981.

Dechter, R., and Pearl, J.," Generalized best-first strategies and the optimality of A*", Technical Report UCLA-ENG-83-19, p95-99, Cognitive Systems Laboratory, UCLA.

Douglas, J.M.," Conceptual Design of Chemical Processes", McGraw Hill Book Company, 1988.

Doyle,J.," A truth maintenance system", Artificial Intelligene, 12, pp231- 272, 1979.

Duda, R.O., Gaschnig, J.G.,Hart,P.E, "Model Design in the Prospector consultant for mineral exploration", In D.Michie, ed., Expert Systems in the Micro-Electronic age, Edinburgh University Press, pp153-167, 1979.

Duxbury, H.A. and Preston, M., "The Process Systems Contribution to Process Safety", Proceedings of Foundations of Computer Aided Process Operations, Park City, Utah, (CACHE Publication), Elsevier Science Pub. Inc., 1987.

Earl,W.B., and Williamson,C.J., "Control System Synthesis", Proc. of PSE88, Inst. of Eng., Sydney, Australia, 1988.

Erman,L.D., London,P.E., Fickes, S.F., "The design and an example of the use of HEARSAY-III", Proc. of AAAI-83, 409-415, 1983.

Etherington,D.W., "Formalising Non-monotonic reasoning systems", Technical Report 83-1,Dept. of Comp. Sci., University of Bath Columbia, Vancouver, Canada, 1983.

Etherington,D.W., "Formalising Non-monotonic systems", Artificial Intelligene, 31(1), pp 41-85, 1987.

Etherington, D.W., and Reiter, R., "On Inheritance Hierarchies With Exceptions", Proc. AAAI-83, Washington, D.C., pp 104-108, 1983.

Fahlman, S.E., "NETL: A system for Representing and Using Real World Knowledge", the MIT Press Cambridge, MA. (1979).

Fikes,R.E., and Nilsson, N.J, "STRIPS: a new approach to the application of theorem proving to problem solving", Artificial Intelligencem 2(3/4), pp 189-208, 1971.

Fikes,R., and Kehler,T., "Control of Reasoning in Frame Based Representations", Communications of the ACM, Special issue on Architectures for Knowledge Based Systems, 1985.

Fillmore, C., "The case for case", in Universals in Linguistic Theory, ed,. Bach,E. and Harms,R., Holt, New York, 1968.

Forgy,C.L.,"OPS5 User's Manual", Technical Report CMU-CS-81-135, Dept. of Computer Science, Carnegie Mellon University.

Forgy, C. and McDermott, J., "OPS: A domain independent production system language", in IJCAI 5, pp 933-939.

Frost, R.A.," Introduction to Knowledge Based Systems", Collins, London, 1986.

Fusillo, R.H., and Powers, G.J., "Synthesis of Operating Procedures for Complete Chemical Plants", Comp. and Chem Eng., 12(9,10), 1988.

Gardarin, G., and Gelenbe, E., Eds.," New Applications of Data Bases", Academic Press, London, 1984.

Garey, M.R., and Johnson, D.S.," Computers and Intractibility", San Francisco, W.H.Freeman, 1979.

Garvey,T.D, Lowrance,J.D., and Fischler,M.A., "An inference techniques for integrating knowledge from disparate sources", in Proc. IJCAI-81, Vancouver BC, pp 319-325, 1981.

Gibbons, A.," Algorithmic Graph Theory", Cambridge University Press, 1985.

Goldberg,A., and Robson,D.,"Smalltalk-80: The language and its implementation", Reading, Mass., Addison-Wesley, 1983.

Gray, P., "Logic, Algebra and Databases", Ellis Horwood Limited, Chichester, England, 1984.

Grishman, R., "Computational Linguistics", Cambridge University Press, 1986.

Hart,P.E., Nilsson,N.J., and Raphael, B., "A formal basis for the heuristic determination of minimum cost paths", IEEE Trans. Syst. Science and Cybernetics, SSC-4(2), 100-107.

Hayes,P.J., "The frame problem and related problems in Artificial Intelligence", in Artificial Intelligence and Human Thinking,Elithorn, A., and Jones, D., (eds.), San Francisco, Jossey-Bass, 1973.

Hayes, P.J., "Some Problems and Non-Problems in Representation Theory", Proc. AISB Summer Conference, University of Sussex, pp 63-79, 1974.

Hayes, P.J., "The Logic of Frames", Frame Conceptions and Text Understanding, pp 46-61, edited by D. Metzing, Walter de Gruyter & Co., Berlin, 1979.

Hayes, P.J., "The Naive Physics Manifesto", in Expert Systems in the Micro Electronic Age, ed. D.Michie, Edinburgh University Press, 1979(b).

Hayes, P.J., "The Second Naive Physics Manifesto", in Formal Theories of the CommonSense World, Ed. by Hobbs,J.R., and Moore,R.C., pp 1-36, Norwood NJ. Ablex Publishing Corporation, 1985.

Hayes-Roth,B., "The Blackboard Architecture: A General Framework for Problem Solving?", Heuristic Programming Project, Report no HPP-83-30, Stanford

University, May, 1983.

Hayes-Roth, F., Waterman, D.A., and Lenat, D.B., "Building Expert Systems", Addison Wesley, London, 1983.

Hendrix, G.G., "The representation of semantic knowledge", In Speech Understanding Research: Final Technical Report, Ed. Walker,D.E., Stanford Research Institute, Menlo Park, California, 1976.

Henning,G., Leone,H., Stephanopoulos,G.,"MODEL.LA parts 1 and 2", Reports LISPE-89-048, LISPE-89-049, Massachusetts Inst. Tech., 1989.

Hewitt, C., "Description and theoretical anlalysis (using schemata) of PLANNER, a language for proving theorems and manipulating models in a robot", Rep. No. TR-258, AI Laboratory, Massachussetts Institute of Technology, 1972.

Hill, R. "LUSH Resolution and its completeness", DCS Memo No 78, A.I. Dept., Edinburgh University.

Hillier,F.A., and Lieberman, G.J., "Introduction to Operations Research", San Francisco CA, Holden-Day Inc, 1980.

Horn, B.L., "An Introduction to Object Oriented Programming, Inheritance and Method Combination", Reprt CMU-CS-87-127, Carnegie Mellon Uni., 1988.

Huang,Y.W., and Fan,L.T.,"Designing an object relation hybrid database for chemical process engineering", Comp. and Chem. Eng., 12(9/10), pp 973-985, 1988.

Hudson, R., "Word Grammar", Blackwell Inc., Oxford, 1984.

Hutton, D., "Knowledge Based Flowsheeting Report", Internal Document, Dept. Chem. Eng., Edinburgh University, 1990.

Hutton,D., Ponton,J.W., and Waters,A., "AI applications in process design, operation and safety", The Knowledge Engineering Review, 5(2), 1990.

Ibaraki, T., "The power of dominance relations in branch and bound Algorithms", JACM 24(2):264-79.

Jackson, P., "Introduction to Expert Systems", Addison Wesley Pub., Reading, Mass., 1986

Jackson, P.,"A Brief Review of Knowledge Representation Tools and Techniques", DAI Research Paper No. 343, University of Edinburgh, 1987.

James, A., "Selection Criteria for an Integrated CAE System", PSE 85, Institution of Chemical Engineers, Sysmposium Series No 92, 1985.

Johnson-Laird, "Mental Models", Cambridge University Press, 1983.

Johnson,L., and Keravnou,E.T., "Expert Systems TEchnology", Abacus Press, 1985.

Jones, J.K., and Liles, J.A., "Integrated Process Design Systems can boost productivity by over 50%", ChemAsia 83, Computers in Design, Singapore, 1983.

Jones, N., "An expert system for first aid", MSc Thesis, Department of Artificial

Intelligence, University of Edinburgh, 1984.

KEE User's Manual, Intellicorp. Inc., 1985.

de Kleer,J., and Brown,J.S.,"A qualitative physics based on confluences", Artificial Intelligence, 24, pp7-83, 1984.

de Kleer, J., Doyle, J., Steele, G.L. and Sussman, G.J., "AMORD: Explicit Control of Reasoning", Proc. Symposium on Artificial Intelligence and Programming Languages, SIGPLAN Notices 12(8), and SIGART Newsletter, No 64, August 1977.

Kirkwood, R.L., Locke,M.H., Douglas, J.M., "A prototype expert system for synthesising chemical process flowsheets", Comp. and Chem. Eng., 12(4), pp 329-343, 1988.

de Kleer, "An Assumption based TMS", Artificial Intelligence, 28, pp127-162, 1986(a).

de Kleer, "Extending the ATMS", Artificial Intelligence, 28, pp163-196, 1986(b).

de Kleer, "Problem Solving with the ATMS", Artificial Intelligence, 28, pp197-224, 1986(b).

Kling, R.F, "A Paradigm for Reasoning by Analogy", Artificial Intelligence, 2, 1971.

Knowledge Craft, Knowledge Craft Reference Manual, Carnegie Group Inc., 1986.

Knuth, D.E., and Moore, R.W., "An Analysis of alpha-beta pruning", Artificial Intelligence, 6(4), p293-326, 1975.

Kocis, G.R., and Grossmann, I.E., "Computational Experiences in Solving MINLP Problems with DICOT", PSE 88, Third Int. Symposium on Process Sytems Engineering, 28 August - 2 Sept. Inst. of Engineers, National Conf. Publication No 88/17,Sydney, Australia, 1988.

Kosslyn, S.M., "Images and Mind", Cambridge, Mass., Harvard University Press, 1980.

Kowalski,R.A., and Kuehner,D, "Linear Resolution with Selection Function", Artificial Intelligence 2, 227-260, 1971.

Kowalski,R.A., "A proof procedure using connection graphs", JACM 22(4), 572-595, 1975.

Kowalski, R., "Logic for Problem Solving", Elsevier, New York, 1979.

Kramer, M.A., "Expert Systems for Process Fault Diagnosis", A General Framework, Proceedings of FOCAPO, Park City, Utah, 1987.

Kripke,S.A., "Semantical Considerations on Modal Logic", in Reference and Modality, L.Linsky, pp63-72, Oxford University Press, 1971.

Kripke,S.A., "Naming and Necessity", Davidson and Harman, 1972.

Kuipers,B., "Qualitative Simulation", Artificial Intelligence, 29, pp 289-338, 1986.

Kumar, V, Kanal, L.N., Artificial Intelligence, 21, p179-198, 1983.

Lawler, E.L., and Wood, D.E., "Branch and Bound methods: a survey", Operations Research 14(4) p699-719, 1966.

Leesley,M.E., Buchmann,A., and Mulraney, D., "An approach to a largely integrated systems for the computer aided design of chemical process plants", Proc. Int. Congr. Contribution of Computers to the Development of Chemical Engineering and Industrial Chemistry, Paris, pp61-66, 1978.

Levesque, H.J. and Brachman, R.J., "A Fundamental Tradeoff in Knowledge Representation and Reasoning (Revised Version)", Readings in Knowledge Representation, edited by R.J. Brachman and H.J. Levesque, Morgan Kaufmann Publishers Inc., Los altos, CA, 1985.

Levey, P., "Modelling Cognition: some current issues", in Modelling Cognition, ed. Morris, P., John Wiley and Sons, 1987.

Lien,K., Suzuki,G., and Weterberg,G., "The Role of Expert Systems Technology in Design", Report EDRC-06-13-86, Carnegie Mellon University, 1986.

Lindley,D.V, "Making Decisions", Wiley (2nd Edition), 1985.

Linhoff,B., et al,"A User Guide on Process Integration for the Efficient Use of Energy", I.Chem.E., Rugby, 1982.

Locke,J., "An essay concerning human understanding",Abridged edtn., Woozley,A.D. (ed),London, Fontana/Collins, (1960).

Lorie, R., and Plouffe, W., "Relational Databases for Engineering Data",Proceedings of the Second International Conference on Foundations of Computer Aided Process Design, Snowmass, Colorado, CACHE Publications,1983.

Loveland, D.W., "Automated Theorem Proving: A Logical Basis", New York, North Holland., 1978.

Lu,M.D., and Motard, R.L., "Computer Aided Total Flowsheet Synthesis", Comp. and Chem. Eng., 9(5), pp 431-445, 1985.

Luckham, D.C., and Nilsson, N.J., "Extracting Information from resolution proof trees", Artificial Intelligence, 2(1), pp27-54,

McCarthy, J., "Epistemological Problems of Artificial Intelligence", Proc. IJCAI-77, Cambridge, MA, pp 1038-1044, 1977.

McDermott, D. and Doyle,J., "Non-monotonic Logic I", Artificial Intelligence, 13, pp 27-39, 1980.

McDermott, D., "Artificial Intelligence meets Natural Stupidity", in J.Hagueland, Mind Design, Cambridge MA. MIT press, 1981.

McDermott, D., "Non-monotonic Logic II: non-monotonic modal theories", TACM, 29(1), pp33-57, 1982.

McDermott, J., "R1: An expert in the computer system domain", Proc. of AAAI-80, p269-271, 1980.

McDermott, J., "R1 Revisited: Four years in the trenches", AI Magazine, Autumn, pp21-32, 1984.

McDermott, J., and Sussmann, G.J., "The CONNIVER Reference manual", MIT AI Lab, Memo 259, 1972.

Mah, R.S.H., "Effects of Thermophysical Property Estimation on Process Design", Computers and Chem. Eng., Vol. 1, pp 183-189, 1977.

Manna, Z., "Mathematical Theory of Computation", McGraw-Hill,New York, 1974.

Manna, Z., and Waldinger, R., "A deductive Approach to program synthesis", in IJCAI-6, pp 542-551, 1979.

Marciszewski,W., "Dictionary of Logic", The Hague, Martins Nijhoft, 1981.

Martelli, A., and Montanari, U., "From dynamic programming to search algorithms with functional costs", in IJCAI, 4, pp345-350, 1975.

Martelli, A., and Montanari, U., "Optimising decision trees through heuristically guide search", Comm. ACM, 21, 1025-1039., 1978.

Matthew, I. and Dietz, K., "Complete Process Plant Design Using the Intergraph System", Eurochem 83, Inter. Chem. and Proc. Eng. Conference, UK. 1983.

Miller, G.A., and Johnson-Laird, P.N., "Language and Perception", Cambridge University Press, 1976.

Minsky, M., "A Framework for Representing Knowledge", in the Psychology of Computer Vision (ed. Winston P.H.) McGraw Hill. New York, 1975.

Montague,R.," Formal Philosophy: Selected Papers", New Haven, Yale University Press, 1974.

Moszkowski,B., "Executing Temporal Logic Problems," Cambridge University Press, 1986.

Moore, R.C.," Reasoning from incomplete knowledge in a procedural deductin system", AI-TR-347, Artificial Intelligence Laboratory, Massachussets Institute of Technology, 1975.

Moore, R.C., "Possible World semantics for auto-epistemic logic", Proc. AAAI-Workshop on Non-monotonic reasoning, New Palatz, New York, pp 344-354, 1984.

Moore, R.C., "Semantical considerations on non-monotonic logic", Artificial Intelligence, 25(1), pp 75-94, 1985.

Moore, R.C., "Autoepistemic logic", in Smets et al 1988, see below.

Moore,J, and Newell,A., "How can Merlin Understand?", in L.Gregg, Knowledge and Cognition, Hillsdale, New York, pp201-310, 1973.

Morris, P., "Modelling Cognition", John Wiley and Sons, 1987.

Motard, R.L., "Computer technology in Process Systems Engineering", Computers and Chemical Engineering, 7(4), pp 483-491, 1983.

Motard,R.L., "Integrated Computer-Aided Process Engineering", Proc. of PSE88, Inst. of. Eng., Sydney, Australia, 1988.

Murray, N.V., "Completely Non-Clausal Theorem Proving", Artificial Intelligence, 18, 1, 67-85, 1982.

Nado,R., and Fikes,R., "Semantically Sound Inheritance for a Formally Defined Frame Language with Defaults", pp 443-447, Proc of AAAI, 1987..

Newell, A., and Simon, H.A., "Human Problem Solving", Englewood Cliffs, NJ, Prentice Hall, 1972.

Nilsson, N.J., "Principles of Artificial Intelligence", Springer-Verlag, 1982.

Niida, K., Yagi, H., Umeda, T., "An application of data base management systems to process design", Computers and Chem. Eng., Vol. 1, pp33-40, 1977.

Niida,K., Koshijima,I., and Umeda,T., "Diagnosis Kit, Proc. of PSE88, Inst. of Eng., Sydney, Australia, 1988.

Norman,D.A.,and Rumelhart,D.E., "Explorations in Cognition", Freeman, San Francisco, California, 1975.

Palay, A.J. "Searching with Probabilities", Pitman Publishing, London, 1985.

Pearl, J., "Asymptotic properties of minimax trees and game-searching procedures", Artificial Intelligence 14(2), pp113-38, 1980.

Pearl, J., "Machine Intelligence", 4(4), pp392-399, July, 1982.

Pearl, J., "Heuristics, Intelligent Search Strategies for computer Problem Solving", Addison Wesley, Reading, Mass., 1984.

Pereira, L.M., and Porto,A., "Selective Backtracking for Logic Programs", Proc. of 5th Automatic Deduction Conf., Pub. in. Lecture Notes in Computer Science, Springer-Verlag, 1980.

Piaget, "Play, Dreams and Imitations in Childhood", New York, Norton, 1951.

Piela,P.C., "ASCEND, An Object oriented environment for modelling and analysis", PhD Thesis, Cept. Chem. Eng., Carnegie-Mellon University, Pittsburgh PA, 1989.

Pohl, I., "First results on the effect of error in heuristic search", In Machine Intelligence 5, ed. Meltzer, B., and Michie, D., pp219-236, New York, American Elsevier, 1970.

Pohl, I., "The avoidance of relative catastrophe, heuristic competence," genuine dynamic weighting and computational issues in heuristic problem solving, Proc. IJCAI 3, Stanford, 1973.

Ponton, J.W., "ESSPROS User Notes", Internal Document, University of Edinburgh, 1985.

Pospesel, H., "Introduction to Logic : Predicate Logic", Englewood Cliffs New Jersey, Prentice Hall, 1976.

Proctor, S.I., "Challenges and Constraints in Computer Implementation and Applications", Proceedings of the Second International Conference on Foundations of Computer Aided Process Design, Snowmass, Colorado, CACHE Publications,1983.

Prolog, Edinburgh Prolog (The New Implementation) User's Manual, Version 1.4, A.I.Applications Institute, Edinburgh University, 1986.

Pulman, S.G.," Word Meaning and Belief", Helman, London, 1983.

Quillian, M.R., "Word Concepts: A Theory and Simulation of Some Basic Semantic Capabilities", Behavioural Science, 12, pp 410-430, 1967.

Rebok,R., et al, "QLISP: A langauge for the Interactive Development of Complex Systems", SRI Technical Note, SRI International, AI Centre, Stanford, 1976.

Reiter, R., "On Reasoning by Default", Proc. TINLAP-2, Theoretical Issues in Natural Language Processing-2, University of Illinois at Urbana-Champaign, pp 210-218, 1978.

Reiter,R., "A logic for default reasoning", Artificial Intelligence, 13, pp 81-132, 1980.

Reiter,R., and Criscuolo,G., "On interacting defaults", Proc. IJCAI-81,pp270-276, 1981.

Reklaitis,G.V., Rippin,D.W.T., Eds., Special Issue on "Artificial Intelligence in Chemical Engineering - Research and Development", Comp. and Chem. Eng., 12(9/10), 1988.

Roberts,R.B., and Goldstein,I.P., "The FRL Manual", AI Memo No 409, MIT Artifical Intelligence Laboratory, Cambridge, MA, 1977.

Robinson, J.A.,"A machice oriented logic based on theresolution principle", JACM,12, 25-41, 1965.

Roizen, P., and Pearl, J., Artificial Intelligence 21, p190-220, 1983.

Rosch,E.H. et al, "Basic Objects in Natural Categories", Cognitive Pstchology, 8, pp382-439, 1976.

Rosch,E., and Lloyd,B., "Cognition and Categorisation", Hillsdale NJ, Lawrence Erlbaum Associates, 1978.

Rosch,E., and Mervis,C., "Family Resemblances: Studies in the Internal Structure of Categories", Cognitive Psychology, 7, pp573-605, 1975.

Ross, P., "Expert Systems Course", DAI Teaching Paper No. 1,1985.

Ross, P., "CRESS - a Credible Expert System Shell", User's Notes, Department of Artificial Intelligence, University of Edinburgh, 1987.

Rumelhart,D.E. and Norman,D.A., "Active Semantic Networks as a model of human memory", in Proceedings of 3rd Int. Joint Conf. on AI., 1973.

Sacerdoti, E.D., "Planning in a hierarchy of abstraction spaces", Artificial Intelligence, 5(2), pp 115-135, 1974.

Sacerdoti, E.D., "A Structure for Plans and Behaviour", Elsevier, North Holland, 1977.

Said, K.M., "Formalisation of Reasoning in Judicial Expert Systems through Deontic

Logic", in Proc. of 1st Int. Expert Systems Conf., 1985.

Sargent, R.W.H., "Process Systems Engineering: Challenges and Constraints in Computer Science and Technology", Proceedings of the Second International Conference on Foundations of Computer Aided Process Design, Snowmass, Colorado, CACHE Publications,1983.

Sargent,R.W.H., "Process Design - What Next?", pp 529-553, Proc. of Foundations of Computer Aided Design, July 1989, Published by Elsevier, Amsterdam, 1990.

de Saussure, F., "Course in General Linguistics", London, Peter Owen, 1960.

Schank,R., "The structure of episodes in memory", in D.G.Bobrow and A.Collins, Representation and Understanding, Academic Press, New York, 1975.

Schmidt, J.W., and Brodie, M.L., "Relational Database Systems - Analysis and Comparison", Springer Verlag, New York, 1983.

Shafer, G., "A mathematical theory of evidence", Princeton NJ, Princeton University Press, 1976.

Shapiro,S.C., "A net structure for semantic information storage, deduction and retrieval", Proc. of 2nd Int. Joint Conf. on AI. pp 512-523, 1971.

Shortliffe, E.H.," Compute-Based Medical Consultations", MYCIN, New York, Elsevier, 1976.

Shortliffe, E.H., and Buchanan,B.G., "A model of inexact reasoning in medicine", Mathematical Biosciences, 23, pp 351-379, 1975.

Simmons,R.F.,"Semantic Networks: Their computation and use for understanding English sentences", pp 63-113, in Computer models of thought and Language, ed. Schank,R.C., and Colby, K.M., Freeman, San Francisco, 1973.

Smets,P., Mamdani,E.H., Dubois,D., Prade, H., "Non-Standard Logics for Automated Reasoning", Academic Press, 1988.

Smith, J.M., and Smith, D.C.P., "Database Abstractions: Aggregation and Generalisation", ACM TODS, 2, pp105-133, 1977.

Smoke, K.L., "An objective study of concept formation", Psychological Monographs, 42(191), 1932.

Sriram, D., "A Bibliography on Knowledge Based Expert Systems in Engineering", Report DRC-12-23-84, Carnegie Mellon University, 1984(b).

Sriram, D., et al., "Knowledge Based Expert Systems: An Emerging Technology for CAD in Chemical Engineering", Report DRC-06-76-84, Carnegie Mellon University, 1984(a).

Stallman, R.M., and Sussmann, G.J., "Forward reasoning and dependency directed backtracking in a system for computer aided circuit analysis", Artificial Intelligence, 9(2), 135-196, 1976.

Stefik,M., Bobrow,D.G., Mittal,S., and Conway.L, "Knowledge Programming in LOOPS", AI Magazine Autumn, 1983.

Stephanopoulos, G., Johnston,J., Kriticos, T., Lakshmanan,R., Mavrovouniotis,N., and Siletti,C., "Design Kit: An Object Oriented Environment for Process Engineering", Comp. and Chem. Eng., 11(6), pp 655-674, 1987.

Stephanopoulos,G., "Artificial Intelligence and Symbolic Computing, in Process Engineering Design",Proc. of Foundations of Computer Aided Design,July 1989, Published by Elsevier, Amsterdam, 1990.

Sterling, L. and Shapiro, E., "The Art of Prolog", MIT Press, Cambridge, Massachusetts, 1986.

Stickel, M.E., "Theory Resolution: building in non-equational theories", In Proc. of the AAAI 83, Washington, Colombia, 1983.

Stockman, G.C., Artificial Intelligence 12, pp 179-196, 1979.

Struthers, A., "Literature Survey on Process Synthesis", Hons. Research Project, Chem. Eng. Dept., Edinburgh University, 1984.

Struthers, A., "CLAP User Document", Dept. Chem. Eng., Edinburgh University, 1987.

Szolovits,P., and Pauker, S.G., Categorical and Probabilistic Reasoning in Medical Diagnosis, Artificial Intelligence, 11, pp115-144, 1978.

Takamatsu, T., "The Nature and Role of Process Systems Engineering", Computers and Chem. Eng., 7(4), pp203-218, 1983.

Tate, A., "Generating Project Networks", in IJCAI-5, pp888-893, 1977.

Tate, A. , A review of Knowledge Bases Planning Techniques, Proc. of 5th Technical Conference of British Computer Society on Expert Systems 1985, Cambridge University Press, 1986.

Thayse, A. (editor), "Form Standard Logic to Logic Programming", John Wiley & Sons, Chichester, 1988.

Touretzky,D.S., "The Mathematics of Inheritance Systems", Pitman, London, 1986.

Trickett, K.G., and Chaney, J.C., "PDMS: plant layout and piping design", Computer Aided Process Plant Design, pp 1123-1182, Gulf Publishing Company, 1982.

Tsubaki, M., "DBMS with the cpability of program management and user language management - DPLS", Inf. Proc. Soc. Jap., 17(10), pp921-926, 1976.

Tsubaki, M., and Motard, R.L., "Data based process simulation", Computer Applications in Chemical Engineering, proc. of 12th European Symposium, CACE 79, volume 2, pp741-753, Montreux, Switzerland, April, 1979.

Turner,R. "Logics for Artificial Intelligence", Ellis Horwood, 1984.

Ullman, J.D., "Principles of Database Systems", Computer Science Press, Maryland, USA, 1982.

van Melle, W., Scott, A.C., Bennet, J.S. and Pears, M.A.S., "The EMYCIN manual", Report No HPP-81-16, Heuristic Programming Project, Computer Science

Dept., Stanford University, 1981.

Vancoille,M.J.S., Bogaerts, W.F.L., "Practical Building of Expert Systems for Materials Selection and Corrosion Problems", in Expert Systems in Structural Safety Assessment, ed. MPA Stuttgart, Springer Verlag, Heidelberg, 1989.

Vancoille,M.J.S., Bogaerts, W.F.L., Perdieus,F.,"AI Based Corrosion Risk Analysis in Oil Refinery Operation", Internal Report, Dept. MTM - Engineering, University Leuven, Belgium, 1989.

Venkatasubramanian,V., and Dhurjati,P., "An Object-Oriented Knowledge Base Representation for the Expert System FALCON", Proc. of Foundations of Computer Aided Process Operations, Park City, Utah, 1987.

Venkatasubramanian,V., and Rich,S.H., "An Object-Oriented Two Tier Architecture for Integrating compiled and deep level knowledge for process daignosis", Comp. and Chem. Eng., 12(9/10), pp 903-921, 1988.

Warren, D.H.D., "WARPLAN, A system for generating plans," Memo 76, Dept. of Computational Logic, University of Edinburgh, 1974.

Warren, D.H.D.," Implementing Prolog - Compiling Logic Programs 1 and 2", DAI Research Reports 39 and 40, University of Edinburgh, 1977.

Warren, D.H.D.,Pereira,L.M., and Pereira, F.C.N., "Prolog the language and its implementation compared with LISP", Proceedings of the Symposium on AI and Programming Languages, p109-115, reprinted in SIGPLAN Notices/ SIGART Newsletter (1977).

Waters,A., and Ponton,J.W.," Qualitative Simulation and Fault Propagation in Process Plants", Chem.Eng.Res.andDes., 67, pp407-422, 1989.

Weatherhill,T., and Cameron,I., "Preliminary Hazop Studies using Expert Systems", Proc. of PSE 88, Inst. of Eng. Australia, Aug., 1988.

Wehe,R., Lien,K., and Westerberg,A., "Control Considerations for a Separations System Design Expert", Report EDRC-06-29-87, Engineering Design Research Centre, Carnegie Mellon University, 1987.

Weiner,R.S., Pinson, L.J., "Object Oriented Programming and C++", Addison-Wesley, 1988.

Weiss,S.M. and Kulikowski, C.A., "A Practical Guide to Designing Expert Systems", Chapman and Hall, London, 1983.

White, A.P., "Inference Deficiences in Rule Based Expert Systems", in Research and development in Expert Systems, Ed. M.A.Bramer, Cambridge University Press,pp 39-51, 1985.

Whitney, J.S., "The Plantman Computer Aided Design", Eurochem 83, Inter. Chem. and Proc. Eng. Conference, UK. 1983.

Weinreb,D., and Moon,D., "LISP machine manual", Symbolics Inc. 1981.

Wilkins, D., "A non-clausal theorem proving system", Proceedings of the AISB Summer conference, Brighton, UK, 1974.