

Knowledge-based Design Support and Inductive Learning

Ming Xi Tang

DEPARTMENT *of* ARTIFICIAL INTELLIGENCE

Submitted to the Science and Engineering Faculty for the Degree of
Doctor of Philosophy

at the

University of Edinburgh
June, 1996



Declaration

I declare that this thesis has been composed by myself. Some of the work presented was carried out by me as part of the Castlemaine project, from 1990 to 1992. My contributions to this project included: knowledge acquisition in the domain of drug design, and the design, specification, and implementation of the Castlemaine drug design support system.

Acknowledgements

The work presented in this thesis was partly supported by a Sino-British Friendship Scholarship from the British Council which allowed me to join the department of Artificial Intelligence, University of Edinburgh in 1988 as a visiting researcher. I subsequently joined the AI in Design research group in the department and started working on an Alvey large scale demonstrator project 'Design to Product' and an AI-based design support system known as Edinburgh Designer System. I would like particularly acknowledge the help from Dr. Karl Millington, Dr. Brian Logan, Dr. Nils Tomes, Alistair Conkie, Jim Doheny, Andy Robertson, Dave Corne, Amaia Bernaras and Gillian Morrice.

My involvement in the Castlemaine project was supported by grant number GR/F/3567.8 from the U.K. Science and Research Council and through the UK Department of Trade and Industry's Information Directorate.

I would like to acknowledge Logica Cambridge Ltd, British Bio-technology Ltd. and CamAxys Ltd. for their involvement in and financial commitment to the Castlemaine project. They provided necessary background knowledge for the design problems chosen as cases to test the computational techniques developed in this thesis.

The completion of this thesis is due to strong support and excellent supervision from my supervisors, Dr. Tim Smithers who invited me in 1988 to join the AI in Design Research group in the Department of Artificial Intelligence at the University of Edinburgh and encouraged me to develop inductive learning approach within and after the Castlemaine project, and Dr. Peter Ross who supported me not only in theoretical aspects but also in software development by allowing me to use his C-based ATMS in both the Castlemaine project and in the development of ATMB architecture which is part of the research of this thesis.

Finally, I would like to acknowledge the support from Dr. Chris S. Mellish and Dr. Alex B H Duffy whose comments have been usefully incorporated in this thesis.

Abstract

Designing and learning are closely related activities in that design as an ill-structure problem involves identifying the problem of the design as well as finding its solutions. A knowledge-based design support system should support learning by capturing and reusing design knowledge. This thesis addresses two fundamental problems in computational support to design activities: the development of an intelligent design support system architecture and the integration of inductive learning techniques in this architecture.

This research is motivated by the belief that (1) the early stage of the design process can be modelled as an incremental learning process in which the structure of a design problem or the product data model of an artefact is developed using inductive learning techniques, and (2) the capability of a knowledge-based design support system can be enhanced by accumulating and storing reusable design product and process information.

In order to incorporate inductive learning techniques into a knowledge-based design model and an integrated knowledge-based design support system architecture, the computational techniques for developing a knowledge-based design support system architecture and the role of inductive learning in AI-based design are investigated. This investigation gives a background to the development of an incremental learning model for design suitable for a class of design tasks whose structures are not well known initially.

This incremental learning model for design is used as a basis to develop a knowledge-based design support system architecture that can be used as a kernel for knowledge-based design applications. This architecture integrates a number of computational techniques to support the representation and reasoning of design knowledge. In particular, it integrates a blackboard control system with an assumption-based truth maintenance system in an object-oriented environment to support the exploration of multiple design solutions by supporting the exploration and management of design contexts.

As an integral part of this knowledge-based design support architecture, a design concept learning system utilising a number of unsupervised inductive learning techniques is developed. This design concept learning system combines concept formation techniques with design heuristics as background knowledge to build a design concept tree from raw data or past design examples. The design concept tree is used as a conceptual structure for the exploration of new designs.

The effectiveness of this knowledge-based design support architecture and the design concept learning system is demonstrated through a realistic design domain, the design of small-molecule drugs one of the key tasks of which is to identify a pharmacophore description (the structure of a design problem) from known molecule examples.

In this thesis, knowledge-based design and inductive learning techniques are first reviewed. Based on this review, an incremental learning model and an integrated architecture for intelligent design support are presented. The implementation of this architecture and a design concept learning system is then described. The application of the architecture and the design concept learning system in the domain of small-molecule drug design is then discussed. The evaluation of the architecture and the design concept learning system within and beyond this particular domain, and future research directions are finally discussed.

1.2.2 The Building Block Approach	8
1.2.3 The Design Prototype Approach	9
1.2.4 The Constraint-based Approach	10
1.3 Supporting Design Using Machine Learning Techniques	14
1.3.1 Inductive Learning	15
1.3.2 Explanation-Based Learning	17
1.3.3 Case-Based Reasoning	18
1.4 Research Overview	20
1.5 Organisation of the Thesis	22

Chapter 2

Knowledge-based Design Support System Architectures	24
2.1 Knowledge Representation	25
2.1.1 Domain and Design Knowledge	25
2.1.2 Rule-based Representation	27
2.1.3 Object-Oriented Representation	28
2.1.4 Design Knowledge Source	30
2.1.5 Design Knowledge Base	30
2.2 Intelligent Control System	32
2.2.1 Blackboard Control	32
2.2.2 Review of Existing Architectures	34
2.2.2.1 Edinburgh Designer System (EDS)	34
2.2.2.2 HOBS	35
2.2.2.3 DICE	37
2.2.2.4 IFe	38
2.2.2.5 IDP	40
2.3 Design Context Management	41
2.3.1 Modelling Context to Design	41
2.3.2 Reasoning about Context	45
2.3.3 Design Context Management in DMS 2V	47

Table of Contents

Chapter 1

Introduction.....	1
1.1 The Design Process.....	1
1.2 Knowledge-based Design Support.....	3
1.2.1 Intelligent CAD	5
1.2.2 The Building Block Approach	8
1.2.3 The Design Prototype Approach.....	9
1.2.4 The Constraint-based Approach	10
1.3 Supporting Design Using Machine Learning Techniques.....	14
1.3.1 Inductive Learning.....	15
1.3.2 Explanation-Based Learning	17
1.3.3 Case-Based Reasoning.....	18
1.4 Research Overview	20
1.5 Organisation of the Thesis	22

Chapter 2

Knowledge-based Design Support System Architectures.....	24
2.1 Knowledge Representation	25
2.1.1 Domain and Design Knowledge	25
2.1.2 Rule-based Representation	27
2.1.3 Object-Oriented Representation.....	28
2.1.4 Design Knowledge Source	30
2.1.5 Design Knowledge Base.....	30
2.2 Intelligent Control System	32
2.2.1 Blackboard Control.....	32
2.2.2 Review of Existing Architectures.....	34
2.2.2.1 Edinburgh Designer System (EDS).....	34
2.2.2.2 HOBS	35
2.2.2.3 DICE.....	37
2.2.2.4 IFe	38
2.2.2.5 IDF.....	40
2.3 Design Context Management	41
2.3.1 Modelling Context in Design.....	41
2.3.2 Reasoning about Context	45
2.3.3 Design Context Management, an Example	47

2.4 User Modelling.....	50
Summary.....	53

Chapter 3

Inductive Learning Techniques.....	55
3.1 Supervised Inductive Learning.....	55
3.1.1 Concept Learning System.....	56
3.1.2 ID3.....	57
3.1.3 Inductive Logic Programming.....	58
3.2 Unsupervised Inductive Learning.....	58
3.2.1 Cluster Analysis.....	59
3.2.2 Hierarchical Clustering Techniques.....	60
3.2.3 The Agglomerative Approach.....	60
3.2.4 Similarity Measures.....	61
3.2.5 The Divisive Approach.....	62
3.3 Concept Formation Systems.....	63
3.3.1 EPAM.....	65
3.3.2 UNIMEM.....	67
3.3.3 COBWEB.....	68
Summary.....	70

Chapter 4

Inductive Learning and Design Support.....	72
4.1 Inductive Learning and Design.....	72
4.2 Application of Inductive Learning Techniques in Design.....	74
4.2.1 Design Knowledge Acquisition.....	75
4.2.2 Design Synthesis and Evaluation.....	76
4.2.3 Reuse of Past Design Plans.....	78
4.3 Utilisation of Design Heuristics in Learning.....	80
4.4 The Development of a Design Concept Learning System.....	80
4.4.1 A Design Concept Learning System.....	80
4.4.2 A Design Concept Learning System.....	84
4.4.2.1 Data Structure.....	86
4.4.2.2 Learning Strategies.....	88
4.4.2.3 Generalisation Rules.....	91
4.4.2.4 Implementation.....	92
Summary.....	93

Chapter 5

Knowledge-Based Support of Drug Design.....	95
5.1 The Metaphor of Lock and Keys.....	96
5.2 Drug Design Concepts.....	97
5.2.1 Molecules.....	98
5.2.2 Molecular Component.....	98
5.2.3 Molecular Fragment.....	99
5.2.4 Feature Pattern.....	102
5.2.5 Pharmacophore Description.....	103
5.2.6 Isosteres.....	104
5.3 Supporting Drug Design Using an Inductive Learning Approach.....	106
5.3.1 Data Preparation.....	107
5.3.2 Generating a Pharmacophore Description.....	108
5.3.3 Designing a New Molecule via Isostere Replacement.....	109
Summary.....	110

Chapter 6

An Architecture for Intelligent Design Support.....	114
6.1 An Overview of the Architecture.....	114
6.2 Knowledge Representation.....	117
6.3 The Control System.....	119
6.3.1 The Design Knowledge Source.....	119
6.3.2 Blackboard Data Structure and Control.....	120
6.4 Integration of ATMS and the Blackboard.....	123
6.4.1 Ross' ATMS.....	123
6.4.2 Integration.....	123
6.5 The Design Context Management System.....	126
6.6 Design Documentation System and User Interface.....	129
6.6.1 Design documentation system.....	129
6.6.2 User Interface.....	130
6.7 Application.....	132
Summary.....	133

Chapter 7

Inductive Learning of Pharmacophore Descriptions.....	141
7.1 An Overview of the Learning Strategy.....	141
7.2 Background Knowledge.....	142

7.3 The Learning Algorithms.....	144
7.3.1 Ranking.....	144
7.3.2 Identifying Feature Patterns.....	145
7.3.3 Building the Design Concept Tree.....	147
7.3.4 Concept Evaluation.....	149
7.4 A Working Example.....	150
Summary.....	162
Chapter 8	
Evaluation and Future Directions.....	164
8.1 Evaluation of the Architecture.....	165
8.1.1 Features of the Architecture.....	165
8.1.2 Limitations for the Architecture.....	166
8.2 Evaluation of the Learning Scheme.....	167
8.2.1 Features of the Design Concept Learning System.....	167
8.2.2 Application.....	168
8.2.3 Limitations.....	170
8.3 Evaluation of Integration.....	171
8.4 Future Directions for the Architecture.....	173
8.5 Future Directions for the Learning System.....	175
8.6 Applicability of the Architecture in Other Domains.....	176
Chapter 9	
Conclusions.....	179
Bibliography.....	181
Appendix A	
Drug Design Concepts and Rules.....	190
1. Smiles String Representation of Molecular Structure.....	190
2. Rules for Component Partition.....	191
3. Rules for Fragment Assembly.....	192
Appendix B	
Software Systems.....	193
1. The Blackboard System.....	193
2. The ATMS Interface.....	194
3. The ATMB Architecture.....	196

4. The Design Concept Learning System	198
---	-----

Appendix C

The Development Tool, GoldWorks II 200

1. Introduction	200
2. Knowledge Representation.....	200
2.1 Frames and Instances.....	200
2.2 Relations and Assertions	201
3. Rule-based and Object-Oriented Programming	202
3.1 Embedded Rules	202
3.2 Rule Syntax.....	202
3.3 Rule-based Inference	203
3.4 Object-Oriented Message Passing.....	203
4. Rule-based Control Facilities.....	204
4.1 Agenda Items	204
4.2 Rule Set	204
4.3 Sponsors	205
5. Dynamic Graphical Toolkit.....	207

List of Figures

Figure 1.1: Life Cycle of a Product.....	2
Figure 2.1: Basic Architecture for Intelligent CAD.....	24
Figure 2.2: A Simple Production Rule.....	27
Figure 2.3: An Object-Oriented Product Data Model.....	29
Figure 2.4: The Basic Structure of a Design Knowledge Base.....	31
Figure 2.5: Blackboard Control Strategy.....	33
Figure 2.6: An ATMS.....	46
Figure 2.7: Exploration of Alternative Design Solutions.....	48
Figure 2.8: Status of the System after Negotiation.....	50
Figure 3.1: Hierarchical Clustering Methods.....	60
Figure 3.2: Similarity Measures.....	62
Figure 4.1: Design Synthesis Via Inductive Learning.....	77
Figure 4.2: A Learning Model for Design Support.....	82
Figure 4.3: Design Concept Learning System.....	85
Figure 4.4: Data Structure of Design Concept Node.....	86
Figure 4.5: Data Structure of Design Concept Tree.....	87
Figure 4.6: Distance Matrix.....	88
Figure 5.1: Molecule Binding.....	96
Figure 5.2: Structure and Attributes of a Molecule.....	98
Figure 5.3: Component Partition.....	99
Figure 5.4: Molecular Fragment.....	100
Figure 5.5: From Molecule to Fragments.....	101
Figure 5.6: Feature Pattern.....	102
Figure 5.7: Pharmacophore description.....	104
Figure 5.8: Model of the Isostere Pair.....	105
Figure 5.9: Model of Indirect Drug Design Process.....	107
Figure 5.10: A Learning Setting for Drug Design.....	109
Figure 5.11: Interface for Molecule Selection.....	112
Figure 5.12: Identify Fragments.....	113
Figure 6.1: The Architecture.....	115
Figure 6.2: Frame-based Representation.....	117
Figure 6.3: Object Class Definition of a Molecular Component.....	118
Figure 6.4: Basic Data Connection Unit for the ATMS.....	119
Figure 6.5: Definition of Design Knowledge Source.....	120
Figure 6.6: The Data Structure of the Blackboard.....	121
Figure 6.7: Truth Maintained Blackboard Control Process.....	125
Figure 6.8: Exploring Design Solutions in Different Contexts.....	128
Figure 6.9: Design Documentation System.....	130
Figure 6.10: Making Decisions on Data and Method.....	136
Figure 6.11: Creating a New Design Branch.....	137
Figure 6.12: Designing New Drug Using an Isostere Library.....	138
Figure 6.13: Graphical Explanation of Domain Concepts.....	139
Figure 6.14: Graphical Explanation Justifications.....	140
Figure 7.1: Level of Concepts.....	143
Figure 7.2: Generating Feature Patterns.....	146
Figure 7.3: Generalisation of Secondary Properties.....	148
Figure 7.4: Molecule Examples.....	151
Figure 7.5: Input Data Structure.....	152
Figure 7.6: Molecule Ranking Result.....	153
Figure 7.7: Initial Design Concept Tree.....	154
Figure 7.8: Display of the Initial Design Concept Tree.....	155
Figure 7.9: A Node in the Design Concept Tree.....	156
Figure 7.10: The Display of the Final Design Concept Tree.....	157
Figure 7.11: List of Final Concept Nodes.....	158
Figure 7.12: A Pharmacophore Description.....	160
Figure 7.13: Relationship Between Examples and Concepts.....	161
Figure 8.1: Architecture of the IFM System.....	177
Figure A1: Smiles String Rules.....	190

Figure A2: Classification of atoms.....	191
Figure A3: Component Partition Rules.....	192
Figure B1: Top Blackboard Control Function.....	193
Figure B2: Creating a foreign function environment.....	195
Figure B3: Defining a foreign function.....	195

1.1 The Design Process

All human workmanship contains some element of design in which knowledge and expertise play an important role. Almost everything we interact with in our everyday life is a product of a design activity, which has involved some form of discovery and creativity. Design is a complex process in which information and knowledge of diverse sources are processed simultaneously by a team of designers involved in the life phases of a product. A good understanding of the design process is essential for us to know how computational techniques could best be utilised to support the design activities of human designers.

Axelrod stated that design is a *goal-directed, problem solving activity* [Axelrod 1970]. Feilden pointed out that *engineering design is the use of scientific principles, technical information and imagination in the definition of a mechanical structure, machine or system to perform pre-specified functions with the maximum economy and efficiency* [Feilden 1963]. Asimow regarded designing as *decision making, in the face of uncertainty, with high penalties for error* [Asimow 1962]. Matchett described the task of designing as *ascertaining the optimal solution to the sum of the true needs of a particular set of circumstances* [Matchett 1968]. Andreassen stated that *design is the common term in industry for a broad range of creative activities such as problem solving, product synthesis, product development and product planning* [Andreassen 1991].

The life phases of a product are illustrated in Figure 1. The task of design is to generate a description of an artefact that satisfies a given set of functional requirements within a given working environment. Three essential design activities early in the life phases of a product are *analysis, synthesis and evaluation*. When the problem of design is defined as the description of an artefact that satisfies a set of specifications, the process of mapping from the specifications to the description is *synthesis* while the process of mapping from design descriptions to behaviour characteristics (which are essentially similar to the specification properties) is *analysis or evaluation*. The early design process contains many cycles of these activities and each cycle is progressively less general and more detailed than the one before [Jones 1992]. The major aim of these activities is to explore the design problem and develop new ideas that can gradually be specialised into a detailed description that satisfies the design requirements. These design requirements are themselves subject to continuous modifications as the design process goes on.

Chapter 1

Introduction

1.1 The Design Process

All human workmanship contains some element of design in which knowledge and expertise play an important role. Almost everything we interact with in our everyday life is a product of a design activity, which has involved some form of discovery and creativity. Design is a complex process in which information and knowledge of diverse sources are processed simultaneously by a team of designers involved in the life phases of a product. A good understanding of the design process is essential for us to know how computational techniques could best be utilised to support the design activities of human designers.

Archer stated that design is *a goal-directed, problem solving activity* [Archer 1970]. Feilden pointed out that *engineering design is the use of scientific principles, technical information and imagination in the definition of a mechanical structure, machine or system to perform pre-specified functions with the maximum economy and efficiency* [Feilden 1963]. Asimow regarded designing as *decision making, in the face of uncertainty, with high penalties for error* [Asimow 1962]. Matchett described the task of designing as *ascertaining the optimal solution to the sum of the true needs of a particular set of circumstances* [Matchett 1968]. Andreasen stated that *design is the common term in industry for a broad range of creative activities such as problem solving, product synthesis, product development and product planning* [Andreasen 1991].

The life phases of a product are illustrated in Figure 1. The task of design is to generate a description of an artefact that satisfies a given set of functional requirements within a given working environment. Three essential design activities early in the life phases of a product are *analysis, synthesis* and *evaluation*. When the problem of design is defined as the description of an artefact that satisfies a set of specifications, the process of mapping from the specifications to the description is *synthesis* while the process of mapping from design descriptions to behaviour characteristics (which are essentially similar to the specification properties) is *analysis* or *evaluation*. The early design process contains many cycles of these activities and each cycle is progressively less general and more detailed than the one before [Jones 1992]. The major aim of these activities is to explore the design problem and develop new ideas that can gradually be specialised into a detailed description that satisfies the design requirements. These design requirements are themselves subject to continuous modifications as the design process goes on.

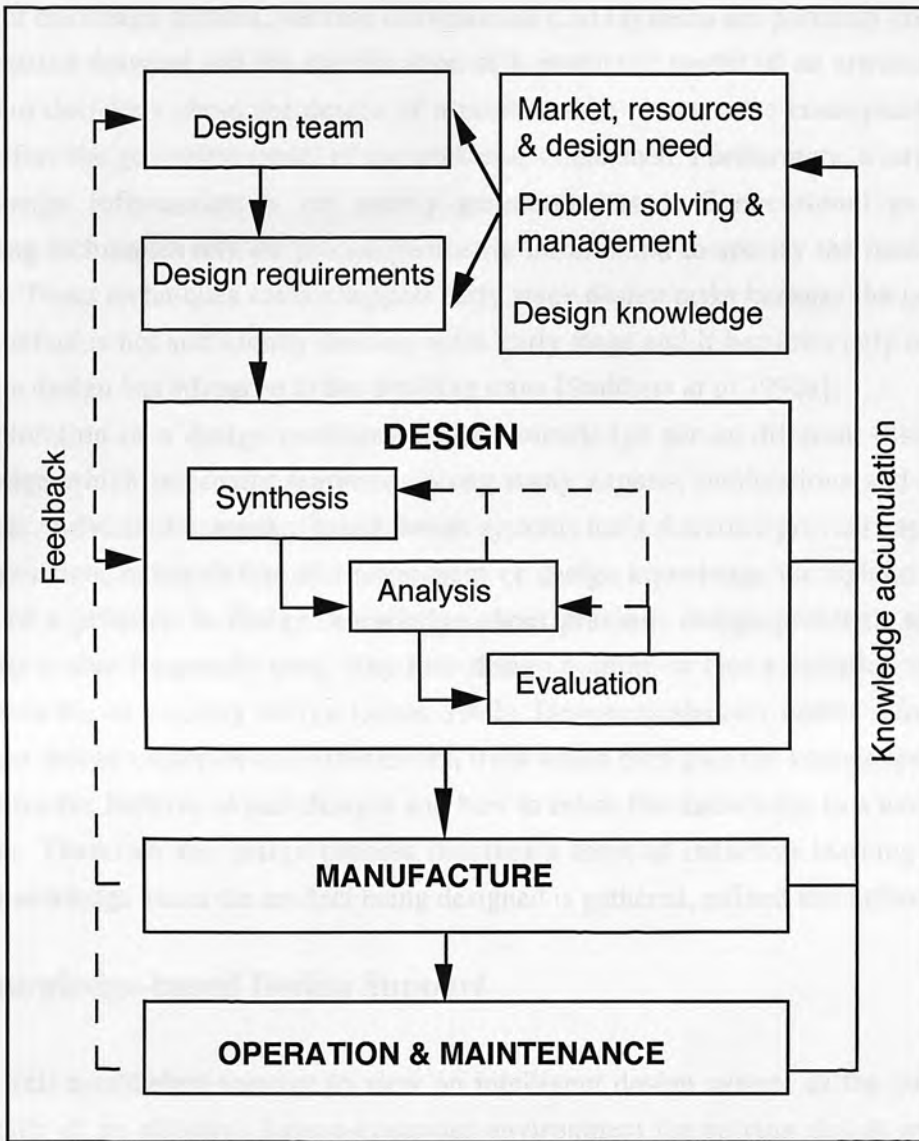


Figure 1.1: Life Cycle of a Product

Design is perhaps best characterised as an *ill-structured* problem to which a solution may not be fully and consistently specified until significant effort to understand the *structure of the design problem* or the *model of the artefact* has been made [Simon 1973]. Usually the initial design requirement does not contain, or imply, all the criteria by which an acceptable solution can be completely identified. Design, therefore, involves developing a complete and consistent requirement description as well as developing a solution to satisfy it. Consequently it is part of the design task to identify the possible inconsistencies in the design requirement and to restate it at the same time as the problem of the design is being *explored* [Smithers *et al* 1990a].

The computer is now an established factor in designers' lives. However, conventional Computer-Aided Design (CAD) approach offers limited support to the early and crucial

phases of the design process, because conventional CAD systems are primarily concerned with detailed drawing and the specification of a geometric model of an artefact. Many important decisions about the design of a product take place at the conceptual design stage before the geometric model of the artefact is established. Furthermore, a large set of vital design information is not purely geometric-based. Conventional geometric modelling techniques rely on precise geometric information to specify the model of an artefact. These techniques cannot support early stage design tasks because the geometry of an artefact is not sufficiently definite at the early stage and it becomes only available when the design has advanced to the detailing stage [Smithers *et al* 1990a].

Exploration of a design problem requires knowledge across different disciplines, knowledge which is usually scattered among many experts, publications and perhaps databases. Advanced computer-based design systems must therefore provide support for the acquisition, manipulation and refinement of design knowledge throughout the life phases of a product. In design, knowledge about previous design problems and their solutions is also frequently used. Any new design is more or less a variation of, or an innovation to, an existing design [Jones 1992]. Designers abstract useful information from past design examples and experiences, from which they gain the knowledge of how to improve the features of past designs and how to relate this knowledge to a new design problem. Therefore the design process involves a form of inductive learning, during which knowledge about the artefact being designed is gathered, refined and utilised.

1.2 Knowledge-based Design Support

It is a well established concept to view an intelligent design system as the computer-based side of an effective human-computer environment for solving design problems. Such an environment should relieve human designers from the routine and more mundane tasks encountered in the course of designing, allowing them to concentrate on high-level activity [Yoshikawa *et al* 1989].

A knowledge-based design support system is a decision support system to enable designers to explore the structures of design problems and their solutions by combining human design expertise with domain and design knowledge stored in a design knowledge base. Knowledge-based design support systems are different from traditional expert systems which use expert knowledge to help non-experts to solve those problems for which the experts have a good understanding and solutions. Knowledge-based design aims at combining design theory, AI methods and computational techniques to support designers through an integration of human intelligence and machine intelligence. As such, they must process a wider range of data and knowledge necessary for solving unfamiliar design problems than conventional CAD systems.

However, human designers will still continue to play the primary role throughout the design process. Knowledge-based design systems are therefore best seen as systems that support rather than replace human designers. This is so because with existing technology and our current limited understanding of the design process and design complexity, many design problems still require experienced human designers to analyse situations and to discover and resolve the inconsistencies and incompleteness of various design requirements and constraints. It is still difficult to represent and infer from every aspect of human design expertise in a computer system in a way which can be as effectively used to replace human designers.

Design as an intelligent behaviour lies in the ability of designers to identify a need and devise a solution, using the knowledge, materials and techniques which are already available, or which can be created. Specialist discoveries and general domain knowledge offer little to design problem solving until a good designer has combined them to form a new design problem structure. The view taken in this thesis is that the primary purpose of a knowledge-based design support system is to help designers to gather, organise, process and refine domain and design knowledge, and to obtain better design solutions.

Four main strategies may characterise the current developments in knowledge-based design systems [Wallace *et al* 1995]:

- Intelligent CAD;
- The building block approach;
- The prototype approach; and
- The constraint-based approach.

Intelligent CAD tries to extend the capability of a CAD system by employing heuristic knowledge on top of geometric models of the artefact. The building block approach decomposes the process into tasks that can be tackled by different classes of CAD tools and AI methods [Brown *et al* 1989 and Mostow *et al* 1987]. The prototype design approach divides design into three different activities: *prototype refinement*, *prototype adaptation* and *prototype creation* based on a library of design prototypes [Gero *et al* 1989 and Maher 1990]. Constraint-based design formulates design problems and requirements as interconnected networks in which design variables/parameters are related to each other through constraints. The networked design variables and constraints can then be manipulated or searched for solutions that satisfy all the requirements [Sussman *et al* 1980, Bowen *et al* 1992 and Smithers *et al* 1990b].

1.2.1 Intelligent CAD

MacCallum argues that intelligent CAD exists as a technique for improving current CAD systems [MacCallum 1990]. An intelligent CAD system plays an active rather than a passive role in the design process, by incorporating a significant amount of design knowledge into the system. Thus, in contrast to a conventional CAD system which merely documents the outcome of a design, an intelligent CAD system makes a meaningful contribution to the actual process of a design activity.

The development of intelligent CAD systems is often motivated by the need to present design as an intellectual process, rather than deal only with the narrow aspects of *design analyses, geometric modelling, or parametric variation*.¹

Designers are often more interested in systems capable of working and learning on a higher level of knowledge than geometry-based data models to support design decision making [MacCallum 1990]. Yoshikawa stated that an intelligent CAD system needs to satisfy at least the following three criteria [Yoshikawa *et al* 1989]:

- It must assist designers through all stages of the design process.
- It must assist designers in the design of a sufficiently large class of dissimilar objects.
- It must be practically possible to integrate it with other information processing systems such as CAD systems and databases.

TROPIC was a system developed by Jean-Claude Latombe which attempted to use AI methods to improve CAD systems by using knowledge representation and automated problem solving techniques [Latombe 1976]. The TROPIC system emphasises the use of three different types of knowledge which are usually absent in the traditional CAD systems:

- design problem knowledge - which states the requirements to be achieved by a design solution;
- domain knowledge - design objects and concepts from which the domain is defined and solutions are formed;

¹ Parametric variation is a process during which changes are propagated through a product model from a set of input variables via "rules of change" embedded in each component of the product model to output values. Propagation is uni-directional and any violation of the rules or constraints halts the process.

- problem solving knowledge - the strategies behind selecting particular problem-solving methods to be applied to particular design problems.

TROPIC was capable of accepting problem, domain and design problem-solving knowledge described by the user. It was also capable of gathering, selecting, and applying the means to the solution from this problem solving knowledge, and of producing a user-oriented trace of the solution process so that the user could easily correlate the steps of the solution process devised by the system. With TROPIC, the designer still had the essential responsibility for collecting the knowledge, and judging the final solution.

Dixon improved a CAD system for **V-belt** design by building a rule-based system based on a *design-evaluation-redesign* process model [Dixon 1986]. In this system, a rule-based language called **OPS5** was used to represent the design knowledge in terms of *design specification methods*, *design evaluation procedures*, and the *strategies for redesign*. If a design solution failed to satisfy a design requirement, then a set of redesign rules was triggered to modify the design based on the results of evaluation. However, a rule-based system, while useful for formulating well established design rules and heuristics in a knowledge-based design support system, can become inefficient when dealing with the complexity inherent in engineering design problems such as those involving complicated numerical computation as well as heuristic-based reasoning.

In mechanical engineering design, for example, some of the main design problems are the synthesis of a physical embodiment of a product that fulfils a given *functional requirement*, and the systematic checking of a design to ensure that all the components, their interfaces and features are consistent with the intended functions. Conventional CAD systems are capable of understanding the physical shape of a design. But they are unable to handle the knowledge and the reasoning which has led to the generation of that shape. *Functional modelling* is an approach that attempts to build intelligent CAD systems by a mapping from a function space to a solution concept space by using components and their interfaces by function rather than by shape [Johnson 1988, Smithers *et al* 1990b, Thornton 1993, Bracewell *et al* 1993 and Chakrabarti 1994].

For example, the main steps in the mechanical embodiment of a functional requirement can be highlighted as:

1. Decomposition of the overall function into simple sub-functions;
2. Identification of solution principles for these sub-functions;
3. Combining sets of solution principles into solution concepts;
4. Comparative evaluation of solution concepts;

5. Detailed design of the selected solution concept;
6. Generation of manufacturing information.

The basic idea of functional modelling in engineering design is to use abstract knowledge about the function and behaviour of components, parts and their causal relationships to derive initial design solution concepts, which are subsequently specified further in terms of structure and geometry. This is based on the perception that, at the early stage of design, the designers use abstract, and sometimes qualitative, knowledge to build up a model of an artefact for further exploration, and that conceptual design solutions can be derived without having to be precise about detailed geometric information such as dimensions [Johnson 1988].

For example, if the required function/sub-function is to transform a torque, then a functional component *shaft* can be selected without any precise information about its shape. An approximate sizing calculation can be carried out when the required speed and torque become available as input to the shaft. The methods and constraints for analysing the functional and spatial relations of components can be encapsulated with the components themselves. The computer performs increasingly precise analytical calculations as the design proceeds, and reminds the designer which parts of the design still require further specification by continuously checking the functional requirement and the constraints. For example, a bearing must be selected and located appropriately for the shaft at some stage of the design. If any part of a design is changed at any stage by any member of a team, the computer is able to predict the consequences by checking the constraints and interfaces associated with the changed part.

Schemebuilder is a knowledge-based design environment that integrates a number of software systems to support the designers in the rapid development of product models through the stages from conceptualisation to embodiment generation. These tools are integrated within a framework which is based on a *function mean tree development* method and a library of working principles to perform the task of generating alternative schemes (design solution concepts) [Bracewell *et al* 1993]. The Schemebuilder incorporated a Bond Graph [Thoma 1990] approach to a continuous-time energy system.

FuncSION is a system developed by Chakrabarti to synthesise design concept solutions based on simple functional requirements (input/output) and qualitative transformation rules. There are a number of design systems developed based on the functional modelling approach [Chakrabarti *et al* 1994]. A review of functionality modelling in design based on *Bond Graph, Qualitative Physics and Input/Output Transformation* can be found in [Winsor *et al* 1994].

1.2.2 The Building Block Approach

The generic task approach is proposed by Chandrasekaran for developing knowledge-based design and diagnosis applications [Chandrasekaran 1986 and Brown *et al* 1985, 1989, 1991]. Chandrasekaran classified design tasks into three categories:

1. Class 1 - design tasks in which the components of the object being designed are unknown;
2. Class 2 - design tasks in which the components of the object being designed are known, but the design plans, i.e. the methods of how to design, are unavailable in a compiled form; and
3. Class 3 - design tasks where ways of decomposing a design problem are already known and for which compiled design plans are available for each stage of design.

Class 3 design tasks are regarded by Chandrasekaran and his colleagues as *routine design tasks*. They use an approach called *hierarchical design by plan selection and refinement* to support these design tasks [Brown *et al* 1989]. This involves the realisation of a design that is achieved by invoking a design plan (or a design knowledge source) that makes some contributions to the design task, and also calls upon other relevant plans for further refinement. This approach to plan refinement is seen by Chandrasekaran *et al* as an elementary *building block* for design support which has a great deal of generality. The building block approach decomposes the process into tasks that can be tackled by different classes of CAD tool and AI methods such as *plan refinement*.

In addition to *hierarchical design by plan selection and refinement*, Chandrasekaran also identified five other generic tasks, for which knowledge representation and reasoning modules can be developed to form the building blocks for knowledge-based applications in general, and for knowledge-based design and diagnostic applications in particular. These include:

- *state abstraction* - which involves qualitative reasoning on design components;
- *abductive assembly* - which builds an explanatory composite based on highly plausible classificatory hypotheses;
- *knowledge-directed information passing* - which is used to infer new information from a structured representation and common knowledge within a domain;

- *hypothesis matching, or assessment* - which is concerned with the task of matching a concept against relevant data and determining a degree of appropriateness; and
- *hierarchical classification* - which builds hierarchical knowledge structures for diagnosis or information retrieval.

AIR-CYL was a system which was developed using this generic task approach, for designing air cylinders [Brown *et al* 1989]. This system used a hierarchy of *design agents*, called *specialists*, each with a set of *design plans* that accomplish a *design task*. The specialist at the root of the hierarchy contained a most abstract task, to design an air cylinder for example, while the lower-level specialists in the hierarchy contained less abstract design tasks, such as to design the sub-systems, or the components of an air cylinder. The plans of specialists called less abstract specialists to accomplish their tasks. A specialist had a number of design plans at its disposal for different conditions or for different design purposes. When a specialist's plan failed to accomplish a task, i.e., failed to satisfy the constraints that are required to accomplish the task, an alternative plan was invoked.

In AIR-CYL, failure of a specialist to accomplish its task was handled by *chronological backtracking* which backtracks to the point where an alternative plan is available. If all of a specialist's plans fail, then it returns a failure message to its parent, which considers another child specialist. The design fails if the root-level specialist returns a failure message, but is otherwise successful.

The building block approach works well for the parametric design problem such as the one performed by AIR-CYL, where the product models are well understood and whose dimensions and material types alone are to be selected and varied. But it does not suit non-routine design tasks that involve components configuration.

1.2.3 The Design Prototype Approach

Gero classified design as: *routine design*, where the functions, expected behaviours and structure variables of design are known, and the problem of design is one of instantiating values for structure variables; *innovative design*, where certain aspects of a defined design space need to be modified or extended because no existing solution within that space meets the design requirements; and *creative design*, in which an entirely new design problem space is to be defined [Gero *et al* 1989, Balachandran *et al* 1990, and Coyne *et al* 1990].

A *design prototype* represents a class of design elements and embodies the knowledge necessary to produce a particular instance of that class, or to evaluate the performance of

a given instance. It is a description of a class of generalised designs that embodies a notion of the design description to be produced in a parameterised form.

Design prototypes may be defined at various levels of abstraction, and may be related to each other by topological or structural properties. For example, a design prototype may have links, (such as *a-type-of*, *an-element-of* etc.), with some other design prototypes, thus forming a hierarchy of design components both from a taxonomic and an elemental view. Instances of design prototypes may inherit properties from other, more general, design prototypes through appropriate links. In this way, a design knowledge base can be developed as a prototype base, which contains all the prototypes in a given domain. This prototype base is, in many ways, similar to a frame-based system. But in addition to the basic feature of a frame-based representation, a design prototype also contains the links between frames, through a semantic modelling of design concepts and relationships between design objects.

Based on the representation scheme of design prototypes, three types of design activity are: *prototype refinement*, *prototype adaptation* and *prototype creation*. Prototype refinement involves instantiating the variables in a design prototype and determining their values. Routine design tasks fall into the prototype refinement category. Prototype adaptation becomes necessary when a design prototype is found to be inadequate in some way, and needs to be adapted to a new design problem, or made generally more useful for other design problem solving. Any modification to an existing prototype may result in new variables being created and new constraints being specified. Once a prototype is adapted and found useful, its specifications become one of the prototype refinements.

Prototype creation is seen as the ultimate design endeavour in this approach. So far research based on this approach has mainly concerned the problems of prototype refinement and prototype adaptation. Little progress has been reported on the issue of prototype creation and it remains a difficult subject.

1.2.4 The Constraint-based Approach

The Constraint-based approach models an engineering system as a constraint network or a hierarchical structure, i.e. collections of constraints that are interconnected by design variables [Sussman *et al* 1980, Mittal *et al* 1986, Marcus *et al* 1988., Young *et al* 1991 and Smithers *et al* 1990b]. Instantiation of this structure, or part of it, forms a basic structure of a new design problem. Computer-based constraint-based reasoning techniques are used to derive the values of unknown design variables from some initial data provided by the designers to form the solutions to the new design problem. Computational methods are used to detect constraint violation and to suggest alternative values for design variables. The application of design strategies or design methods is

reflected in the way in which design variables are created, constrained, and manipulated.

The basic operations in a constraint-based design system include:

- Describing design constraints and objectives (constraint specification);
- Specifying design variable values without violating any constraints (constraint satisfaction);
- Resolving conflicts and detecting inconsistencies (consistency maintenance); and
- Comparing alternative sets of variable values (exploring design contexts).

The advantage of the constraint-based approach is that it enables a designer to describe object behaviour as a system of relations among variables and to describe a design as a collection of constrained objects or as conceptual graphs [Gross *et al* 1988]. As the designer specifies the variable values, a *constraint manager* calculates consequences, checking consistency and propagating constraint. Any change in a variable value is propagated throughout the constraint network. This is essential for the designers to examine the basic relationships and trade-offs between design variables.

A constraint manager can be seen as a main inferencing support system in a constraint-based design system. In general such an inferencing system operates on algebraic, geometric, logical, and qualitative functions and performs the tasks of constraint simplification, evaluation, and propagation, solving algebraic equations or qualitative equations, and detecting inconsistencies and constraint violations. Usually, a constraint set may contain quantitative or qualitative equations which can be termed equalities and inequalities. For equalities, symbolic algebraic equation solving techniques such as PRESS [Bundy *et al* 1981] can be used. For inequalities, other techniques such as constraint-based reasoning are needed.

Algebra has at least two roles to play in constraint-based modelling of design. One is the notation for expressing constraints, and the other is a set of transformation rules which permit the derivation of the consequences of the constraints [Sussman *et al* 1980]. Since constraints are continually being added, deleted or modified during the process of specification and satisfaction, inconsistency may occur as a result. The constraint manager must be able to identify the redundant and conflicting constraints, and the possible sources of conflicts such as parameters causing the over-constrained situation. More importantly, the constraint manager provides useful information regarding the dependencies among the design variables. This information is useful for determining which of the known variables (inputs) derive the output parameters or conversely which of the output parameters are affected by changes in a particular input.

Dependency-directed backtracking is a technique that maintains the design states over asynchronous retraction. This means that whenever any one variable value is changed or retracted by the designer, all other variables dependent on that variable will also be changed or retracted, independently of when (chronologically) their values were set. Retraction and modification of variable values is an essential part of design exploration and generation of variants and an important contribution of dependency-directed backtracking is the ability to explain its reasoning [Serrano *et al* 1994].

Sussman's team [Sussman *et al* 1980] developed an interactive system organised around networks of constraints. They described a language for building and manipulating hierarchical constraint networks. Dependency analysis is used to spot and track down inconsistent subsets of a constraint set. Propagation of constraints involves performing symbolic manipulations on algebraic expressions. However, the dependency directed backtracking involved only the updating of the constraints when one parameter was changed in the propagation. The method failed in many cases owing to the lack of knowledge of the structure of the constraint network being modelled.

VT is a constraint-based design system for designing elevators [Marcus *et al* 1988]. VT uses a forward chaining and backtracking scheme to manipulate and maintain the constraint network. With VT, a new design is the result of entering an extension of the knowledge currently held in the system's knowledge base, i.e. the constraint network. When a design extension is proposed by a designer, the system identifies the necessary constraints from the constraint network, and then tests to see whether there is a constraint violation. If a violation is detected, the system chooses a so-called *fix* for the violation from a list of candidate values ranked by design cost. Once a fix is selected, the system uses the constraint network to correct any values that might be inconsistent with the value suggested by the fix. The design is considered complete if the last step in the extension is completed without detecting a constraint violation.

PRIDE is another constraint-based design system for designing paper handling systems in photocopiers [Mittal *et al* 1986]. PRIDE uses a hierarchy of design sub-goals to break down a design process into simpler and smaller design steps. Each sub-goal has a corresponding design method, or plan. Constraint violations are handled by dependency-directed backtracking via redesign advice. This redesign advice represents an alternative strategy for the original plan. The design is complete when the top-level goal is accomplished without any constraint violations.

Edinburgh Designer System (EDS) [Smithers *et al* 1990b] is an AI-based design support system that uses a number of computational techniques, including constraint satisfaction, to support mechanical engineering design, such as water turbine system design and fuel injection pump design. In the EDS, design and domain knowledge is represented in a *design knowledge base* (DKB) as a taxonomy of *module class definitions*. Each module class definition represents, in a declarative form, the concepts,

structures and constraints of a *component* (for example, a jet pump in a water turbine system, or a heat exchanger in a fuel injection system). These module class definitions are connected by a single inheritance scheme. For example, one module class definition can be related to the others through '*part-of*' or '*has-parts*' relations. Design is carried out through a *design exploration process*, where a designer explores the possible ways of constructing a new design using the available module class definitions, or tests different values of design variables within existing module class definitions. The EDS inference engines can infer on mathematical equations, shape equations, spatial relationships and functional and catalogue relations. These engines form the core of a constraint propagation system which is controlled using a blackboard control system.

GALILEO2 [Bowen *et al* 1992] is a constraint programming language that supports multiple context problem solving in life-cycle engineering using the concept of '*multiple perspectives*'. In GALILEO2, design variables and constraints can be viewed from different perspectives by the people involved in design, manufacturing and maintenance etc. A perspective is a small set of design variables and constraints. The variables within each perspective can be declared as being *local* or *overwritable*. The local variables within a perspective can only be modified by the person who created the perspective, whilst overwritable variables can be changed from within any other perspectives. Any change to an overwritable variable in any perspective triggers a so-called negotiation process during which people who created the affected perspectives are asked to resolve the potential conflicts.

1.2.5 Current Position

The original aim of AI was to produce general purpose, domain independent AI tools that would automate (non trivial) tasks requiring intelligence. The early promise of AI systems has not been fulfilled because these tools [Wallace *et al* 1995a]:

- did not *scale* within and across domains,
- could not *adapt* to new contexts,
- failed to handle *complexity* adequately,
- could not *acquire knowledge* satisfactorily, and
- placed too much emphasis on automation at the expense of assistance.

The reasons for this included:

- an artificial separation of problem specification from overall domain context,
- over reliance on deductive (and explicit) inference mechanisms,
- poor or inappropriate knowledge representations,
- knowledge bases that were difficult to maintain and extend,
- inability to adapt and to learn,
- poor interfaces, and
- a lack of commitment to design knowledge acquisition.

None of the design systems discussed in this section addressed the issue of developing a knowledge-based design support system *architecture* capable of supporting the exploration and maintenance of multiple design contexts and capable of utilising machine learning techniques for design tasks.

1.3 Supporting Design Using Machine Learning Techniques

Learning is a process of gaining new knowledge and it is at the centre of human intelligence. Machine learning is the key to machine intelligence just as human learning is the key to human intelligence. The ability to learn, to adapt, to modify behaviour is an inalienable component of human intelligence [Carbonell 1990]. Machine Learning is the area of AI that focuses on self-improvement processes and it is a characteristic of adaptive systems which are capable of improving their performances on problems from previous experience. Computer-based design support systems need to have this learning capability, which is an integral part of design activity.

Computational learning systems are divided into three main categories: *symbolic learning*, *connectionist (or neural network approach)* and *genetic algorithms*. Symbolic learning uses symbolic computation as a way of inducing new concepts from well-represented examples. The connectionist approach models computational learning using networks of small, neuron-like units. The genetic algorithms approach investigates computational learning, as well as other types of computation, using processes similar to genetic mutation and evolution. In all these approaches, learning operationally means the ability to perform new tasks that could not be performed before or perform old tasks better (faster, more accurately etc.) as a result of changes produced by the learning process [Carbonell 1990].

Within the learning systems that adopt a symbolic approach, there are mainly two different methodologies: inductive learning and explanation-based learning. Research on symbolic inductive learning has a relatively longer history than other forms of learning, such as genetic algorithms. This is because induction has a firm mathematical basis and has been used by scientists in different fields for a long time as a way of discovering new knowledge, with or without the use of a computer system. In particular, the most widely investigated method for symbolic learning is the induction of a general concept description from a sequences of known examples and counter examples of the concept. One of the methods in this approach is to develop a tree of concept descriptions with which all the previous examples, but none of the counter examples, can be re-derived.

Because this thesis focuses on the issue of integrating inductive learning techniques into a knowledge-based design support system architecture to support conceptual design, *connectionist (or neural network approach)* and *genetic algorithms* are not discussed although the use of genetic algorithms in embodiment design of mechanical systems is briefly mentioned in chapter 8. In this section, inductive learning, explanation-based learning and their relevance to design support are discussed briefly. A more detailed investigation of inductive learning techniques and its relation to knowledge-based design support will be presented in chapter 3 and 4. Case-Based Reasoning (CBR) is also discussed briefly in this section because it will be used in chapter 8 for a discussion on future directions for the software system developed in this thesis.

1.3.1 Inductive Learning

Reasoning and learning involve three factors: *fact*, *rule* and *conclusion*. The process of induction is to generalise rules from facts and conclusions [Mortimer 1988]. Inductive inference is one of the major learning methods used in science, engineering and our everyday life. As such, inductive learning is often characterised as a *data-intensive* approach to learning because it usually uses a large number of examples as a learning basis.

In a computational inductive learning system, the output of the system is a concept, a description of a theory, or a rule for classification. Such an output is generated by the learning system, using a set of well represented examples as input. In machine learning terms, this set of examples is called the *training set*, in the sense that a computer system is first trained to learn a concept by being given examples of that concept. After a training session the computer system will have accumulated enough knowledge or experience to deal with that concept. Typically, a learning system is expected to be able to classify the data relevant to that concept, to remember the features of that concept, or to reason about other data using that concept, etc. In the learning session an inductive learning system performs abstraction and produces generalities from specifics by processing various

training examples, while in the application session the same system applies the concepts that have been learned to classify new data [Winston 1975].

An inductive learning system uses the operations of generalising, transforming, correcting and refining the knowledge implicitly embodied within the training examples, until the knowledge for an explicit description of a concept is derived. In addition to accommodating the given facts of the concept, an inductive learning system may also need to introduce the background knowledge for cutting short an otherwise vast search space, and to satisfy the criteria for deciding the preferences among candidate hypotheses. In so doing, an inductive learning system is said to be *biased* towards an expected outcome.

The inductive bias of a learning system is often expressed in the form of *background knowledge* as preferences in the type of concept to be acquired. For example, *simplicity of the concept description* can be regarded as a common form of domain-independent inductive bias [Carbonell 1990 and Muggleton 1990].

The main application of inductive inference is to support classification and discovery in information processing systems or concept recognition systems. Classification is the typical inductive learning process of assigning, to a particular input, the name of the class to which it belongs. Discovery is more concerned with determining the features of the examples by characterising their similarities or differences. Both are seen as important components of intelligence. Inductive learning techniques are suitable for acquiring new concept descriptions from examples or for generating characteristic concept descriptions, which permits explanation to human users or manipulation by other system modules [Carbonell 1990].

Inductive learning systems can be divided into supervised and unsupervised learning, depending on whether or not the training examples are pre-classified. Some of the inductive learning systems use a bottom-up (or *agglomerative*) approach to develop concepts whilst others use top-down (or *divisive*) approach. Inductive learning systems also differ in the ways in which the examples are presented to the learning system (*incremental* or *non-incremental*). Chapter 3 will give a more detailed review of inductive learning techniques.

Inductive learning techniques have advanced in recent years to a stage where they can be usefully integrated into a knowledge-based design support system architecture. A number of systems that utilise inductive learning techniques in engineering design can be found in [Persidis *et al* 1989, Duffy *et al* 1993, Mostow *et al* 1989 and Reich *et al* 1991]. These systems will be reviewed in chapter 4 when the issue of integrating inductive learning techniques into a knowledge-based design support system architecture is discussed.

One of the challenges of developing a computational theory of the design process is to support learning by the use of computational mechanisms that allow for the

generation, accumulation and transformation of design knowledge learned from design experts, or design examples. One of the ways in which inductive inference methodologies can be integrated in a knowledge-based design support system architecture is to model the early stage of the design process as an incremental and inductive learning of design problem structures. The need for such a model arises from the need to capture, refine, and transfer design knowledge at different levels of abstraction so that it can be manipulated easily. In design, the knowledge generated from design solutions provides a feedback for modifying and enhancing the design knowledge base. Without a learning capability a design system is unable to reflect the designer's growing experience in the field and designer's ability to use knowledge abstracted from past design cases.

1.3.2 Explanation-Based Learning

Explanation-Based Learning (EBL) involves a *knowledge-intensive* reasoning process during which *domain knowledge* is utilised to *speed-up* the learning process [Mitchell *et al* 1986, Dejong *et al* 1986, Minton *et al* 1986 and Mostow 1989]. EBL systems adopt a more *analytical* approach to machine learning than inductive learning approaches. An EBL system can generalise a concept from a *single example* by explaining why that example is an instance of the concept. The explanation itself can then be used to describe the concept. This description is constituted by the explanation that identifies the *relevant* features of the example and the conditions under which these features are causally related to each other. Learning behaviour is demonstrated through the fact that the system's performance is improved over time as more and more explanations (cases) are accumulated. These cases can be used to 'cut short' the search space defined by the domain theory when a new but similar problem is encountered.

At the centre of the EBL learning technique, is a mechanism called Explanation-Based Generalisation (EBG) [Dejong *et al* 1986 and Minton 1990]. An EBG program accepts one *training example*, a *goal concept*, an *operationality criterion* and a *domain theory*. The training example represents some facts about the concept to be learned. The goal concept gives a high level description of what the system is supposed to learn given the available domain theory. The domain theory is, in general, a set of rules that determines relationships between objects and sub-concepts in the domain. The operationality criterion is used to judge whether a concept is acceptable or not. The learning process consists mainly of two steps: explanation and generalisation. In explanation, domain theory is used to prove that a training example is an instance of the target concept. This then forms an explanation in which the features of the training example that are irrelevant to the target concept have been pruned away, leaving only a special case of the concept. This explanation is acceptable if it satisfies the operationality criterion. An accepted explanation is then generalised further to make it useful for future

applications. In generalisation, the explanation is generalised as far as possible while still describing the target concept. This is achieved mostly by replacing constants in the explanation with variables.

One of the difficulties of EBL is dealing with the problem of imperfect domain theories [Mitchell *et al* 1986]. In EBL hypotheses are constrained to those derivable from background knowledge. However, background knowledge in applications is rarely complete. A domain theory may be incomplete, inconsistent or intractable [Mitchell *et al* 1986]. This constraint is, therefore, now generally believed to be over restrictive. If a domain theory is incomplete, the missing knowledge will result in a failure to explain the training example. One of the ways to address this problem is to combine similarity-based learning with EBL by using similarity-based techniques to generate a candidate set of possible generalisations from a large number of potentially noisy training examples. The EBL method is then used to prune and refine such empirical generalisations by using the domain knowledge available in the system.

1.3.3 Case-Based Reasoning

Case-Based Reasoning (CBR) is a psychological theory of human cognition and it is a general paradigm for reasoning from experience. It addresses issues in memory, learning, planning, and problem solving. CBR also provides a foundation for intelligent computer systems that can solve problems and adapt to new situations [Christopher *et al* 1989]. In CBR, whenever a new problem is encountered, a case-based reasoner is reminded of an old case which has been previously successfully applied to a similar problem, and whose result has been stored in a case library. The system then tries to adapt the previous solution to fit the current problem, taking into account any difference between the current and previous situations [Schank *et al* 1989]. This style of reasoning closely mimics human reasoning by emphasising the role of memory in general problem solving. A human expert often relies on cases of past success, rather than on reasoning just from first principles.

CBR is based on the assumption that new situations or experiences can remind a learner of previous cases and events. A new event (or an episode) can be classified in terms of past cases. One can therefore rely on past episodes to understand a new situation. Learning takes place when a new situation does not conform to prior cases. That is, if one's expectation based on a prior event does not occur in the new situation, one is forced to learn by classifying this new situation and remembering it. This learning process is sometimes called *failure-driven learning*.

The major significance of case-based reasoning is its emphasis on the role of memory in learning. Without memory it is impossible for knowledge to grow. Any case-based reasoning system must address the issues of how to obtain the original and correct cases,

how to store and organise the cases in a memory device so that they can be retrieved easily and efficiently, how to apply the cases to new problems, and how to learn from mistakes.

A CBR system works typically in the following steps:

1. the features of a new event are assigned as indexes characterising the event;
2. these indexes are used to retrieve a similar past case from the memory (the case base);
3. if the solution succeeds, then the indexes and the new solution are stored; otherwise
4. the failure is explained and the solution is repaired.

One of the early influential case-based reasoning systems is CHEF [Hammond 1989]. CHEF is a typical '*learning from mistakes*' system that develops new plans based on its own experience in the domain of cooking. When asked to prepare a dish for which it has no recipe, the system modifies an existing plan to fit the new situation, and then tries to detect and correct any errors that may result. CHEF tests an initial solution by simulating the process of making the dish. The results of simulation are examined to see if they match the requirements of the intended dish. If the program detects a failure, i.e., the requirements are not met, then it tries to analyse and explain the failure by asking questions. Finally the program modifies the recipe based on its knowledge and the user's answers to the questions.

1.3.4 Current Position

Machine Learning has become an important branch of AI. However, its potential in knowledge-based design support is yet to be fully explored. The reason for this is that there is limited understanding of the role of learning in design. Inductive learning is suitable for solving under-constrained problems such as *design*, *planning* and *diagnosis* [Carbonell 1989]. However existing inductive learning techniques are not readily usable for design applications. For example, clustering analysis methods are suitable for classifying categories of data for which there is no prior knowledge. These methods, however, impose a structure on the data set which may not necessarily be desirable for the purpose of designing something new. Therefore these techniques have limited use in design application².

² A more detailed discussion on the relation between inductive learning and knowledge-based design will be presented in chapter 4.

Concept formation is an approach that utilises domain specific knowledge to restrict the output of cluster analysis to only those that are desirable. It is therefore suitable for early stage design, the task of which is to derive the conceptual structure of a design problem from raw data or past design examples. However, few design systems have utilised such a technique as part of a knowledge-based design support system architecture for conceptual design support.

1.4 Research Overview

Knowledge-based design systems stress the explicit use of knowledge throughout the design process by the members of a design team. Although various knowledge-based design system architectures have been developed, the issue of exploring and maintaining design justifications and multiple design contexts within an integrated design environment using AI techniques has not been appropriately addressed and tested.

A primary source of difficulty in maintaining multiple contexts of design based on different perspectives defined by members of a design team in a computer-based design system is to keep track of the many constraints that necessarily exist between the function, physical characteristics, structure, cost and reliability of the various components. A mechanism is needed for keeping track of the reasons for and against the many choices made by the designers and for maintaining networks of constraints representing complex design problems. A computer-based system should support the designer to seek the optimum balance between conflicting demands of cost, performance, durability, size and weight and not overlooking safety regulations or other legal requirements.

The research into knowledge-based design support and inductive learning in this thesis is motivated by the need to develop an integrated architecture that can support designers using the best available AI techniques, and by the belief that the capability of a knowledge-based design support system can be enhanced by utilising inductive learning techniques.

The work described in this thesis consists of two separate but interrelated areas. The first is a knowledge-based design support system *architecture* in which a number of computational techniques such as an object-oriented knowledge representation, a blackboard control strategy and an assumption-based truth maintenance system are integrated. This architecture supports the development of knowledge-based design applications in general by providing facilities for the representation of design knowledge intelligent control of the design process and the exploration and management of multiple contexts of design. The second is the development of a design concept learning system that combines inductive inference and design knowledge to derive initial design concepts from design examples. This design concept learning system is integrated with the

architecture to support conceptual design tasks in the early stage of the design process when conceptual design solutions are highly valued.

This thesis addresses the computational techniques for design support by focusing on the development of a knowledge-based design support system architecture, and a design concept learning system. It involves both theoretical research, software development and real design application, the main aims of which are:

- To develop a knowledge-based design support system architecture capable of supporting design knowledge representation and acquisition, intelligent control, management of multiple design contexts, documentation and explanation of design results. This architecture integrates a blackboard control system and an ATMS within an object-oriented knowledge representation scheme for the exploration and management of multiple contexts of design. This integration forms the basis of a design context management system with the potential to support concurrency management in engineering design. This problem of maintaining multiple design contexts within a knowledge-based design support system architecture is not sufficiently addressed and tested by many existing design support system architectures to be discussed in detail in chapter 2.
- To investigate the role of inductive learning in design and to integrate inductive learning techniques, in the form of a design concept learning system, into a knowledge-based design system architecture to support the conceptual design tasks for a class of design problems where the structure of a new design is derived by inductively learning from the structures and features common to a number of existing design examples, which have similar features of varying degrees.
- To integrate the architecture and the design concept learning in such an integrated way that: (1) the design concept learning system is an integral part of the architecture that supports conceptual design by utilising design heuristics as background knowledge; and (2) that the architecture has an intelligent control and design documentation system which can replay and explain the documented design information as design history.

In order to address two main areas of research, i.e., architecture and learning in a unified manner, an inductive learning model for design is first developed to provide a basis for designing and implementing the architecture and the design concept learning system. This inductive learning model for design identifies the role of learning in exploratory design and integrates design heuristics with concept formation techniques in order to provide better support to a class of design tasks whose design problem structures

are initially not well known. In particular a number of concept learning approaches including an approach utilising design heuristics as background knowledge to build the design concept tree are investigated and tested. The tree generated by the design concept learning system is not a general tree for classification. It represents multiple design concepts that can be further explored within the integrated architecture.

In order to evaluate the methodologies and the software systems developed in this research, the problems in small-molecule drug design is used as a test case for both the architecture and the design concept learning system. The problems in the drug design application are addressed by embodying an incremental inductive learning system into the knowledge-based design support system architecture. Detailed studies of drug design concepts and knowledge representation of drug design objects are carried out in order to identify the design heuristics that can be used in the design concept learning system as background knowledge. This study contributes to a good understanding of complexity in the domain and thus helps to identify the appropriate learning strategy for this realistic and complicated application.

Both the architecture and the design concept learning system have been developed as general design support tools. Their applicability beyond the domain of small-molecule drug design is addressed and evaluated with respect to other approaches and systems in the field.

In summary, this thesis presents a knowledge-based design support system architecture based on an integration of a blackboard control system and an ATMS, and a design concept learning system that can provide support to design knowledge representation, design knowledge capture, intelligent design control and consistency maintenance in general, and provides a unique facility for exploring and maintaining multiple contexts of design and for concept learning in the early conceptual design phase of the design process in particular.

1.5 Organisation of the Thesis

This thesis is divided into eight further chapters. Chapter 2 discusses the techniques for intelligent design support including mainly knowledge representation, intelligent control, design context management and user modelling. This chapter also investigates the architecture of a knowledge-based design support system based on a review of the existing architectures for intelligent design support.

Chapter 3 reviews inductive techniques to provide a technical background for a study of the relationship between learning and design in chapter 4. This study compares and evaluates a number of design systems utilising inductive learning techniques. This leads to the description of an incremental learning model for design and a design concept

learning system that derives a design concept tree by combining concept formation techniques and design heuristic knowledge.

Chapter 5 focuses on the application of knowledge-based design techniques in the domain of small molecule drug design. This chapter elicits the problem of drug design and reveals the context in which this problem can be tackled by the combined use of inductive learning techniques and design heuristic knowledge within the architecture discussed in chapter 6.

Chapter 6 describes the architecture and the components of a knowledge-based design support system architecture. This architecture is based on an integration of a blackboard control system and an ATMS in an object-oriented environment. This chapter focuses on the integration aspect of developing this knowledge-based design support system architecture.

Chapter 7 describes the application of the design concept learning system in a key problem in drug design, i.e. the identification of a pharmacophore description. Examples are used to demonstrate how the learning approach works.

Chapter 8 evaluates the developed knowledge-based design support system architecture and design concept learning system. It also discusses the problems encountered in the research and possible further research topics based on the evaluation results. Chapter 9 concludes the thesis.

Appendix A lists some of the specific concepts and rules in the domain of small-molecule drug design. Appendix B describes the architecture software components and their integration. Appendix C describes an Expert System toolkit called GoldWorks II that has been used to implement the software systems described in this research.

Chapter 2

Knowledge-based Design Support System Architectures

A knowledge-based design support system architecture is a computational environment within which a number of computer programs (or sub-systems) operate in a co-ordinated manner to provide design support functions. In the domain of engineering design, a knowledge-based design support system can also be referred to as an intelligent CAD system. As summarised by Kimura in the group discussion on Architecture and Implementation in IFIP WG5.2 Workshop on Intelligent CAD, the basic functional components in an intelligent CAD system consist of *user interface*, *design objects*, *design manager*, and *design knowledge* [Kimura 1989]. These functional components and their relationships are illustrated in Figure 2.1.

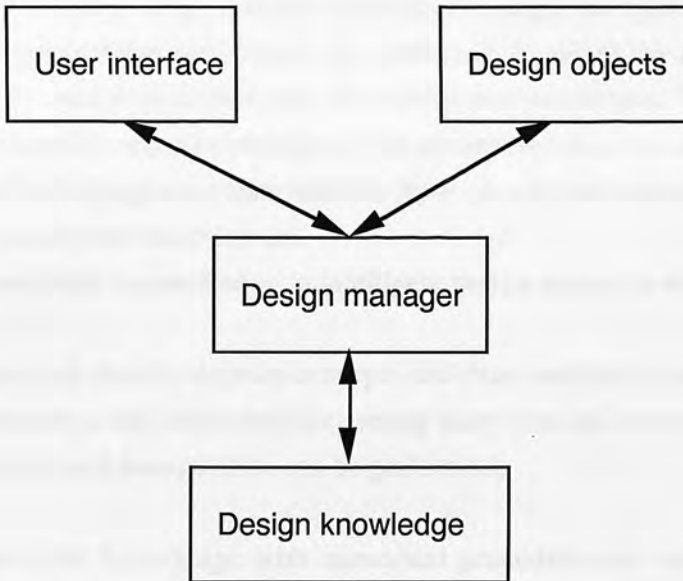


Figure 2.1: Basic Architecture for Intelligent CAD

In this basic architecture, design objects represent models of artefacts that can be combined and modified in a new design. Design knowledge represents a variety of data and knowledge for manipulating the design objects. The user interface presents a user-oriented dialogue scenario for various design activities. The design manager controls all the necessary management activities for exploiting these three functional components. Although this basic architecture is defined at an abstract level, it identifies the importance of a control system (the design manager) for interfacing three different design resources: human ability (user) design data (design objects), and design knowledge (design

methods). Knowledge about the design process and management, standards, company procedures etc. is also regarded as design knowledge [Kim 1990].

The development of system architectures for intelligent design support has advanced in recent years. Some of the design system architectures such as the Edinburgh Designer System (EDS) [Smithers *et al* 1990b], the Hierarchical Object-oriented Blackboard System (HOBS) [Carter *et al* 1991], the Intelligent-Front end (IFe) [Clarke *et al* 1991], the Distributed Integrated environment for Computer-aided Engineering (DICE) [Sriram *et al* 1989, and 1992], and the Integrated Design Framework (IDF) [Ball *et al* 1992] are reviewed in this chapter. This review provides a basis for discussing knowledge-based design support in terms of *design knowledge representation, intelligent control system, design context management, user modelling and integration.*

2.1 Knowledge Representation

Design of any kind is a knowledge intensive activity. In design, the ability of a designer to structure the design problem and identify its solutions is based on the observation and judgement gained by well organised design knowledge and experience. This knowledge includes not only common sense knowledge of the domain but also the experience at all levels of a hierarchical design environment, i.e., from system performance, sub-system functionality and component reliability etc.

The role of knowledge representation in intelligent design support is to:

- help to organise and classify domain concepts and data; establish at a high level the multiple relationships and dependencies among many design concepts with which effective reasoning and manipulation can be performed;
- incorporate heuristic knowledge with numerical procedures to cope with design complexity and the discontinuities of the design process;
- manage the creation, deletion, modification and recording of various design information; and
- represent previous designs as a source of knowledge.

2.1.1 Domain and Design Knowledge

A knowledge-based design support system should be able to represent and supply useful amounts of the *well understood and well structured knowledge*, that is, the kind of

knowledge found in text books, design handbooks, manufacturers catalogues, design notes, design guidelines, or design databases etc. [Smithers *et al* 1990b].

From a system management point of view it is useful to classify the knowledge necessary for intelligent design support into three major categories:

1. static knowledge representing design objects and concepts (domain knowledge);
2. heuristic and inferential knowledge representing problem solving strategies and methods including user decisions (design knowledge); and
3. knowledge generated during design when applying 2 to 1 (new knowledge).

Domain knowledge is relatively static and is the common concepts and objects in the domain of a design application. For example, in the domain of mechanical engineering design, this domain knowledge can be broken down into *components*, *parts*, and *assemblies* that can be selectively combined to represent the basic structure of an artefact. Such a structural representation is often referred to as a *product data model* [Yoshikawa *et al* 1989].

A product data model contains the relevant product description information such as *specification*, *function*, *attributes*, *behaviour*, *documentation*, *geometry*, *history*, and its *associated sizing methods* etc. This kind of knowledge gives shape to the basic structure of a design solution space, i.e. it identifies and locates interesting regions and important interfaces between design objects [Smithers *et al* 1991a]. In a knowledge-based design support system, the components for building the product data models should be made available to anyone involved in the design process as a member of a design team. In addition, a product data model needs to be associated with viewpoints or contexts in which this knowledge is to be accessed and explored by different types of user. The user may be a designer, a production engineer, a maintenance engineer, a manager etc.

Design knowledge is, on the other hand, about how to explore the solutions of design problems. It is knowledge about the design process and design problem solving. This kind of knowledge includes mainly design standards, prototypes or documented previous designs, heuristics and management strategies and generic knowledge including mathematics, chemistry and physics that can be used in the different stages of the design process.

In a knowledge-based design support system, domain knowledge is usually stored in a knowledge base as *design objects* and *relations* whilst design knowledge is represented as *inference engines* that manipulate the design objects. An important issue in design knowledge representation is how to capture and organise both domain and design

knowledge so that they can be effectively processed by computers in design problem solving.

2.1.2 Rule-based Representation

The functional, structural and causal relationships among design objects may not necessarily be described fully in quantitative terms. Quite often, at the early stages of the design process where detailed information about the artefact being designed has not yet been made available or established, the behaviour of the artefact may only be described in *qualitative* terms [Forbus 1984 and Kuipers 1986]. For example, a causal relationship between the structure and feature of an artefact may be used to answer the questions like "*what happens if something changes ?*" [de Kleer *et al* 1986 and Iwasaki *et al* 1986]. A functional relationship may describe something like "if a component is a shaft then it transfers a torque from input to output".

A production rule specifies actions to be taken in certain situations. A production rule takes the form of **IF A THEN B** in which **A** is a set of situations or preconditions which must be logically evaluated and found to be true if actions specified in **B** are to be carried out. An example of such a production rule is illustrated in Figure 2.2. This rule states that if *?drug* is an instance of a class named *drug* and with a unknown slot named *components*, then send a message to a method named *identify-components* which is attached to object class *drug* and store the returned results in the slot *components*.

```
(define-rule drug-design-rule-1 ()
  (instance ?drug is drug
    with components unknown)
  then
  (instance ?drug is drug
    with components
      (send-msg ?drug :identify-components)))
```

Figure 2.2: A Simple Production Rule

A production rule can be regarded as a *self-contained and independent knowledge source* capable of manipulating a set of object instances through pattern matching. Rules can be used to perform a design task in either a *forward chaining*, *backward chaining*, or *goal-directed forward chaining* style. Rules can also be grouped into *rule-sets* to model a *sequential* or *hierarchical* design process [Smithers *et al* 1992]. A *rule set* is a set of production rules grouped together to perform a specific design task. A rule set can be

activated or deactivated. If a rule set is deactivated, then the rules associated with it will not be triggered when a rule-based forward chaining engine or backward chaining engine is started.

Rule-based inference works well with frame-based data structures. However, it is unable to manipulate complex data structures and it is inefficient in dealing with complicated quantitative relations. There are few reported design systems that solely rely on a rule-based representation. More often it is necessary to combine rule-based representation with other representation schemes.

2.1.3 Object-Oriented Representation

Object-Oriented representation overcomes the difficulties of conventional computational models by bridging the gap between a piece of data and its operations. An object *encapsulates* both state and behaviour by having a set of procedures (or methods) that specify operations. Sets of similar objects are grouped together under *classes* or *frames*. This simplifies association of knowledge within objects by keeping the implementation details private within each class, thus allowing interactions between objects of different classes to be easily controlled and manipulated. The information about how an object behaves is hidden from the behaviours of other objects with only their interactions and relationships being described explicitly, thus allowing more abstract high level relationships to be established.

Three key object-oriented concepts are *abstract data types*, *inheritance* and *polymorphism*. Abstract data types and inheritance enhance code extendibility and reusability by encapsulating the state (data structure) of an object with its dynamic behaviour (reasoning methods). Polymorphism allows a reasoning method to take more than one form, and make polymorphic references that can refer, over time, to instances of more than one class [Tello 1989 and Wallace *et al* 1995b] A design situation can be modelled as an object hierarchy which represents the structure and the functional relationships of various parts and components. For example, Figure 2.3 illustrates a simple object hierarchy in the domain of mechanical engineering design.

Data manipulation in an object-oriented system is based on *message passing methods*, *automatic object updating functions*, and *embodied rules*. Message passing is the major inferencing mechanism in an object-oriented system. When one object sends a message containing usually an operation name and arguments to other objects, these objects respond to the message by searching for an operation of the same name in their class definitions. If this operation exists, then it is carried out on all the instances of the same class and its sub-classes. Otherwise the system searches in the object's parent classes for the operation that can be inherited. Automatic object updating functions can be attached to the slots of an object. When the values of these slots change, these functions will be

invoked to carry out the necessary operations. A set of rules can also be embodied with an object. When the object instance receives a message, this set of rules will be invoked to carry out the necessary inference.

The combination of *message passing*, *automatic object updating functions*, and *embodied rules* allows a complex design system to quickly respond to a message of change, updating the affected objects automatically until the change has been propagated throughout the object structure. In design applications, object-oriented representation presents a natural way to structure design objects and for building inferential design knowledge sources [Smithers *et al* 1991b, Wallace *et al* 1995b and Tang *et al* 1991a, 1991b and 1992].

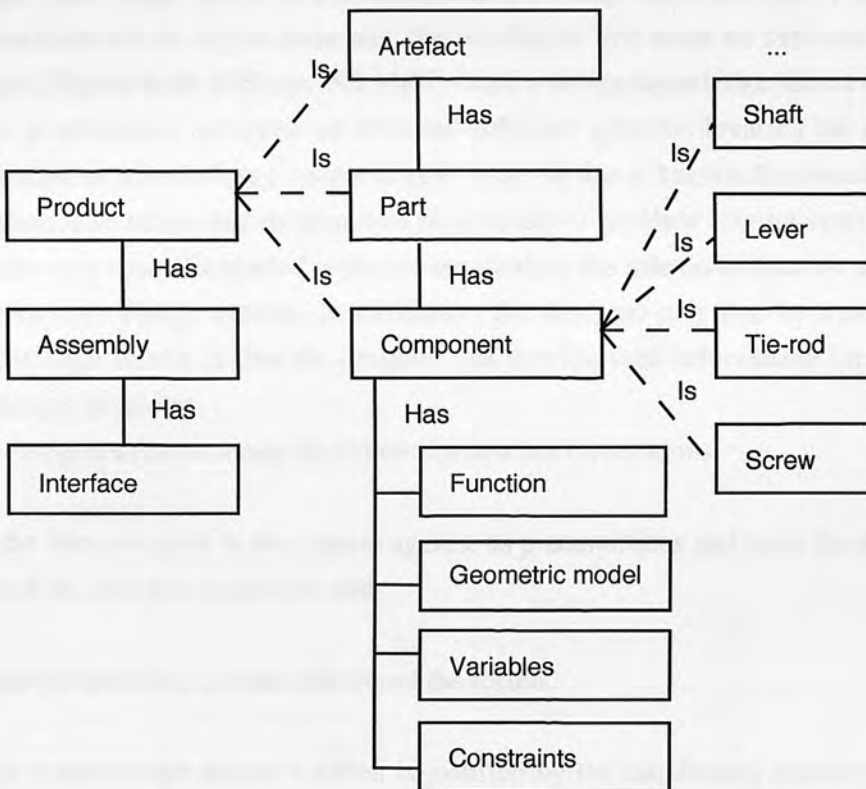


Figure 2.3: An Object-Oriented Product Data Model

In this object-oriented knowledge structure, an artefact is a kind of product which has an assembly. An artefact itself has parts each of which has components. A component has in general function, geometry, variables and constraints. Shaft, lever, tie-rod and screw etc. are all sub-classes of a component. These sub-classes inherit general attributes associated with a component and also have their own specific attributes. For example, the function of a shaft is to transform an input torque to an output torque. The shaft has variables such as speed s , diameter d , length l , bearing force Fb , torque T , power P and

mass m . Well defined constraints such as the relation $P = sT$ and $m = \rho d^2 \pi l / 4$ can be associated with the shaft in advance.

2.1.4 Design Knowledge Source

In a computer-based design system, design knowledge needs to be organised to perform various design tasks. A *design knowledge source* is an encapsulation of objects, methods and rules representing an autonomous piece of design expertise for a specific design task such as kinematic analysis in mechanical engineering design, or a general design task such as satisfying a set of design constraints.

A design knowledge source is self-contained and independent because it defines a reasoning method for an object class and the conditions that must be satisfied for the method to act [Hayes-Roth 1985 and Nii 1986]. Such a design knowledge source can be a single rule, a procedure, or even an external software system. Treating an external software system as a knowledge source makes sense in that a knowledge-based design support system is an integrated environment of a variety of problem solving systems. The notion of user as a special knowledge source emphasises the role co-ordination aspect of a human/computer design system. In particular, the designer can also be treated as a special knowledge source in that the designer can provide vital information for solving particular design problems.

A knowledge source performs the following two basic operations:

1. match the data available in the system against its preconditions and carry the bindings forward if the match is a success; and
2. carry out the specified actions and record the results.

Because a knowledge source's action is justified by the satisfactory matching of its preconditions against the available data, in a computer system these preconditions can be recorded together with the results generated so that when a design is concluded it is possible to trace the original decisions that have led to the results. A design knowledge source plays an important role in determining the justification or the context in which a piece of new knowledge is generated. This issue is to be discussed in more details in chapter 6 when describing the architecture of a knowledge-based design system.

2.1.5 Design Knowledge Base

A *design knowledge base* provides design knowledge in a *behavioural*, *functional* and *structural* vocabulary in a computer-based design system. In such a knowledge base, a

structural relationship between design objects determines how objects are geometrically or physically related to each other and how, for example, the characteristics of a power train can be derived from its constituent components such as piston, crank pin, crank web, crank shaft, bearing etc.; a functional relationship between design objects relates objects in terms of their performances or behaviour such as input/output transmission, or their relevance to a particular design requirement and design task; a causal relationship between two design objects decides how they depend on each other and what the consequences are of a change in either of them.

This thesis adopts an object-oriented design knowledge base structure as illustrated in Figure 2.4.

Design objects including physical design components and high level design concepts and their dependent functional, causal and structural relationships are classified and structured within this object-oriented design knowledge base structure. Ways of reasoning, in terms of procedural and heuristic design knowledge, are represented as message-passing methods, automatic object updating demons, external reasoning modules, heuristic rules or rule sets, and domain dependent algorithms performing specific design tasks. The design knowledge base can be linked to databases representing static domain knowledge about the structural, property and cost information of various design objects and concepts. Design results are documented as text-based files and design documents.

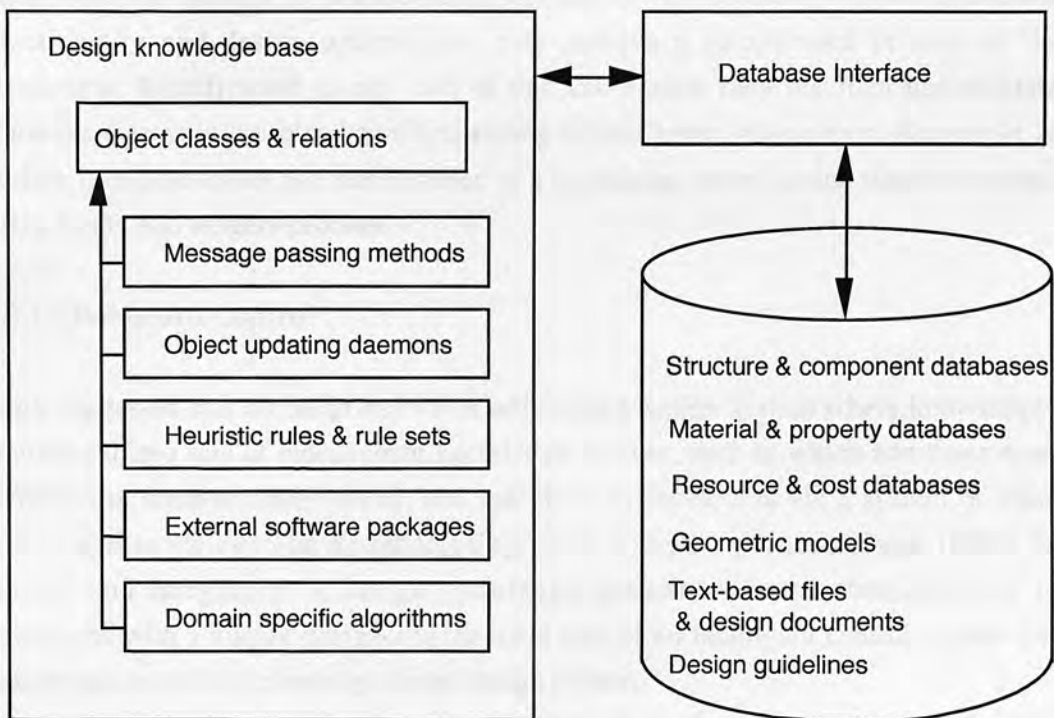


Figure 2.4: The Basic Structure of a Design Knowledge Base

2.2 Intelligent Control System

Advanced design systems are often developed in an environment where

- groups of designers work co-operatively to complete a complex and frequently large design,
- close interaction exists between group members through shared dynamic design data and information,
- interdependent knowledge sources exist and need to be controlled and co-ordinated,
- use of data bases, modelling systems and management tools is commonplace, and
- control of multiple contexts is necessary for a group of designers.

Analysing, synthesising, and evaluating designs in a computer-based design system is difficult because a vast amount of knowledge and information from diverse sources need to be processed. Such knowledge and information are not necessarily readily available in a unified format. Further complications exist when various design tools need to be integrated. Any attempt to balance design options on various sub-systems to obtain total functionality and design optimisation may involve a complicated process of data processing. Modification to any part of one sub-system may result in unpredictable consequences and unresolved conflicts among various other sub-systems. As a result, the design, implementation and maintenance of a knowledge-based design support system is still a costly and lengthy process.

2.2.1 Blackboard Control

Pugh suggested that an integrated knowledge-based design system where knowledge is aggregated into sets of independent knowledge sources, each of which addresses a sub-problem, is fundamentally sound, and that the way forward is via a system of linked *knowledge modules* to be designed using multi-disciplinary teams [Pugh 1989]. The control and integration of design knowledge sources to ensure their effective co-operations with a human designer is the main task of an intelligent control system (or a design manager) in a knowledge-based design system.

A blackboard is a flexible control system that allows the system's knowledge sources to interact with user input via the blackboard [Hayes-Roth 1985]. In such a control

system, the working memory of the system is viewed as a blackboard where communication between different knowledge sources takes place. This blackboard is the dynamic knowledge base of the system and it provides globally available data structures to enable all the design knowledge sources to operate.

The blackboard model represents a highly structured automatic control scheme, and a special case of opportunistic problem solving. It has a simple architecture consisting of a number of *knowledge sources*, a blackboard *data structure* and a control system, and it controls knowledge sources on an opportunistic basis.

In a blackboard system, knowledge sources as self-contained intelligent agents are controlled in a systematic way, as illustrated in Figure 2.5. The system invokes each knowledge source in turn in each control cycle. Those knowledge sources whose preconditions are matched by the information already held on the blackboard propose a bid for work in the form of a Knowledge Source Activation Record (KSAR). KSARs are managed by the control system in a blackboard agenda. The system executes KSARs in the agenda one by one until it becomes empty.

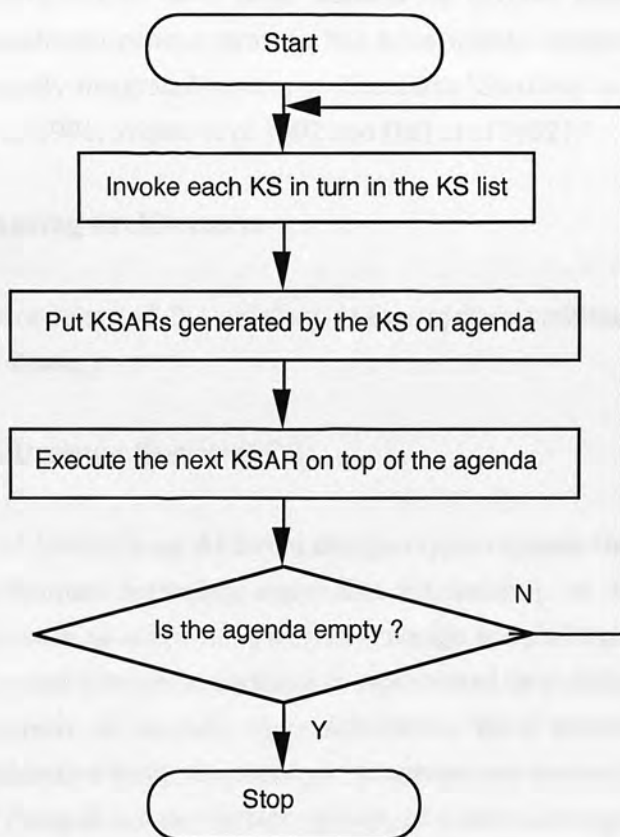


Figure 2.5: Blackboard Control Strategy

A blackboard control strategy contributes to design support tasks in the following aspects:

- it allows a design support system to work as a self-organising system whose problem solving knowledge sources can respond dynamically to design situations;
- it encourages the development of independent and self-contained design knowledge sources, thus making it easy to integrate and modify design knowledge; and
- it has a control mechanism that is suitable for different knowledge representation schemes, i.e., anything that can be treated as a black box such as a rule set, an internal message passing handler, or an external software package can be regarded as a knowledge source.

Given the integrated nature of knowledge based design systems, the issue of integration and co-operation of a large number of diverse knowledge sources is important. The blackboard control strategy has been widely adopted in design system architectures, especially integrated system architectures [Smithers *et al* 1990b, Carter *et al* 1991, Clarke *et al* 1991, Sriram *et al* 1992 and Ball *et al* 1992].

2.2.2 Review of Existing Architectures

This section reviews some of the existing design system architectures that utilise a blackboard control strategy.

2.2.2.1 Edinburgh Designer System (EDS)

EDS [Smithers *et al* 1990b] is an AI-based design support system that uses a number of computational techniques including constraint satisfaction, to support mechanical engineering design, such as *water turbine system* design and *fuel injection pump* design. In the EDS, design and domain knowledge is represented in a *design knowledge base* (DKB) as a taxonomy of *module class definitions*. Each module class definition represents, in a declarative form, the concepts, structures and constraints of a *component* (for example, a jet pump in a water turbine system, or a heat exchanger in a fuel injection system). These module class definitions are connected by a single inheritance scheme. For example, one module class definition can be related to the others through '*part-of*' or *has-parts* relations. Design is carried out through a *design exploration process*, where a designer explores the possible ways of constructing a new design using the available

module class definitions, or tests different values of design variables within existing module class definitions.

In the EDS, a variant of the blackboard system was used to control the interactions between different domain independent inference engines [Smithers *et al* 1990b], [Millington *et al* 1988 and Jones *et al* 1986]. The EDS blackboard control system carries out inference whenever the designer enters something into the system. It is the responsibility of the EDS control system to interact with the design knowledge sources so that something can be derived using the available module class definitions stored in the design knowledge base. The EDS inference engines can infer on mathematical equations, shape equations, spatial relationships and functional and catalogue relations. These engines form the core of a constraint propagation system which is controlled using a blackboard control system.

The EDS supports the exploration of multiple design solutions using an Assumption-based Truth Maintenance System (ATMS). This allows all the knowledge (consistent or inconsistent) relevant to a design exploration attempt to be maintained. The EDS maintains the knowledge generated during a design exploration session using a knowledge structure called the Design Description Document (DDD). The DDD contains: a complete and consistent description of the final design requirement, a final design specification for all the design variables and design parameters that match the final design requirement description, and a history record of the way in which the design space has been explored.

However, the module class definition syntax in the EDS has limited facilities for representing complex design objects and design tasks. An example of this difficulty was presented and discussed in [Smithers *et al* 1990a]. The issues of intelligent control and the maintenance of the consistency of the design knowledge were addressed in the EDS [Smithers *et al* 1990a and Logan *et al* 1991]. But the facility developed in the EDS was developed as a domain specific application in mechanical engineering design. It therefore cannot be used as a general reusable design support system architecture. The EDS II project was concerned with improving the control of inferences of the EDS system. The EDS II system improved the EDS by developing a view system that can create multiple contexts [Logan *et al* 1991]. However, again the EDS II system was not developed as a general design support system architecture and its facilities are not sufficiently tested in more than one domain. Both EDS systems did not address the issue of learning in design.

2.2.2.2 HOBS

Carter and MacCallum developed a Hierarchical Object-oriented Blackboard System (HOBS) for supporting Electromagnetic design [Carter *et al* 1991]. In the HOBS architecture the segments of design expertise or resources are organised into a hierarchy

of *knowledge sources* which are controlled through a blackboard control system called the *executive*, and communicate with one another about the *product* through a common data area called the *workspace*.

The knowledge sources contain design procedures. The workspace provides a global data area for the development of design solutions. The executive provides a general operating cycle during a design session. In addition to these components are a *user interface* providing the user's view and methods of access to the system, and some *external facilities* for connection to other systems [Carter *et al* 1991].

Each knowledge source in HOBS comprises a *body* and a *shell*. The body contains the actual design expertise, such as how to design a component, or how to analyse the performance of a product. The shell is used to link the body to the system by defining the *contents* of the knowledge source and *when* and *how* it should be used. Knowledge sources are organised to form a hierarchical structure. The shell of a knowledge source defines its *parent* (higher level knowledge sources) and *children* (lower level knowledge sources) in this hierarchical structure can be invoked by the body of the KS as and when required.

The workspace in HOBS is divided into the *database area*, the *toolbase area* and the *message board*. The database area stores the requirements and data models for the design problem and is used as a communication base between the knowledge sources. The data models are design objects representing physical components and abstract concepts which are structured thorough *part-of*, *has-features*, *connected-to* relations. The toolbase is a set of generic representation and manipulation facilities, which can be used to manipulate specific aspects and concepts of design problems. The message board allows messages to be passed between various components of the system.

The executive in HOBS is a blackboard control system which decides *what* action to perform *next*. The executive contains a primary operating cycle, which *solicits*, *reserves* and *sorts* bids for work from the knowledge sources, and *selects* the next one to operate. The control system works effectively in a *depth-first* manner down the hierarchy structure of knowledge sources. Once a knowledge source has finished its operations stored in its body the control mechanism looks in its shell for its child knowledge sources and solicits bids from them. Once all the child knowledge sources have finished, the control system moves back up the structure to find other suitable knowledge sources.

Hierarchically structured knowledge sources enable the decomposition of a design problem and a task by the knowledge engineer. This hierarchical structure allows normal design management practice such as company procedure to be reflected in the control systems and allows the resources of the system to be focused on a small number of knowledge sources at one time. However, a drawback of this is that a hierarchical structure is imposed on the knowledge sources rather than on the objects. As a consequence, the knowledge sources become less independent and less opportunistic

compared with other blackboard control systems in EDS, DICE, IDF etc. [Sriram *et al* 1992, Smithers *et al* 1990b and Ball *et al* 1992]. HOBS does not maintain justifications of the design results. It is therefore unable to deal with multiple context problem solving. It does not have a learning facility.

2.2.2.3 DICE

Design and manufacturing are often accomplished by a team of engineers, each with specialist knowledge in a particular aspect of the problem, but with little knowledge of the concerns of others involved in the decision process. The Distributed Integrated environment for Computer-aided Engineering (DICE) is a design system architecture developed by Sriram [Sriram *et al* 1992] that provides co-operation and co-ordination among multiple designers working in separate engineering disciplines in the domain of construction and building design.

The architecture of DICE consists of a *blackboard*, a *control system* and a number of *Knowledge Modules* (KM). The blackboard is divided into three partitions: a *solution blackboard*, a *negotiation blackboard*, and a *co-ordination blackboard*. The solution blackboard contains the design and construction information in the form of object hierarchy generated by various KMs. The negotiation blackboard consists of the negotiation trace between various engineers taking part in the design and construction process. The co-ordination blackboard contains the information needed for the co-ordination of various KMs.

A KM in DICE could be one or a number of knowledge sources. KMs are classified as *strategy*, *specialist*, *critic*, and *quantitative* KMs. The strategy KMs comprises the design task control knowledge which is used in the co-ordination and communication process. The specialist KMs perform individual design tasks. The critic KMs contains the knowledge and mechanism for checking the consistency of the design data and constraints. The quantitative KMs are mainly algorithmic packages used for specialised design tasks.

The control system performs two tasks:

- evaluates and propagates implications of actions taken by particular KMs; and
- controls the negotiation process.

The first task is achieved by methods associated with objects in the object-hierarchy of the solution blackboard. These methods are used for performing procedural calculations, propagating implications of actions taken by KMs, and helping with the co-ordination process. The negotiation process takes place once a conflict is detected by the

system, and it is achieved through a strategy KM. Conflicts occur either due to the interface constraint violation or due to contradictory modifications of an object. Another type of constraint violation occurs when a KM changes a particular solution posted by another KM. In the negotiation process, constraint relaxation is first attempted for those constraints in conflicts. A goal negotiation is then tried when the first attempt fails. The goal negotiation involves the redefinition of a design goal by the design team members concerned in the conflicts.

The objects in the solution blackboard contain justifications, assumptions, time of creation, creator, constraints, ownership KM, other concerned KMs etc. The justification information provides a designer's rationale and intent for the creation of the object. Assumptions made during design and construction are also stored with the object. In DICE, status facets are associated with data attributes (slots). These status facets, for example, can take the following values (unknown, assumed, and calculated).

The KMs in DICE are more independent than those of HOBS in that the design problem decomposition is reflected in the object-oriented data structured on the solution blackboard. The partition of blackboard into solution blackboard, negotiation blackboard and co-ordination blackboard in DICE is a useful idea for supporting concurrent engineering design. However, the negotiation in DICE largely relies on constraint relaxation techniques and it therefore lacks a formal representation and mechanism for negotiation in concurrent engineering design. The issue of exploring and maintaining multiple contexts is not addressed in DICE although the necessity for a truth maintenance system is mentioned in [Sriram *et al* 1992]. In the DICE architecture, the learning issue is not addressed.

2.2.2.4 IFe

The Intelligent Front-End is an approach that addresses the issues of intelligent human/computer interactions in integrated knowledge based systems [Clarke *et al* 1991]. An IFe system distinguishes the *front-end* (user end) and the *back-end* (system end), and emphasise the importance of user modelling and effective interaction between the two ends.

IFe is a typical Intelligent Front-End system developed in Strathclyde by Clarke for computer-aided building design [Clarke *et al* 1991]. The IFe system is an integration of several intelligent *clients* within a blackboard system. The IFe consists of a *blackboard* for collecting, organising and storing data, a *dialogue handler* for conversing with the user in the appropriate terminology, a *knowledge handler* for generating the description of a building, a *data handler* for generating the program specific input data, an *appraisal handler* for generating the program-specific control inputs, and an *application handler* for invoking the targeted application programs.

The blackboard in the IFe provides two main functions:

- storing the data representing the problem definition as input from a user or inferred by the knowledge handler; and
- acting as a communication centre for its various clients.

A simple data structure called *Tuple* is used in the IFe blackboard. Each Tuple has a concept name, a string indicating whether it is "user-set" (defined by the user) or "kb-set" (inferred by the system), a list of auxiliary keys, and a value associated with the concept. Tuples can be stored in named areas on the blackboard to simplify the task of locating relevant or specific information. Auxiliary keys can be used by the handlers in the IFe to create arbitrary complex data structures linking multiple concepts.

Handlers in IFe are classified into two kinds: user-end handlers and back-end handlers. User-end handlers including the dialogue handler, the knowledge handler and the user handler are mainly concerned with extracting from the user the building and appraisal definitions whilst the back-end handlers including the appraisal handler, the data handler and the application handler are responsible for creating the input data and control instructions to invoke the application programs.

All the handlers are treated in IFe as *autonomous processes that run synchronously*. Handlers can ask the blackboard to create new, named areas for posting Tuples. The name of the poster and time of posting are recorded with each Tuple. Handlers can either explicitly ask the blackboard for information by identifying the area and the Tuple concept, or can ask the blackboard to keep them informed of any new information posted to any particular area. This provides a mechanism for one or more handlers to create a blackboard area to serve as a communication channel between them.

The IFe system simply passes on messages to handlers in the order in which the requests for the messages arrived. A problem with this scheme is that handlers wishing to exchange structured Tuples must all know the details of the imposed data structure. Further, the IFe system does not support learning. There is no explicit control circle in the IFe blackboard system. The blackboard is merely used as a data storage and communication base whilst in many other blackboard schemes this is a case of collecting and scheduling the responses of the knowledge sources to the current situation, and on some occasions the application of a strategy [Hayes-Roth 1985 and Nii 1986].

2.2.2.5 IDF

The Integrated Design Framework (IDF) is a system developed by Ball [Ball *et al* 1992] to support mechanical engineering design. The IDF architecture has two blackboards: a *control blackboard* and a *domain blackboard*.

The control blackboard contains design problem solving data at a high level of abstraction. The designer can access this blackboard through a *graphical user interface* which provides a friendly, transparent interface between the designer and all *knowledge sources* operating within the IDF system. The control blackboard represents what actions are desirable, feasible and actually performed at each point in the design process. The domain blackboard represents the design objects generated during a design session. These objects can be grouped into hierarchical assemblies.

Knowledge sources in IDF are classified by the blackboard on which they operate and by the type of knowledge they embody. Three types of knowledge sources are:

- System knowledge sources which are algorithmic and domain independent programs that generate the basic IDF system cycle of (1) scheduling and executing control and domain knowledge sources; (2) maintaining the product data model on the domain blackboard, and (3) logging all the blackboard events;
- Control knowledge sources are heuristics that model the design process according to the current design problem and designer profile, i.e., the definition of the design problem. This heuristic knowledge includes sequential plan for design problem solution; ranking of design objects/activities; and resolution of conflicting rankings.
- Domain knowledge sources are separate, stand-alone systems that assist the designer in the development of specific design objects from functional modelling through to detailed layouts. Databases owned by domain knowledge sources are accessible by other knowledge sources either directly or via a data model translation program.

The IDF architecture separates general design process control and domain specific design problem solving by having two separate blackboards. IDF has a scheduling knowledge source to deal with the control and maintenance of various bids from different knowledge sources. This knowledge source performs KSAR (Knowledge Source Activation Record) enumeration, evaluation and invocation.

The IDF has the advantage of modelling the design product as well as the design process within the environment by having a *design strategy knowledge source*, a *conflict policy knowledge source* and a *design focus knowledge source*. However, the IDF does not support the exploration and maintenance of design contexts or design alternatives.

The domain knowledge sources in IDF are domain specific problem solving packages which do not necessarily operate on a unified product data model. It is therefore difficult to utilise these packages for general design problem solving.

The IDF has a process logger knowledge source that records the events on the control blackboard. A prototype induction program is identified within the IDF that can transfer these events to product data models or design planning strategies that can be reused. However, this part of the IDF is yet to be fully developed and tested [Wallace *et al* 1995a and 1995b].

2.3 Design Context Management

In design, multiple design solutions or alternative design solutions arise in the presence of under-constrained design variables and parameters. The exploration of these multiple design solutions is context dependent, i.e., design solutions are derived from a space in which *design requirements*, *design methods* and *design criteria* are subject to frequent change. The management of design context in a knowledge-based design support system is concerned with the following issues:

- maintaining the consistency of design knowledge in the dynamic knowledge base;
- representing newly derived design objects, their dependants and antecedents; and
- exploring multiple design solutions based on different design decisions.

While some of the above discussed blackboard-based design system architectures such as DICE, EDS, HOBS have the notion of maintaining the consistency of design knowledge [Sriram *et al* 1992, Smithers *et al* 1990b, Logan *et al* 1991 and Carter *et al* 1991], the above issues are yet to be further addressed. It is necessary to develop and test a general mechanism for defining, exploring and maintaining multiple design contexts within an integrated design system architecture that can support current engineering design.

This section discusses computer-based exploration and maintenance of multiple contexts of design within a knowledge-based design support system architecture.

2.3.1 Modelling Context in Design

One of the frequently carried out tasks in design is to make plausible modifications to part of an initial design solution to observe the repercussions in order to find alternative design solutions. The design decision making process is *nonmonotonic*, *constructive* and

incremental in that even experienced designers cannot guarantee to get it right first time. More often, designers need to make changes when information about the consequence of earlier decisions becomes available, and the design solution space has become more and more constrained. Most design tasks are likely to create new design objects and concepts rather than simply modifying existing designs.

Traditionally, multiple design solutions are explored by human designers using an approach known as the Analysis of Interconnected Decision Area (AIDA) [Jones 1992]. This approach employs a generate-and-test strategy to identify alternative design solutions. It works in five steps:

1. Identify the decision areas in a design problem structure by focusing on some interesting design variables and dependent design parameters.
2. Identify several feasible options in each decision area within its isolated context.
3. Indicate which options are incompatible with others.
4. List those that can be combined together to form compatible candidate designs.
5. Evaluate the candidate designs using some quantitative measures in order to choose the acceptable ones.

Two problems arise when using this approach in a computer-based design system: first, one design problem is not easily separated from another and the decision areas are not clear to designers until considerable effort has been made to explore them; second, a generate-and-test strategy is too expensive for complex design problems involving a large number of design variables. This view is supported by noting that:

- design activities are often undertaken simultaneously by a group of designers;
- designers explore the design solution space rather than systematically search in it;
- it is necessary for incompatibility in design requirements, constraints and evaluation criteria to be discovered early during the design process;
- designers need to know the consequences and implications of any change either in input data, design method or design evaluation criteria; and

- designers need explicit explanations of the design results derived from any design decisions, or the reasons for any difficulties in reaching a satisfactory design solution.

These issues can be dealt with in a knowledge-based design support system using a consistency maintenance and context management system, the central role of which is to maintain the consistency of the knowledge generated during design and to support the exploration of this knowledge when design context changes.

GALILEO2 [Bowen *et al* 1992] is a constraint programming language that supports multiple context problem solving in life-cycle engineering using the concept of '*multiple perspectives*'. In GALILEO2, design variables and constraints can be viewed from different perspectives by the people involved in design, manufacturing and maintenance etc. A perspective is a small set of design variables and constraints. Perspectives are typically overlapping in that the same variable may appear in more than one perspective. The variables within each perspective can be declared as being *local* or *overwritable*. The local variables within a perspective can only be modified by the person who created the perspective, whilst overwritable variables can be changed from within any other perspectives. Any change to an overwritable variable in any perspective triggers a so-called negotiation process, during which people who created the affected perspectives are asked to resolve the potential conflicts. Here a perspective can be regarded as a context that defines a region in the design solution space with the user who created the region being part of the context.

Nonmonotonic reasoning is supported in GALILEO2 by the use of overwritable variables in multiple perspectives and by the process of *negotiation*. That is, when an overwritable variable is changed, all the users who have their own perspectives containing the changed variable are alerted and asked to approve the change. The negotiation process starts when any of the concerned users disagrees with the change. However, in GALILEO2 the original context is deleted after some overwritable variables are overwritten and the new values are propagated throughout the constraint network. In other words, the information associated with an earlier context is not maintained, even though it could still have potential value as an alternative design solution. The negotiation process assumes that there is no interdependency between perspectives. As soon as someone changes an overwritable variable and someone else disagrees, the matter must be resolved immediately. For true concurrency, it must be accepted that two or more activities can occur simultaneously [Medland 1995 and Tang 1995d].

In a knowledge-based design support system, a design context can be defined based on the following three different sources of knowledge [Tang 1995c and 1995d]:

- knowledge loaded from a design knowledge base in the form of instantiated design objects, variables, and constraints;

- knowledge provided by the designer in the form of design decisions (value assignments or design method selections etc.); and
- knowledge inferred by the system as a result of propagating the values of design parameters or variables throughout the whole set of data and constraints already held in the system.

When any of these knowledge types are changed or modified, the design context is changed and an alternative design solution may be derived based on the new context. In a constraint-based approach to design, the product data model (or structure of the design problem) is described by design objects and their relations. Design objects contain attributes which can be classified as *design variables* and a set of *dependent design parameters*. The values of design variables and dependent design parameters are determined by the constraints in which these variables and parameters are functionally, structurally or causally related to each other. Design variables and dependent design parameters are used in order to distinguish the part of a design problem that is flexible to change (described by the design variables) from other parts of the design problem (described by the dependent design parameters) that are relatively dependent on design variables.

A design solution is a complete set of values for all the design variables and dependent design parameters which jointly describe the features of the design problem and satisfy the constraints. The space of design solutions can be determined by the identification of the relationships between design variables and dependent design parameters. The constraints are combinations of mathematical equations, rules, and reasoning modules (such as a program determining input/output relations). In exploring the design space described by such a network of constraints, variables and dependent parameters, many plausible choices arise in the presence of under-constrained design variables, giving rise in turn to many plausible values of dependent design parameters. Furthermore the way in which design variables and dependent design parameters are related and explored may depend on what design strategies (or methods) are employed. In other words, using a different design problem solving strategy (or method) may result in the same set of design variables and dependent design parameters being differently constrained.

Design exploration involves exploring the convergence of all the design variables using the most appropriate design methods available. The exploration of a design solution is carried out by the designer making decisions in terms of specifying a product model containing design variables and dependent design parameters, assuming the values of design variables, or selecting design methods. The design system processes these

decisions in a systematic way by propagating the values throughout the constraint network.

A design context is a design solution (or a partial design solution) that is derived as a result of the system's inferencing from the designer's choice of initial data, design method or design procedure and design evaluation criteria. Because a designer's assumption is part of a design context, different designers working on sets of overlapping variables can explore the design solution within their contexts simultaneously. The co-existence of potentially conflicting assumptions and their derivatives presents a problem for maintaining the truth of the knowledge in a computer-based design system.

2.3.2 Reasoning about Context

Designing and manufacturing involve a team of engineers, each with specialist knowledge and concerns in a particular aspect of the design process, but with insufficient information about the concerns of others involved in the same process. A knowledge-based design approach assumes that designers still play a central role in the design exploration process. An intelligent support system architecture for concurrent engineering design therefore needs to deal with the problem of reasoning about context and coordinating the decisions made by members of a design team.

A number of truth maintenance or reasoning maintenance techniques have been developed in AI to deal with the problem of inconsistency and context management. While some of the truth maintenance systems are concerned only with maintaining the truth (the justification of derived knowledge) [de Kleer 1984a, 1984b and Doyle 1979], the Assumption-based Truth Maintenance System (ATMS), based upon de Kleer's work, offers sufficient facilities for working with inconsistent information, and for multiple context problem solving. This section explains the ATMS mechanism in order to provide a basis for discussing the issue of design context management in section 2.4.3.

As illustrated in Figure 2.6, an ATMS is composed of two units: the problem solver and the truth maintenance system. ATMS deals with two kinds of information: assumption nodes and derived nodes. Assumptions are taken as the primitives of the inference while derived nodes depend on other nodes, including assumption nodes. Each ATMS node contains a *datum*, a number of justifications, an *environment*, and a *label*.

An ATMS datum is a relation of a node corresponding to the problem solver's data. A justification defines a *dependency relation* between a set of *antecedent nodes* and a *consequent* node. A justification takes the form

$$A1 \wedge A2 \wedge \dots \wedge An \rightarrow C$$

where A_1, \dots, A_n are the antecedent nodes and C is the consequent node. The antecedent nodes can be assumption nodes, derived nodes or a combination of both. Since an assumption node is treated as a primitive in the ATMS, tracing back through the justifications of a consequent node ultimately results in a set of assumptions. Such a set of assumptions is referred to as an *environment*. A node is said to hold in an environment if it can be derived from the set of assumptions and their completions in the environment.

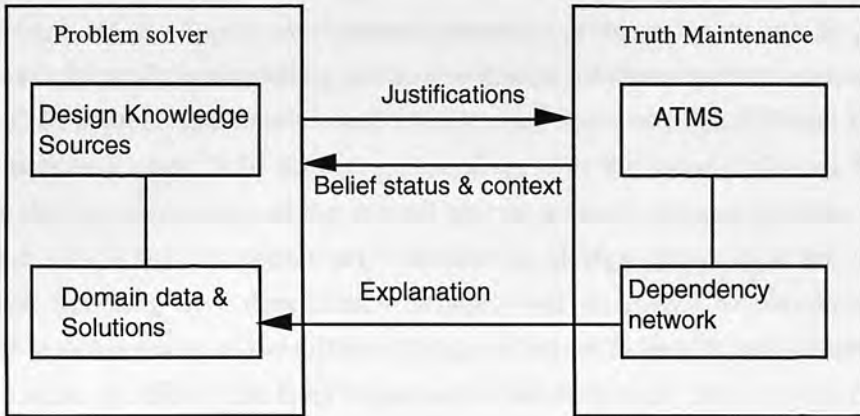


Figure 2.6: An ATMS

An ATMS manages the case when an inconsistency is derived or explicitly assumed by the user. A false node is a special node denoting inconsistency. An environment containing any false node is called a no-good environment.

An ATMS context is the set formed by the assumptions of a consistent environment combined with all nodes derivable from those assumptions. An ATMS label is a set of environments associated with an ATMS node. While a justification describes how the datum is derived from immediately preceding antecedents, a label describes how the datum ultimately depends on assumptions.

In an ATMS, the continuous inspection and the subsequent dispatch of the valid justifications to the ATMS is the responsibility of the problem solver whilst the ATMS guarantees that the conclusions reached by the problem solver are always kept updated and consistent in its database. That is, the ATMS guarantees that each label of each node is consistent, sound, complete and minimal with respect to the justifications. A label is consistent if all its lists of assumptions are consistent [de Kleer 1984].

Because a design context ultimately depends on the assumptions made by designers, different designers working on sets of overlapping design variables can explore the solutions within their own contexts *simultaneously*. Potential conflicts on the choices of design variable values are separated by contexts.

A design solution in a particular context does not have to satisfy the constraints imposed by the others working in other contexts. Negotiation or conflict resolution can be delayed until such a time as a number of alternative design solutions has already emerged from those contexts. However, a final design solution must be the outcome of a final evaluation or negotiation process during which design solutions from different contexts are *evaluated* and compared with respect to the overall design requirements. The co-existence of potentially conflicting assumptions and their derivatives within a computational design support environment presents a problem for maintaining the truth of the knowledge and discriminating alternative design solutions by their contexts.

An ATMS is preferable to other truth maintenance systems when different knowledge sources, including users, hold distinct perspectives over the same problems, because it can maintain the consistency of the overall system as well as create contexts for each perspective. An ATMS is particularly suitable for design exploration because of its incremental updating of a dependency network and its ability to maintain a multi-contextual environment to allow different design solutions to be explored simultaneously. The application of ATMS has been reported in [Smithers *et al* 1990, Logan *et al* 1991, Smithers *et al* 1993 and Banares-Alcantara 1991]. But none of these systems formulated design context within a general knowledge-based design support system architecture and none of them explored the potential of a design context management system involving an integration of an ATMS and a blackboard control system for concurrency management in the domain of engineering design [Tang 1995d].

2.3.3 Design Context Management, an Example

Based on the above discussion and review, the role of a design context management system in a knowledge-based design support system architecture can be highlighted as being:

- to maintain the design results and their justifications generated during the design process;
- to provide easy access to, and a good explanation of the existing knowledge in the system;
- to make the best use of the knowledge already held in the system to generate new knowledge without performing redundant inferences; and
- to help designers compare different, sometimes conflicting design solutions.

For example, the diameter (d) and speed (n) of a motor may be constrained by the two separate constraints as illustrated in Figure 2.7 (Nodes A1 and A2). Each constraint may represent a designer's preference on the use of a particular design method. In this example A1 represents an assumption made by designer A whilst A2 represents the assumption made by designer B. These two assumptions in the system represent an initial design problem structure that the two designers want to explore.

Node	Datum	Justification & Context	Status
A1	$d = 9.81 \cdot \sqrt{n} / 2$	{A1}	True
A2	$d = 0.3 \cdot \sqrt{n} / 2$	{A2}	True
A3	$n = 125$	{A3}	True
A4	$n = 200$	{A4}	True
NG1	Detected by system	A1 & A2 -> False	False
NG2	Detected by system	A3 & A4 -> False	False
N5	$d = 54.83$	{{A1, A3}} --> N5	True
N6	$d = 1.67$	{{A2, A3}} --> N6	True
N7	$d = 69.36$	{{A1, A4}} --> N7	True
N8	$d = 2.12$	{{A2, A4}} --> N8	True
NG3	Assumed by user	A1 -> False	False
NG4	Assumed by user	A3 -> False	False

Figure 2.7: Exploration of Alternative Design Solutions

The two designers (A and B) might want to explore the possible values of both d and n in different ways. Suppose Node A3 ($n = 125$) in Figure 2.7 is an assumption made by designer A and Node A4 ($n = 200$) is an assumption made by designer B.

Given this information, the system will detect an inconsistency and create two inconsistent nodes (NG1¹ and NG2). This inconsistency is detected because the system assumes that:

$$\{A1, A2\} \rightarrow \text{false}$$

¹ NG denotes No-Good node.

{A3, A4} -> false

This means that two different constraints for the same variables, or two different values of the same design variable, together cannot derive anything meaningful for the initial problem structure. However, A1 and A2, A3 and A4 can still be considered independently in different contexts. Based on these 4 assumptions and two inconsistent nodes, a constraint-based reasoning system will derive the values for d , resulting in 4 nodes being created (N5, N6, N7 and N8) as shown in Figure 2.7. Node 5, 6, 7, and 8 represent different values of d derivable from four separate contexts.

The value of $d = 54.83$ (Node 5), for example, is a consequent node justified by two antecedent assumption nodes A1 and A3. It therefore represents a solution expected by designer A because A1 and A3 are the assumptions made by designer A. Node 8 represents the result expected by designer B because it is derived in the context of A2 and A4.

Neither of the designers may have expected the values represented by Node 6 and Node 7. But the system derives them because they have valid contexts as a result of an overlap in the two designers' concerns. That is, the two designers happen to be exploring the same design problem in different ways. Therefore both are valid alternative design solutions justified by the current information held in the system.

In the above example, design solutions are explored simultaneously by the designers despite the fact that there are conflicts on the choices of design variable values and constraints. For example, although A3 and A4 represent a conflict on the value of design variable n , there is no need for an immediate resolution for such a conflict. The designers can explore alternative design solutions in their own contexts. The solutions emerging from these contexts can be compared with each other only when it is considered necessary by the design team. For example, if later in the design process it is concluded through evaluation or negotiation that A2 and A3 are incompatible with some other design considerations, or rejected by other members of the design team, then two assertions can be added to the system.

A2 -> false

A3 -> false

This means that assumption A2 and A3 are no longer considered *true* by the design system (this is reflected in Figure 2.7 by NG3 and NG4). As a result, the justifications of various nodes currently held in the system will be automatically adjusted by the ATMS, the result of which is illustrated in Figure 2.8.

The rejection of A2 and A3 resulted in N5, N6 and N7 becoming unjustified (this is indicated in Figure 2.8 by an empty justification $\{\{\}\}$). After this is done the only justified solution, or the consistent context for d is Node 8. Subsequent inferences made by the system will then only carry this value forward.

Node	Datum	Justification & Context	Status
A1	$d = 9.81 \cdot \sqrt{n} / 2$	$\{\{\}\}$	False
A2	$d = 0.3 \cdot \sqrt{n} / 2$	$\{A2\}$	True
A3	$n = 125$	$\{\{\}\}$	False
A4	$n = 200$	$\{A4\}$	True
N5	$d = 54.83$	$\{\{\}\} \rightarrow N5$	False
N6	$d = 1.67$	$\{\{\}\} \rightarrow N6$	False
N7	$d = 69.36$	$\{\{\}\} \rightarrow N7$	False
N8	$d = 2.12$	$\{\{A2, A4\}\} \rightarrow N8$	True

Figure 2.8: Status of the System after Negotiation

Real design applications involve much more complicated design constraint networks and decision processes than the above presented simple example. However, the mechanism works in the same way. This mechanism is particularly suitable for design exploration because of its incremental updating of a dependency network and its ability to maintain a multi-contextual environment to allow different design solutions to be explored simultaneously. Designers within a design team can selectively work on subsets of design variables and constraints by creating their own contexts, or inherits others' contexts for further exploration. After the negotiation, alternative solutions are still kept in the system. Only their status and justifications become false [Tang 1995c and 1995d].

2.4 User Modelling

A knowledge-based design support system needs to model the close co-operation between the designer and the system by providing explicit explanations of the system's behaviour, and by providing ways for designers to intervene in the design process at various stages of design exploration. Designers' expertise, intuition and understanding of design problems play an important role in this co-operation.

In a knowledge-based design support system, the user of the system (a design or a team of designer) can be treated as special high-priority knowledge sources. In EDS, DICE and HOBS, for example, the user is treated as a special knowledge source [Smithers *et al* 1990b, Sriram *et al* 1992 and Carter *et al* 1991].

User modelling is concerned with data input, inferencing control and result explanation. Clarke [Clarke *et al* 1991] proposed a framework for an Intelligent Front-End (IFE) for making more productive the human/computer relationship. An IFE tackles the two fundamental problems that underly the use of computer-based support systems including design systems: the quantity and nature of the data being manipulated and the expertise and conceptual outlook of the user.

The approach of IFE is to construct an *intelligent user interface* which employs the knowledge of the user and styles of interaction required by different types of users. Such an intelligent user interface allows designers to more easily abstract an engineering artefact into a form suitable for computer manipulation and then initiate and control the required evaluations.

In IFe, for example, a generic *Forms Program* manipulates a set of forms which correspond to a given level of users. Each form entity (a labelled field, a button, a multi-option pop-up etc.) corresponds to a particular concept within the user conceptualisation in question such as *number of rooms* or *heights of windows*. Groupings of related concepts are located on the same form to comprise a meta-concept such as *building geometry*. A set of meta-concepts (forms) then defines a particular user conceptualisation in terms of only those related concepts that are acceptable to the user type the conceptualisation represents. Through these forms the user can ask about concepts as well as associate values with them.

Ganeshan developed a model for design decision making that explicitly identifies the intent of the user by recording the design decision process [Ganeshan *et al* 1991]. In this model, the intent behind a decision is the set of all the objectives starting from the highest-level of objectives, including all their refinements, leading to the decision. The design paradigm in this model is characterised by five kinds of activities:

- determine design focus,
- refining the objectives,
- evaluating design alternatives with respect to objectives,
- selecting an alternative based on the results of evaluations, and
- resolving conflicts among various design objectives.



In this approach a design record is a tree whose nodes represent design states and whose arcs represent the activity that caused the transformation from one state to the next. The current state of the design is the description of the artefact plus the objectives, both satisfied and unsatisfied.

In addition to the user type issue, a knowledge-based design support system must be able to answer user questions such as '*what has been derived?*', '*how is something derived?*', and '*what could be done next?*' etc. It should allow the user to:

- specify particular problem solving strategies;
- provide additional constraints; and
- make modifications to any part of the design that has already been derived.

It is necessary to define the role of a design documentation and explanation system that can record the design solutions as well as the design decision process [Smithers *et al* 1991b]. In particular, a record of design history in terms of the sequence of decisions made during a design session can be used to:

- explain how and why a particular decision was made to different types of user;
- determine whether the design solutions are consistent with the intended characteristics; and
- reuse design solutions or replay the design process in the synthesis of new designs.

In the architecture developed in this thesis, a design documentation system is implemented to address the user modelling issue. In this architecture, the user is a special knowledge source that can:

- inspect the status of the system via a graphical explanation system;
- control the system's inference by making decisions on the data and design method used and on the values of design variables; and
- record design results as design documents, and design process as design history.

This issue of documenting and explaining design results and recording design history will be discussed in chapter 6 when the architecture of a knowledge-based design support system is to be presented.

Summary

Knowledge-based design techniques provide knowledge representation methods that suitably represent and manipulate different types of design objects, heuristic design knowledge, and design constraints; they provide inferencing support and control mechanisms for designers to perform design tasks, including analysis, synthesis, evaluation and optimisation; they also provide mechanisms for detecting and resolving conflicts among various design requirements and design constraints.

The blackboard control strategy is suitable for developing integrated design systems. However, few systems have attempted to develop a generic system architecture that integrates the ATMS with a blackboard architecture to form a knowledge-based design support system kernel. In DICE, for example, the role of truth maintenance in its architecture is identified as to provide justifications for a knowledge module. But the use of an assumption-based truth maintenance in the exploration and maintenance of multiple contexts of design is not explored in DICE [Sriram *et al* 1992].

The competence of a knowledge-based design support system architecture (to be presented in chapter 6) can be evaluated based on the following criteria:

- It should help designers to construct and extend the design knowledge base within which domain concepts, design objects, dependency information of design objects are well structured and consistently maintained.
- It should help designers to derive solutions quickly from initial, not necessarily complete and consistent, design requirements, in other words it should provide an efficient mechanism to transform an initial design requirement description to a design specification.
- It should provide explicit explanations and justifications for any chosen aspects of the current status of a design, not only in terms of how something has been derived, but also to explain why something is not happening as expected. Locating areas of difficulty and suggesting strategies for solutions contribute to good decision making in design.
- It should allow designers to vary data, design requirements, problem solving strategies, or evaluation criteria, to obtain alternative designs. Simply speaking, a

knowledge-based design support system must support multiple-context problem solving so that a design problem can be explored from many different perspectives by a team of designers working in a co-operative manner.

- It should provide mechanisms for capturing and refining design knowledge so that the design knowledge base can be incrementally enlarged and enhanced. In other words, it should have some learning facilities to support conceptual design tasks.

The suitable integration of an ATMS within a blackboard architecture is one of two main contributions of this thesis to the development of knowledge-based design support system architecture. This integration allows the development of a design context management system capable of supporting the exploration of multiple design contexts by a design team. This, for example, cannot be done by EDS, HOBS, DICE, IFe and IDF.

3.1 Supervised Inductive Learning

An important distinction in inductive learning systems is whether the examples used for learning have been pre-classified or not (Geman et al 1989). When the learner is presented with explicitly classified (supervised) examples, the learning process is supervised because that is the situation where a teacher helps a student to learn a concept by giving further (often) examples of that concept. Inductive learning systems that do not pre-classify examples are called unsupervised systems. The main feature of a supervised learning system is to learn something from examples so that the learned concept can be applied to the same class of problems. As such, supervised learning systems require pre-classification of training examples.

According to the origin of a learning process and the strategy adopted, approaches to inductive learning can be divided into main two: inductive learning of symbolic representations (generalisation and specialisation) (Dizdarevic et al 1981 and Forsyth 1984) and inductive learning of structural rules via classification (Quinlan 1986, DeRaedt 1986, Mitchell 1977). DeRaedt (1977) defines the inductive learning of a structural description as

a heuristic search through a space of symbolic descriptions, guided by the application of various inference rules in the light of structural elements.

The basic methodology in this approach is *generalisation-and-specialisation*. Quinlan's *inductive learning of structural description* and Mitchell's *candidate elimination* (Mitchell 1981) are typical examples of learning systems that employ this methodology (Quinlan 1975; Mitchell 1977 and 1982). A learning system using this methodology is given a set of examples, some of which are examples of the concept to be

Chapter 3

Inductive Learning Techniques

In this chapter, computational inductive learning techniques are reviewed and discussed. The major features of inductive learning including supervised concept learning systems and unsupervised learning systems are discussed. This discussion provides a theoretical background for chapter 4, the focus of which is to describe a design concept learning system within a knowledge-based design support framework based on a discussion of the relationship between inductive learning and design. Other systems that utilise inductive learning techniques in design [Mostow *et al* 1987, Persidis *et al* 1989, Reich *et al* 1991 and Duffy *et al* 1993] will be discussed in the context of design support in chapter 4.

3.1 Supervised Inductive Learning

An important distinction in inductive learning systems is whether the examples used for learning have been *pre-classified* or not [Gennari *et al* 1989]. When the learner is provided with explicitly classified input/output examples, the learning process is *supervised* because this is like the situation where a teacher helps a student to learn a concept by giving him/her explicit examples of that concept. Inductive learning systems that rely on pre-classified examples are called supervised learning systems. The main concern of a supervised learning system is to learn something from examples so that the learned concept can be applied to the same class of problems. As such, supervised inductive systems require the pre-classification of training examples.

According to the output of a learning process and the strategies adopted, approaches to inductive learning can be divided into mainly *inductive learning of structural descriptions via generalisation and specialisation* [Dietterich *et al* 1981 and Forsyth 1986], and *inductive learning of decision trees via classification* [Quinlan 1988]. Michalski [Michalski 1977], defines the inductive learning of a structural description as

a heuristic search through a space of symbolic descriptions, generated by the applications of various inference rules to the initial observational statements.

The basic methodology in this approach is *generalisation-and-specialisation*. Winston's *inductive learning of structural description* and Mitchell's *candidate elimination (version spaces)* are typical examples of learning systems that employ this methodology [Winston 1975, Mitchell 1977 and 1982]. A learning system using this methodology is given a set of examples, some of which are examples of the concept to be

sought (the positive instances) and some of which are not (the negative instances). The task of learning is to construct a relevant concept or structural description that characteristically describes all the positive instances but none of the negative instances. In this approach, the commonalities among structural descriptions for the same type of configurations are first explored through generalisation. The significant differences between positive and negative examples are then found through specialisation of the concept so far formed.

A *specific-to-general* strategy for learning conjunctive concepts is to initialise a concept to contain the properties of the first positive instance, and gradually drop properties as they fail to be observed in subsequent positive instances. An inverse strategy (*general-to-specific*) begins with a maximally general concept (e.g., the empty concept of no conditions) that is gradually specialised by adding conditions that exclude negative examples. A system that combines both specific-to-general and general-to-specific has the ability to carry out a bi-directional search during learning. Mitchell's version space is a typical incremental approach to concept learning [Mitchell 1977].

The version spaces approach works by maintaining sets of possible descriptions and evolving those sets as new examples are presented. In this approach concept learning is viewed as a problem of searching for possible *generalisation/specialisation* of concepts that match the positive instances but none of the negative instances. Mitchell described the task of learning in version spaces as:

Given: a language in which to describe instances and generalisations, a matching predicate that matches generalisation to instances, and a set of positive and negative instances of a target generalisation to be learned,

Determine: generalisations within the provided language that are consistent with the presented training instances (i.e., plausible descriptions of the target generalisation).

3.1.1 Concept Learning System

Concept Learning System (CLS) is a system developed by Hunt for the *induction of decision trees* which emphasises the importance of deciding on important features of the examples in order to generate simple tree structures [Hunt *et al* 1966]. In CLS, a simple *attribute-value* representation is used and the way in which the most distinguishable features of the instances are decided at each stage of the classification is decided by a probability calculation on the possible values of all the attributes. That is, the most widely shared feature of all examples is selected to split the decision tree. Although CLS used a simple attribute-value representation, it provided a good basis for investigating a class of

inductive learning problems based on a divisive approach. The major CLS learning algorithm can be outlined as follows:

Initialise by setting variable T to be the complete training set. Then apply the following four steps to T:

1. If all elements in T are positive, create a 'yes' node and stop.
2. If all elements in T are negative, create a 'no' node and stop.
3. Otherwise select an attribute F with values v_1, v_2, \dots, v_n (n is the number of values an attribute might take); partition T into subsets T_1, T_2, \dots, T_n , according to their values of F; create a branch with F as parent and T_1, T_2, \dots, T_n as child nodes.
4. Apply 1,2,3 recursively to each child node.

3.1.2 ID3

The CLS classification algorithm does not have any principled criterion for deciding how to split the instances at any given point. This means that the algorithm may build deep, inefficient decision trees. Quinlan further developed Hunt's idea and built an induction system called ID3 [Quinlan 1979 and 1988]. The ID3 system has found some commercial uses as the approach has a more formal method than that of CLS in splitting the decision tree. In ID3 an *information theoretic heuristic* is employed for deciding how to split sets of instances. This enables the algorithm to produce shallower trees. In addition, it uses heuristic knowledge for selecting a focused set of examples for learning. This enables the system to cope with large training sets.

A decision tree in ID3 is constructed as *nodes, arcs* and *leaves*. A node corresponds to an attribute of the objects to be classified, while arcs refer to the alternative values of an attribute. Leaves of the tree represent the sets of objects in the same class. A decision tree is a particular kind of rule, or set of conjunctive rules: each path from root to leaf represents a rule. An arbitrary input can be processed by simply applying the tree to the input, i.e., propagating the input down through the tree. Such a decision tree is mostly used for classifying data for information retrieval purposes.

ID3 employs a generate and test strategy to build up decision trees, preferring simple trees over complex ones, based on the assumption that simple trees are more accurate classifiers of unseen inputs. It begins by choosing a random subset of the training examples called a window. The algorithm builds a decision tree that correctly classifies all examples in the window. This tree is then tested on the remaining examples. If all the

remaining examples can be classified correctly, the algorithm halts. Otherwise, it adds a small number of randomly chosen examples to the window and then generates a new tree.

In ID3, building a new node means choosing some attributes to test. At a given point in the tree, some attributes will yield more information than others. For example, testing the attribute colour first for classifying cars would not produce a simple tree if the colours of all cars are different. Ideally, an attribute will separate training examples into small subsets whose members share a common label (classes), in which case branching is terminated, and the leaf nodes are labelled. However, ID3 is based on a simple attribute-value representation and there is no way to utilise background knowledge during learning.

3.1.3 Inductive Logic Programming

Recently Quinlan has described a program called FOIL, which induces first-order Horn Clauses. The method relies on a *general to specific* heuristic search which is guided by an information criterion related to entropy. This approach has been seen as being a natural extension of ID3. FOIL extends ID3 using an approach called Inductive Logic Programming.

Inductive Logic Programming refers to the work on problems of inductive reasoning within the confines of pure Prolog [Muggleton 1992]. It is the intersection of logic programming and machine learning. In the general inductive setting, a learner is provided with three languages: LO (the language of observations), LB (the language of background knowledge), and LH (the language of hypotheses). The inductive inference involves the use of background knowledge to construct a hypothesis which agrees with some set of observations. Recently Muggleton has defined the term *Universal Learnability* with inductive logic programming based on Inverse Resolution (IR) and Related Least General Generalisation (RLGG) [Muggleton 1992].

3.2 Unsupervised Inductive Learning

All the above discussed approaches rely on externally defined categories. Without prior knowledge about the categories of the training examples, learning systems must use internalised heuristics to organise their observations into categories. Discovering regularities or similarities from examples without prior knowledge of the categorisations of the examples is termed *unsupervised learning*. That is, in an unsupervised learning system, there are no explicit examples of desired input/output associations. There is only a general notion about the way in which input examples should be mapped to output class definitions. In most cases this general notion specifies how the similarities of the input

examples are to be measured. While the task for a supervised learning system is testing hypotheses, clustering methods are intended for generating hypotheses.

3.2.1 Cluster Analysis

The task of unsupervised learning is to discover how some data are divided into similar classes, and what these classes are. Many unsupervised learning procedures closely resemble statistical clustering procedures based on *numerical taxonomy* [Everitt 1981]. Learning algorithms in this paradigm deal with the problem of grouping or clustering similar instances into the same category or segregating dissimilar instances into distinct categories.

Clustering must involve the production of a class hierarchy using a numerical measure of similarity defined over an instance space. In particular, the instances in this space are typically collections of numerical measures of attributes describing the features of observations. The motivation for clustering analysis is that a cluster provides useful information for further and deeper exploration of data. Because a cluster of similar objects may share some features, or a common cause, this information can be used to build some procedures which infer from the organised data in a more accessible and efficient way than on the original data. In AI, *cluster analysis* is referred to as *Unsupervised Pattern Recognition* [Everitt 1981].

Everitt states the problem of cluster analysis as:

Given a collection of n objects, individuals, animals, plants etc., each of which is described by a set of p characteristics or variables, derive a useful division into a number of classes. Both the number of classes and the properties of the classes are to be determined.

At the beginning of a cluster analysis, both the *number* and the *composition* of the classes are unknown. The solution to a cluster analysis problem normally consists of a pattern of n objects (a set of clusters). Each object in the training examples belongs to one cluster only and the complete cluster, i.e., the structure of the data set, contains all the objects. In cluster analysis the measurement of distance is an important consideration as the recognition of clusters involves the assignment of relative distances between points. A clustering system must therefore be concerned with the following issues: representation of data; similarity measurement of clusters; control of the clustering process; and evaluation and interpretation of results.

3.2.2 Hierarchical Clustering Techniques

In a hierarchical approach to clustering, the original data set is not partitioned into a number of clusters in one single step. Instead the classification process consists of a series of partitions which run from a single cluster containing all individuals, to n clusters each containing a single individual, resulting in a hierarchical tree structure in which each node represents a cluster. There are *hierarchical clustering* methods which differ in the direction in which data are processed. These are *agglomerative approaches* (bottom-up) and *divisive approaches* (top-down). The former starts the process at the bottom of the tree by successively clustering the n individuals into a series of groups, while the latter starts it from the top by separating the n individuals successively into finer groupings. Hierarchical classifications can be represented by a two-dimensional diagram known as a *dendrogram* which illustrates the fusion or divisions made at each successive stage of the analysis [Fisher *et al* 1985], as shown in Figure 3.1.

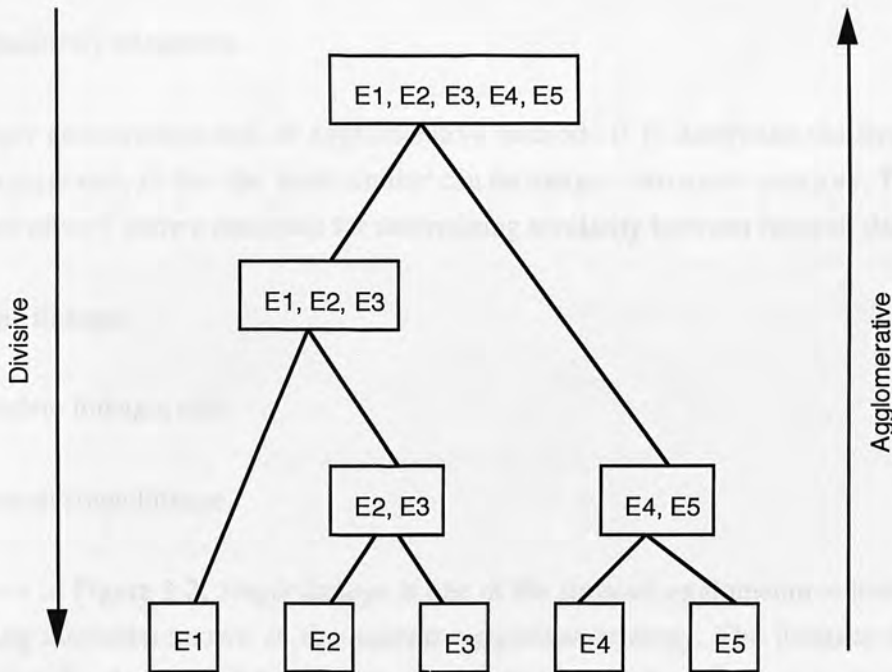


Figure 3.1: Hierarchical Clustering Methods

3.2.3 The Agglomerative Approach

An agglomerative hierarchical clustering procedure produces a series of partitions of data, P_n, P_{n-1}, \dots, P_1 . The first P_n , consists of n single-member clusters, and the last P_1 consists of a group containing all n individuals.

The basic operations for all such methods are similar, and can be outlined as:

1. Creating n clusters $C = [C_1, C_2, \dots, C_n,]$ each of which contains a single observation.
- 2 Finding the nearest pair of distinct clusters, for example, C_i and C_j , merge C_i and C_j into C_i .
3. Deleting C_j from C and decreasing the number of clusters by one. If the number of clusters is one then stop, else return to 1.

The algorithm effectively works bottom-up, trying to build ever larger clusters. At each step the method merges the individuals or groups of individuals that are most similar. Different methods differently define distance (or similarity) between individuals and groups containing several individuals, or between different groups of individuals [Everitt 1981].

3.2.4 Similarity Measures

The major computation task of agglomerative methods is to determine the similarities among categories, so that the 'most similar' can be merged into a new category. There are a number of well known measures for determining similarity between items of data:

- single linkage;
- complete linkage; and
- group-average linkage.

As shown in Figure 3.2, *Single linkage* is one of the simplest agglomerative hierarchical clustering methods, known as the *nearest neighbour* strategy. The distance between groups is defined as that of the closest pair of individuals, where only pairs consisting of one individual from each group are considered. The single linkage distance is calculated using the following formula:

$$\sum_{k=1}^n [(x_{ik} - x_{jk})^2]^{1/2}, \quad (3.1)$$

where x_{ik} and x_{jk} are the values of attribute k for observations i and j , respectively. The closer the values of individual attributes, the smaller the distance, and the greater the similarity.

The *complete linkage* or *furthest neighbour* clustering method is the opposite of single linkage in the sense that the distance between groups is defined as that of the most distant pair of individuals, one from each group.

The distance between two clusters in *group average linkage* is defined as the average of the distances between all pairs of individuals that are made up of one individual from each group. Empirical studies have shown average and complete linkage as the most useful in practice [Everitt 1981].

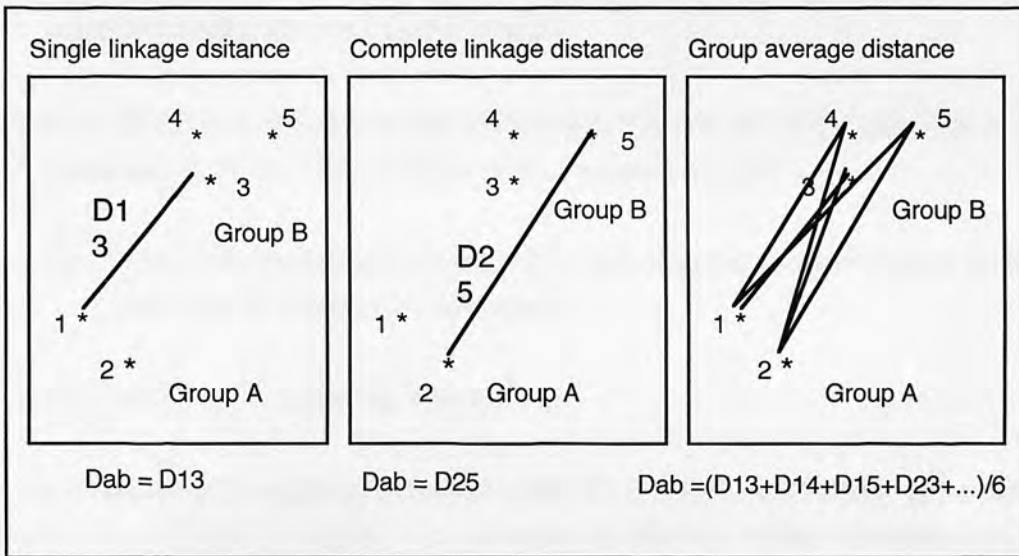


Figure 3.2: Similarity Measures

3.2.5 The Divisive Approach

The divisive method works top-down, repeatedly trying to break down instances into smaller groups. A major concern of this method is to identify the most decisive factors of the observations to construct a simple tree. Divisive clustering techniques are further divided into two types, the *monothetic* technique which divides data on the basis of the possession or otherwise of a single specified attribute, and the *polythetic* technique which divides observations based on the values of all the attributes.

The divisive method measures *dissimilarity* during the course of splitting the instances into groups. In this method, a so-called *splinter* group is accumulated by sequential addition of the individual whose total dissimilarity with the remainder, less its total dissimilarity with the splinter group, is a maximum. When this difference becomes

negative the process is repeated on the two sub-groups. The usual measure of dissimilarity in the divisive approach is the *average Euclidean distance* between each individual and the other individuals in the group. The algorithm of a divisive clustering is as follows:

1. Create a top-node containing the whole set of instances;
2. Stop if the node contains only one instance;
3. Find one instance whose dissimilarity with the rest of instances under the current node is a *maximum*. Call this instance S and the rest of the instances R, and then create two nodes, one for S and one for R
4. Find the instance in R whose total dissimilarity with the rest of individuals in R, less its total dissimilarity with instances in S, is a maximum, and
 - 4.1 if this maximum value is larger than zero, than remove the instance from R, insert it to S, and go to 4, otherwise
 - 4.2 for S, and R, repeat the steps from 2.

For example, given a group of instances (I1, I2, I3, I4, I5), the splinter group can be initialised to, say (I1), if I1 gives the maximum dissimilarity to the remainder than any other instances. That means I1 is most dissimilar to the remainder. This forms a splinter group (I1) and the remainder (I2, I3, I4, I5). Next each element in the remainder is examined in terms of its dissimilarity with the splinter and with the rest of individuals in the remainder. If, for example, I3 is found to be the one whose total dissimilarity with the remainder, less its total dissimilarity with the splinter group, is a maximum, then I3 is added into the splinter group, form two new groups, (I1, I3), and (I2, I4, I5). If by now for the every member of the remainder, its dissimilarity with the remainder, less its total dissimilarity becomes negative, then the splitting at this level stops, leaving two groups (I1, I3) and (I2, I4, I5). The whole process can then be applied to (I1, I3) and (I2, I4, I5), allowing the tree to descend further down.

3.3 Concept Formation Systems

Numerical taxonomy based procedures in cluster analysis divide data solely on the basis of the attributes of the instances themselves. Therefore there is no easy way of taking into account the semantic relationships between instance attributes, or global concepts which

are relevant to the classification being developed. Everitt points out, *sometimes the assumption that useful clusters will arise from a series of local, pairwise comparisons may be invalid* [Everitt 1981]. In addition hierarchical methods cannot repair what has been done in previous steps. Since all agglomerative hierarchical techniques ultimately reduce data to a single cluster containing all the individuals, and the divisive techniques will finally split the entire set of data into n groups each containing a single individual, the user wishing to have a solution with an 'optimal' number of clusters has to decide on which particular stage to stop at.

Duffy and Kerr observed this problem when applying a clustering method in their investigation of customised perspectives [Duffy *et al* 1993]. Furthermore, most similarity-measures are best suited to numerical data. Techniques are needed for dealing with nominal data (e.g., the proportion of identically valued attributes in two observations). In AI applications there are two additional goals: to facilitate the application in AI domains where nominal data are frequently found, and to incorporate some of the human analyst's search strategies into the clustering program itself.

Concept clustering is an approach that attempts to address these problems. A concept clustering system uses additional heuristic knowledge to guide the clustering process so that the structure of instances derived represents some context dependent concepts. The role of concept clustering is to discover appropriate classes as well as concept descriptions for each class. There is an additional intention that each node in the classification should be a concept that can somehow be evaluated in terms of compactness, naturalness of interpretation by humans, or the level at which a concept can be easily recognised, retrieved, or distinguished, etc. It is the explicit coupling of *characterisation* and *clustering* that makes *concept clustering* a different approach from numeric taxonomy.

Fisher and Langely [Fisher *et al* 1985] view conceptual clustering as composed of two sub-tasks: clustering, which determines useful subsets of an object set; and characterisation, which identifies a concept for each *extensionally* defined subset discovered by the clustering process.

Unsupervised systems that employ non-incremental learning strategies require all training instances at the outset. However, a system should be capable of adjusting concepts when new information become available. Concept formation has essentially the same goals as that of conceptual clustering. The main difference between the two is that in concept formation there is an additional constraint on the way in which the concept is induced. That is, the process of learning is *incremental*. A learning system is said to be incremental if it accepts instances one at a time, and if it does not extensively reprocess previously encountered instances when encountering new ones [Gennari *et al* 1989].

Given a sequential presentation of instances and their associated descriptions, a concept formation system must find: *clusters* that group those instances in categories; an

intentional definition (a concept) for each category that summarises its instances; and a *hierarchical organisation* for those categories. The essence of concept formation is to embody external heuristic knowledge into the learning process to form useful concepts rather than simple clusters of data.

The main features of a concept formation system are the hierarchical organisation of concepts, top-down classifications, and an unsupervised, incremental and hill-climbing approach to learning [Gennari *et al* 1989]. The presence of a concept hierarchy suggests that classification begins at the most general (top) node and sorts the instance down through the hierarchy. However, in concept formation an instance can be sorted down more than one branch, indicating that some concepts may overlap. In addition, the sorting process may stop at a node higher than a terminal node in the hierarchy.

In a concept formation system, a description, a definition or an image in the concept hierarchy which represents a concept that matches a subset of the instances is stored in a node of the concept tree. By doing so the system *remembers* what has been learned and is therefore able to compare a new instance with one contributed to a current concept definition. In such a system, classification is based on some similarity measures but the external knowledge of the training instances is also used to control the process of growing the tree, thus influencing the number of classes to be clustered which in turn affects the formulated concepts.

In order to understand the main features of a concept formation system, it is necessary to review in this section some of the early and typical concept formation systems. These systems demonstrate the important features of concept formation technique although they have different control and learning strategies and evaluation methods for the concept descriptions.

3.3.1 EPAM

Elementary Perceiver And Memorizer (EPAM) [Feigenbaum 1961] is an early incremental concept formation system originally built to simulate human behaviour in rote learning of nonsense syllables [Feigenbaum *et al* 1984].

EPAM performs two fundamental tasks, *information retrieval* and *learning*: sorting items in a static net and growing the net incrementally through discrimination learning. The information retrieval task is performed by a sub-system by sorting a decision tree where each node has a concept definition as well as some instances that have already been classified under it. The classification at each node is directed by an example's value along a single attribute. The learning task is performed by a sub-system that learns to identify new instances that have yet to be classified by the tree, and therefore, allows the tree to grow.

The essential learning mechanism of EPAM *discriminates* and *memorises*. The primary information structure used in EPAM is a discrimination net. Image lists containing symbolic information about a partial or total copy of an instance, are stored in the leaves of the net. Testing functions called *tests* are stored in the nodes of the net. These functions are used to evaluate the characteristics of an instance and to signal *branch-left* or *branch-right*, based on results of the evaluation.

The discrimination net can be examined by a sub-system called the *net interpreter* which classifies an instance down the net and returns the image list associated with that instance. This retrieval process is considered to be the essence of a purely associative memory, i.e., the stimulus information itself leads to the retrieval of the information associated with that stimulus. The net interpreter achieves this by finding the test in the topmost node of the tree and calling its associated functions. The results of the function call leads the system to branch left or right to continue this process until a terminal node is reached. The name of the image list associated with the terminal node is then retrieved and the sorting process terminates.

A discrimination-learning sub-system is used in EPAM to grow a net. When a new instance is supplied, it is *added* to the net by classifying it down a path of matching branches to a leaf. At a leaf where the difference between the stored concept and the new instance cannot be further discriminated, the system creates a new test (a new node) to discriminate further upon this difference.

Typically, a new test tries to test an attribute of the instance that has not been tested further up in the net. For example, to distinguish the difference between two words, one possible test is to test the first few letters. If two words have the same first letter, then the second or third and so on can be used to further discriminate between the two words. The images of the new and stored instances are stored within the leaf of the new node along the appropriate branches of the new test. In this way EPAM discrimination net leaves begin as general patterns. They are gradually specialised by the so called *familiarisation* process as subsequent instances are classified.

The discrimination net that EPAM builds is called a monothetic tree as at each node only one attribute of the instance is tested. Moreover, the method retains concept descriptions (images) only at terminal nodes. This means that the final learning result is not a true hierarchical structure and the system only generates mutually exclusive concepts (i.e., allows no overlapping). Nevertheless, EPAM provides a way of modelling the memorisation of stimuli as a *general-to-specific* search at leaves, which reflects the gradual process of memorisation (familiarisation) that presumably occurs in human learning.

3.3.2 UNIMEM

UNIMEM is another incremental learning system based on the concept formation approach [Lebowitz 1987]. Like EPAM, UNIMEM uses a tree structure as its basic representation of concept categories and it demonstrates roughly similar learning and memorising behaviour as EPAM does. UNIMEM stores concept descriptions with each node in the hierarchy. Nodes high in the hierarchy represent general concepts, with their children representing more specific variants. However, unlike EPAM, in UNIMEM, both *terminal and non-terminal nodes* have associated concept descriptions. Therefore the system builds a *hierarchical concept structure*.

While in EPAM each link in the net is labelled with the result of a test, each link in UNIMEM is one of the results of multiple tests (multiple features). In UNIMEM, each instance may therefore be stored with multiple nodes so that categories need not be disjoint. In other words, EPAM builds *monothetic* trees while UNIMEM builds *polythetic* trees. Overlapping clusters are generally motivated in domains where there is prior belief that observations can be members of multiple classes. Each node in a UNIMEM tree represents a complex definition that covers a number of instances. This feature allows the system to handle instances with missing attributes and thus it enables a flexible sorting strategy.

UNIMEM starts with an initialised tree containing an empty root node. When a new instance is supplied, the system first searches the tree in a *depth-first fashion* for the most specific concept nodes that match the instance. The search process returns all the most specific nodes that explain the new instance. The system then generalises each node in this set in order for them to cover the new instance. It does this by comparing the newly added instance with all the other instances stored in the node. If two instances stored in a node appear similar, it creates a child node whose concept definition covers both instances, and relocates these two instances to this child node. Otherwise it simply stores the instance at the node. The search for a place to add the new instance is controlled by similarity matching involving *numeric attributes*.

UNIMEM introduced a so-called performance component in the concept hierarchy. Each node in the hierarchy has a description consisting of a conjunction of attribute-value pairs, with each value having an associated integer. This integer measures the *confidence in that feature or predictability*, i.e., how well the feature can be predicted given an instance of the concept. In addition, each feature on a link has an associated integer score specifying the number of links on which that feature occurs. This score measures the *predictiveness* of the feature, i.e., how well it can be used to predict instances of the various children.

With these two scores it is possible for the system to prune the tree using some pre-defined threshold values. For example, when the predictability score for a feature exceeds

a user-specified threshold, then the system considers that feature as part of the node's permanent description, so that the future instances no longer affect the node on that feature. On the other hand, if a feature's predictability score drops below a threshold, then the system removes that feature from the node.

The measures of predictiveness and predictability are, however, informal and they have no clear semantics [Gennari *et al* 1989]. The system is dependent on *user-specified parameters* to make decisions on both the number of features that must be matched and the measures of closeness used.

3.3.3 COBWEB

Some learning systems build probabilistic concepts rather than deterministic concepts based on the belief that a classification is considered good if the description of a design can be guessed with high accuracy given that it belongs to a specified class, and that a good cluster is a result of minimising the distance between attributes within a cluster and maximising the distance between attributes in different clusters. Here the distance between two objects shows their *dissimilarity*. The quality of a cluster can therefore be determined by whether it maximises *intra-cluster similarity* and minimises *inter-class similarity*.

One of the key characteristics of a concept formation system is the use of heuristic knowledge in controlling the search for intentional concepts [Fisher 1987]. COBWEB is a system that makes use of a heuristic that partitions a set of designs into mutually exclusive classes [Fisher *et al* 1991]. This heuristic called *category utility* [Gluck *et al* 1985] measures the increase of property-value pairs that can be guessed above the same quantity guessed based on frequency alone. Category utility is an optimum trade-off between intra-class similarity and inter-class similarity. It assumes that a good cluster must maximise the category utility:

$$CU(C_k) = P(C_k) [\sum_j P(V_j|C_k)^2 - \sum_j P(V_j)^2] \quad (3.2)$$

where $P(C_k)$ is the probability that an instance is covered by a category C_k , $P(V_j|C_k)$ is the category predictability of value V_j given C_k ; and $P(V_j)$ is the base rate probability of V_j . Category utility can be regarded as the increase in the expected number of attribute values that can be correctly guessed in C_k ($P(C_k) P(V_j|C_k)^2$) over the expected number of correct guesses with no such knowledge $P(V_j)^2$.

COBWEB incorporates new instances into a classification tree in an incremental fashion. Each node in the classification tree represents the non-deterministic but probabilistic concept of a class. It keeps count of the number of times a particular value

appears in the instances stored in a given node. Thus, a node has associated with it a set of counts, each of which correspond to the number of appearances of a particular attribute of the instances stored under the node. Probabilities are derived simply as ratios between counts. Therefore the incorporation of an instance becomes a process of classifying the object by descending the tree along an appropriate path, updating counts along the way, and performing one of the following operations at each level:

- classifying the object with respect to an existing class;
- creating a new class;
- combining two classes into a single class; and
- dividing a class into several classes.

A new instance is added to the tree by first updating the base rate probabilities at the root to reflect the presence of the observation's attribute values. Each child of the root class is then assumed as a possible host of the new instance. The conditional probabilities of each child are tentatively updated as if the newly added instance has been placed in each category. This is to calculate the category utility in order to determine which node is best for the new instance.

By creating a new node for an instance, or merging the two best hosts or deleting one or more of the children of the best host, COBWEB dynamically manipulates the classification tree so as to maintain the quality of conceptual clustering. The use of category utility as the clustering heuristics means that COBWEB needs to represent instances and clusters in a way in which probability values can be derived and updated. COBWEB cannot determine whether a particular instance can be stored at a particular node because a node in the classification tree only represents the probability of an instance being classified by the node. For each distinct instance it can only derive the probability of it being covered by the particular node referred to as the best host. However, an important point is that it uses a heuristic search control method to impose a structure in the observation data that can be evaluated against some criteria. This is the essence of concept formation when one wants to obtain good clustering results influenced not only by the observations themselves but also by some other external heuristic knowledge.

Summary

This chapter has reviewed inductive learning techniques including supervised and unsupervised, incremental or non-incremental, divisive or agglomerative etc. in order to provide a technical background for studying the relationship between inductive learning and design and for describing a design concept learning system in the next chapter. Inductive learning systems differ in the following aspects:

1. **Types of description:** There are two different types of concept descriptions which can be sought by an inductive learning system, namely characteristic descriptions and discriminating descriptions. A characteristic description describes the most important features that characterise the patterns in most input examples in terms of their structural and behavioural similarities. In the absence of negative examples, i.e., in an unsupervised learning situation, it is only possible to derive characteristic descriptions that distinguish examples used from any other examples. A discriminating description, on the other hand, describes the input examples in terms of their difference from examples in some other classes, and is used to discover significant differences between two or more classes of objects.
2. **Control strategy and search direction:** two control strategies are employed by most inductive learning systems, divisive (top-down) and agglomerative (bottom-up). In unsupervised learning, for example, the search for clustering includes operations that merge existing clusters into larger clusters such as in agglomerative methods or that split clusters into groups of finer granularity such as in divisive methods. Together, these operations define the direction in which the search can proceed: from finer to coarser granularity or vice versa. In addition a learning system must use heuristic knowledge to avoid unnecessary and non-productive searches in a large space of hypothetical concepts and logic operators.
3. **Incremental and non-incremental learning:** a primary motivation for using incremental learning systems is that knowledge may be rapidly updated with each new observation, thus sustaining a continual basis for reacting to new stimuli. Incremental techniques produce the best-guess concept or the range of concepts consistent with the data so far (as in the version space approach) and can interleave learning and performance (as in the above concept formation systems reviewed). This is an important property of systems that are used under real world constraints [Carbonell 1990]. In these systems, each new instance triggers small changes to the current categorical structure. However, as a result of processing instances one by one,

these systems may suffer heavily from ordering effects and different categories may be discovered depending on the order in which they process observations.

4. **Evaluation criteria:** In a supervised learning system, it is possible to test a learning strategy by simply applying its output hypothesis (or classification tree) to a set of test examples. In an unsupervised learning system, however, it is only possible to test whether the same regularity discovered in the training examples can be observed in the testing examples. In some cases, this regularity is discovered by some similarity measurement which does not necessarily have a good conceptual explanation. Therefore it is important to introduce some heuristics into an unsupervised learning system that help measure the goodness-of-fit of the concept descriptions.

There exist a number of inductive learning algorithms for building knowledge acquisition tools for knowledge-based design systems. However, most of the currently available inductive learning algorithms are still limited to certain types of problem (such as diagnosis from attribute values). The potential of inductive learning techniques in knowledge-based design support is yet to be further explored.

A problem associated with the current inductive learning techniques is the inability to make use of background knowledge. Human inductive reasoners make use of vast amounts of background knowledge when learning [Muggleton 1992]. Inductive algorithms such as ID3, for example, use only a fixed set of attributes attached to each example. In design applications, inductive learning methods capable of utilising design heuristics as background knowledge to derive basic concept structures from raw data or past design examples incrementally and constructively are yet to be developed.

Most learning systems use simple attribute-value representation. Therefore they are unable to deal with complex structural representation [Dietterich *et al* 1981]. However, appropriate structures in a knowledge base can significantly improve the efficiency of both supervised and unsupervised incremental systems. This means that they cannot be used in areas, such as *temporal reasoning, scheduling, planning, qualitative reasoning, natural language and spatial reasoning*, requiring essentially relational knowledge representations. Knowledge-based design support systems must use sophisticated representation schemes in order to represent and reason about complex object structures.

In chapter 4, inductive learning techniques will be further discussed in the context of knowledge-based design support before an incremental learning model for a class of design problems and the implementation of a design concept learning system are presented.

Chapter 4

Inductive Learning and Design Support

In this chapter the issue of incorporating inductive learning techniques into a knowledge-based design support system is discussed first. An inductive learning model for design is then presented. The data structure and the control strategy of a new design concept learning system which is based on an object-oriented representation and an unsupervised learning approach is then described.

4.1 Inductive Learning and Design

Learning and designing are closely related activities: finding a new design solution involves the use of knowledge abstracted from previous designs; searching for an alternative design strategy can somehow be guided by the knowledge gained from a previous design failure; evaluating a design solution relies largely on the knowledge generalised from the features of, or simulated behaviour of, the design.

Some of the early machine learning application systems in engineering design appeared in [Yoshikawa *et al* 1989]. Persidis and Duffy argued that learning is *inextricably* linked to design and Machine Learning should become an integral part of research into intelligent CAD systems. They identified different types of design related knowledge, *design requirements*, *design descriptions*, *domain knowledge*, *case histories* and *design management*, that are continually altered during the stage of design problem analysis and design solution synthesis. They also pointed out that learning in design takes place when recovering from design errors, negotiating to resolve design conflicts, asking for advice from other design experts, or evaluating design solutions after the completion of a design [Persidis *et al* 1989].

Research projects at the Engineering Design Research Centre (EDRC at Carnegie Mellon University, USA) have focused on understanding the life cycle issues of design, i.e., how to create products which are manufactureable, disposable, and maintainable, and on developing the concepts needed to create design systems that allow the rapid creation and delivery of new as well as existing methodologies [Maher 1988 and Reich *et al* 1991]. The research in this area explored the potential of the SOAR system developed by Newell [Laird *et al* 1987]. SOAR provides two facilities not found in conventional expert system environments: a rich set of general or weak problem solving methods built into the architecture; and the ability to learn by generating new chunks of knowledge from the successful solutions of sub-problems.

In the early stages of the design process there is a great deal of uncertainty about

design problems and design constraints. Faced with these uncertainties, designers must develop a model for the artefact being designed, through the study of existing examples, past cases, and new design requirements, in order to:

- find out what it is possible to achieve, i.e., to define design goals;
- describe and constrain design problems, i.e., to define the design space and identify the operations to explore this design space; and
- define evaluation criteria in order to select good solutions.

In the absence of the explicit knowledge of how to build such a model of an artefact, a solution is to study known design examples, or previously recorded design cases, and design plans to build an abstract knowledge structure for further exploration. This knowledge structure may be a generalised model of past design solutions, or it may be a concept structure in which domain concepts related to the design problem at hand are organised. Therefore the early stages of design, i.e., discovering the structure of a design problem or defining the model of an artefact can be modelled as an *inductive learning process* during which a structural and characteristic description of a class of design problems is learned from previous design examples.

In small molecule drug design, for example, the structural vocabulary of a drug is straightforward, with molecules being composed of atoms. The laws of physics dictate the chemical and physical properties of molecules, and it is possible to predict such properties with reasonable accuracy [Hodgkin 1991]. However, properties of molecules in a biological context are less clear, making it difficult to relate a molecular structure to biological behaviour. One of the major approaches to small molecule drug design is to discover this structure-activity (behaviour) relationship by analysing a set of existing molecules that demonstrate similar biological behaviour to that of the sought after new molecule.

The discovery of a new drug can be seen as a learning process during which the initial structure of a new molecule is induced from example molecules: to learn a means of partially constraining the problem structure of a design in which important features of a possible new molecule are logically related to each other. This is different from normal inductive learning, which produces a general concept from examples. Supporting small molecule drug design by using a design concept learning system will be fully described in chapter 5, 6, and 7.

It is now widely acceptable that learning should be an integral part of an intelligent design system [Zhao *et al* 1988, Katai *et al* 1991, Brown *et al* 1991, Babin *et al* 1991, Matwin *et al* 1991, Chandrasekaran 1986, Anderson 1991 and Bratko 1993]. For

example, in the generic task approach, identifying some domain independent generic design tasks provides the basic components for a multitude of standard tasks usually carried out when assembling a complete model of a design process, as well as providing a basis for explanation-based learning techniques [Brown *et al* 1991]. In the EDS design model, newly generated knowledge is transferred from the dynamic knowledge base (in the form of design description document) to the design knowledge base [Smithers *et al* 1990a]. In the prototype-based approach, the creation of a design prototype is seen essentially as a learning task that generates conceptual knowledge in terms of functional, structural and behavioural properties of design components [Balachandrian *et al* 1990].

Design can be viewed as an evolutionary process in that it includes assimilation of new knowledge for future design through learning. As design progresses, designers not only increase their store of information in the form of facts, but also change the way in which information is organised. That is, they do not merely extract relationships that are explicit in particular situations, but also generalise and abstract concepts so that they can be used in different but analogous situations. In a design situation, experience consists of the knowledge about particular design products as well as the knowledge about the design process (how these products have been designed). A computer-based design support system needs to be able to model this evolutionary process effectively.

Learning in design is concerned with many issues in the acquisition, transformation, modification, generation and reuse of the design knowledge. Within the scope of this thesis, two fundamental problems about computational learning and design are addressed:

1. How to utilise inductive learning techniques to support conceptual design tasks; and
2. How to capture design product as well as design process information within a knowledge-based design support system architecture.

The first problem is addressed by developing a design concept learning system to support conceptual design tasks (in chapter 5 and 7) whilst the second problem is addressed by developing a design documentation system within a knowledge-based design support system architecture that can record and replay a design history (in chapter 6).

4.2 Application of Inductive Learning Techniques in Design

This section reviews some of the design systems utilising inductive learning techniques in the following aspects of design:

- Design knowledge acquisition;

- Design synthesis and evaluation; and
- Recording and replaying past design plans.

4.2.1 Design Knowledge Acquisition

Knowledge acquisition is seen as a bottle-neck problem in developing expert systems and knowledge-based design support systems [Quinlan 1986, Carbonell 1989 and Muggleton 1990]. To build and maintain a design knowledge base is difficult, time consuming and expensive [Burton *et al* 1990].

One approach to alleviating some of the difficulties in knowledge acquisition is the introduction of inductive learning into the development and maintenance stages of knowledge-based systems. Inductive learning techniques can be utilised to develop automatic knowledge acquisition tools to abstract knowledge from domain experts or databases, or refine knowledge already stored in the knowledge base. But so far no achievement of significant scale has yet been made.

A task in the knowledge acquisition for the development of a knowledge-based design support system is to develop a design object hierarchy in which design objects and design concepts are properly organised for the design tasks. A complete hierarchy, not all of which are necessarily used in each design project, forms a complete taxonomy of a design domain. Most knowledge-based system development tools allow the user to enter object classes as frames to develop a hierarchical network of concepts. Features of a frame can be inherited by other frames through a *kind-of* relationship in a top-down fashion. This helps users to realise how their expertise can be represented in an object hierarchy, making it easier to better organise or condense their knowledge.

Three learning techniques, i.e. *chunking*, *abstraction/specialisation*, and *simplification*, are considered to be particularly relevant to concept development activities during design [Persidis *et al* 1989 and Duffy *et al* 1993]. **NODES** is a learning system developed by Persidis and Duffy to utilise these techniques to support design concept modelling. In NODES, concepts are viewed as chunks of related information about some entities or objects that the designer interactively builds as part of a model to formulate a design problem and its solution. Such a model contains *objects*, their *characteristics* and *relations*, *rules* determining the performance/behaviour of the objects, and the *goals* of the design. In the model created by the designer with the assistance of the system, *nodes* represent the characteristics and *links* represent dependencies as contained in the relations.

A *concept library* is used in NODES to assist the designer in the model development stage of the design process. If the designer creates a concept that already exists in one of its libraries, the system adds to the current concept all the relevant knowledge relating to

that concept without having to ask the user to define it explicitly. This provides a means for reusing knowledge gained from previous design sessions. The library itself can be enlarged by saving newly developed concepts. In particular, newly developed concepts for a specific design problem can be generalised so that they can be of assistance to the designers in changing situations. However, in this system, only pre-defined concept structures can be used. The system depends on the user for selecting an appropriate abstraction for a new design.

Duffy and Kerr presented an approach for the rationalisation of design concepts using a clustering approach to rationalise past design examples into abstract groups for an engineering design support system [Duffy *et al* 1993]. In this approach, knowledge in past designs is organised as a topological structure, representing the viewpoints of design concepts that can be customised by the designers. In this system a *group rationalisation strategy* was used to generate a series of groupings based upon a selected attribute of past designs. In the learning process, the past designs were *clustered* first based on the most important aspect of past designs (an attribute shared by all the past designs) using a measure of similarity. This divided past designs into several groups. Some other aspects of past designs could then be used to further cluster each of these groups. The result of group rationalisation was a hierarchy of concepts, each of which consisted of three components, *name*, *range of the value on the chosen perspective* and *members* that had been clustered into the node.

In this approach, using grouping rationalisations with varying degrees of similarity increased the likelihood of identifying the most appropriate group structure to reveal useful trends in past designs. However it was based on an exhaustive clustering method, which could impose a structure with unwanted details on the design examples. There is no easy way to introduce design heuristic knowledge into the learning process to stop at some stage.

4.2.2 Design Synthesis and Evaluation

Inductive learning techniques can be used to support design synthesis and design evaluation tasks. In design synthesis, design examples can be described by *specification properties* and *design properties*. Specification properties are used to generate a classification tree over a set of existing design examples. Concept descriptions, in terms of design properties, can then be used to characterise subsets of examples that are distinguished by their specification properties. A new design is generated by classifying its known specification properties down the classification tree until a description of all the design properties can be retrieved. The inductive learning setting of this approach is illustrated in Figure 4.1.

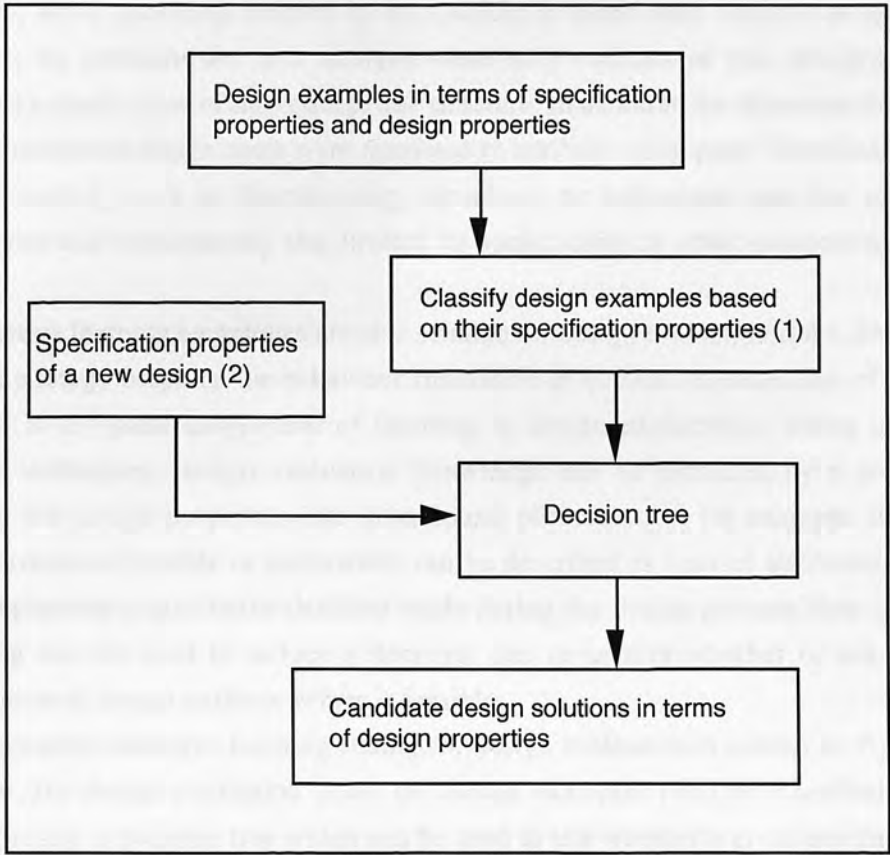


Figure 4.1: Design Synthesis Via Inductive Learning

BRIDGER is a design system developed using an extended COBWEB that can be used to support the design of suspension bridges based on a design concept tree built using the past bridge designs. ECOBWEB extended COBWEB by adjusting its prediction scheme and incorporating continuous properties and constructive induction² [Reich *et al* 1991]. BRIDGER used the same classification scheme for the new design as in ECOBWEB but it used so-called *refinement strategies* to complete the design or retain the abstract design as the solution when the classification process terminates. For example, if the system classified all its specification properties down the classification tree to reach a node where a number of designs were stored, the system could deliver the set of all designs under the class as design candidates; or it could generate a new prototype from the most frequent property-value pairs in a class and deliver it with a set of possible variations; or it could deliver a large set of candidates generated from the combination of all the property-value pairs of designs.

² Constructive induction is a mechanism that groups property-values into higher order features. For example, two complementing values V11 and V12 can be considered into a new feature $G1 = \{V11, V12\}$.

However, three problems existed in BRIDGER: it could only retrieve probabilistic concepts; its concepts for new designs were only variants of past designs, and it supported a single view of the concept tree structure. In addition, the representation of the design classes and design cases were restricted to attribute-value pairs. Knowledge of the design artefact, such as functionality, structure, or behaviour was not explicitly represented and consequently this limited its applicability in other engineering design tasks.

Inductive learning techniques are also suitable for design evaluation tasks. Selecting a redesign strategy based on the behaviour simulation or qualitative evaluation of a design proposal is a typical component of learning in design exploration. Using inductive learning techniques, design evaluation knowledge can be extracted by a process of mapping the design properties into behavioural properties. If, for example, classified design instances (feasible or unfeasible) can be described as a set of attributes, each of which represents a qualitative decision made during the design process, then inductive reasoning can be used to induce a decision tree to predict whether or not a given combination of design attribute values is feasible.

The general inductive learning setting for design evaluation is similar to Figure 4.1. However, for design evaluation tasks, the design examples must be classified and the learning result is decision tree which can be used to test whether a given design belongs to any of the classes. Arciszewski, Mustafa, and Zoarko [Arciszewski *et al* 1987] used a supervised method to differentiate between feasible and unfeasible designs. MacLaughlin and Gero presented a similar approach [MacLaughlin 1987].

4.2.3 Reuse of Past Design Plans

Previous design plans provide useful information for keeping track of the process of solving particular design problems. For example, the order in which design parameters are specified, the ranges of values given to particular design variables, the circumstances and directions in which design parameters are changed, situations where design strategies are changed, can all provide useful information for later analysing how a design problem has been solved.

BOGART is a learning system that automatically replays design plans in the VEXED digital circuit design system [Mostow *et al* 1987]. In VEXED, a circuit design problem is represented as a black box module with input/output specifications and design is carried out using a top-down plan refinement plus constraint propagation approach. Each top-down refinement step decomposes a module into a few interconnected sub-modules, each of which is refined in turn until the entire circuit has been refined into known components such as transistors, gates, standard cells etc. During a design session, the designer makes decisions on how to refine a module, while the system performs the detailed manipulation

and constraint propagation needed to carry out these decisions and deduce their effects.

In the design stage, the VEXED system records the successive refinement steps in a tree-like design plan. A design plan has a node for each module in the circuit. A node represents a step that refined a module into its sub-modules by storing the rule used in the refinement. In the replay stage, BOGART displays a menu of known design plans for the user to decide which of its design plans is to be used. If a design plan is selected, then the user can further decide which of its sub-plans is to be replayed and how far down the tree of the plan is to be replayed. In this way, an abstract version of an original plan can be reused. Once a design plan is selected, BOGART starts the *replay cycle*. BOGART maintains an agenda of unrefined modules that correspond to nodes in the selected design plan, listed in the same order in which the originals were created. BOGART picks the modules in the agenda in turn and refines them, using the rule recorded in the design plan.

BOGART can be seen as a partial design plan replay tool in that, after it replays as much of the requested portion of a design plan as it can, the control mode is returned to VEXED's interactive design cycle. If, at this point, the new design is not completed, i.e., some of the design steps in the design plan do not fit the new design requirements, the designer can continue, either by choosing another design plan to replay for the unspecified modules, selecting an individual rule for VEXED to apply, refining a module by hand, or by backtracking to retract decisions whose results turned out to be unsatisfactory.

However, a difficult situation arises when there is a need to consider how to partially use a previous design; in this case one needs to consider which decisions made in a previous design are to be replayed and which parts of a previous design correspond to which parts of a new problem. BOGART is only suitable for routine design tasks for which design plans exist. It cannot support conceptual design in which the model of an artefact has to be discovered. It can only support the designers to learning or to use the knowledge recorded in past design plans.

An alternative in reusing design knowledge is to reuse the process, not the product of design [Smithers *et al* 1991a and 1991b]. In a knowledge based design support system, if a designer's decision can be recorded in machine understandable form, then it is possible to replay an earlier design decision process, allowing a few changes to original decisions to reflect the need to solve a new design problem. In this approach, gathering design knowledge may be achieved by providing ways of recording, documenting and transferring design results from design documents to the design knowledge base. However, one needs to be concerned with how to capture design decisions at a suitable level of description and how to retrieve a design history given a new design problem.

4.3 Utilisation of Design Heuristics in Learning

Learning in design is not purely a matter of induction. A design system using inductive learning techniques needs not only to abstract class relationships from design examples, but also it needs to make room for modifications by utilising domain specific knowledge. In design, the purpose of learning is to capture knowledge that can be used to create new ideas. An important issue in integrating inductive learning techniques with a knowledge-based design support system architecture is how to make use of both domain knowledge and design heuristics in the learning process. The integration of inductive learning techniques with other AI-based design methods, such as object-oriented representation of design objects and intelligent control of the design process, ensures that the system can provide 'whole solutions' to the design problems.

Learning is much more effective if some *background knowledge* about the domain can be provided [Muggleton 1990]. There are two ways in which design heuristics can be integrated into a learning scheme as *background knowledge*:

- Domain knowledge can be used to control the learning process. For example, COBWEB used a *category utility function*. CLUSTER/S [Stepp *et al* 1986] employed domain knowledge and a *goal dependency network* to identify the most relevant properties in the formation of clusters.
- Design knowledge can be utilised in a learning scheme as *background knowledge* for generalising or specialising concept descriptions. In particular, structured design objects, and the dependencies among them can be used in generalisation and specialisation stages of a learning process. This background knowledge can be incorporated into an inductive learning system as a set of *transformation rules* [Dietterich *et al* 1981].

4.4 The Development of a Design Concept Learning System

To summarise the discussion from section 4.1 to section 4.3 , the importance of learning in knowledge-based design support can be viewed from the following aspects:

- An intelligent design support system should improve its performance over time. Without learning capabilities, a design support system can only respond to situations for which the explicit knowledge has been provided.
- The articulation of design knowledge is labour intensive. A knowledge-based design

support system should be able to learn from examples to produce the knowledge that is more specific to the task at hand, thereby decreasing human effort in design knowledge acquisition.

- Design and learning is an integral part of a creative activity and this activity can be supported by capturing previous design knowledge and then using this knowledge in new designs.

While the systems reviewed in the above sections utilised inductive learning techniques in various design tasks and demonstrated the importance of machine learning techniques for intelligent design support, none of them has addressed the issue of integrating inductive learning techniques with a knowledge-based design support system architecture. It is therefore important to establish a theoretical model for such an integration as well as to develop a software architecture on the basis of this model. It is also important to test such an integrated architecture with sufficiently complex design tasks rather than with simple examples. This section presents a learning model for design support and describes a design concept learning system that can be integrated with a knowledge-based design support system architecture to be presented in chapter 6.

4.4.1 A Learning Model for Design Support

Conceptual design involves identifying a basic structure for the design problem that can be further explored. It is the most crucial stage in the design process where the basic solution to the design problem is discovered. Here this basic structure can be understood as the model of an artefact to be designed. Inductive learning techniques can be used within a computer-based design support system to build such a model, or a number of such models, using past design examples and design heuristics. Without a learning capability, the model of an artefact has to be pre-defined by the knowledge engineers. A pre-defined model of an artefact can have only limited use in a new design as it makes use of past design results alone, and not the process that has led to the specification of the results.

In order to provide a theoretical basis for the development of a knowledge-based design support system architecture in which inductive learning techniques are used to support conceptual design, a learning model is proposed here for a class of design problems for which the initial product data models of the design are not defined or not initially well known. This model involves a three-phase process:

1. learning,

2. designing, and
3. evaluating.

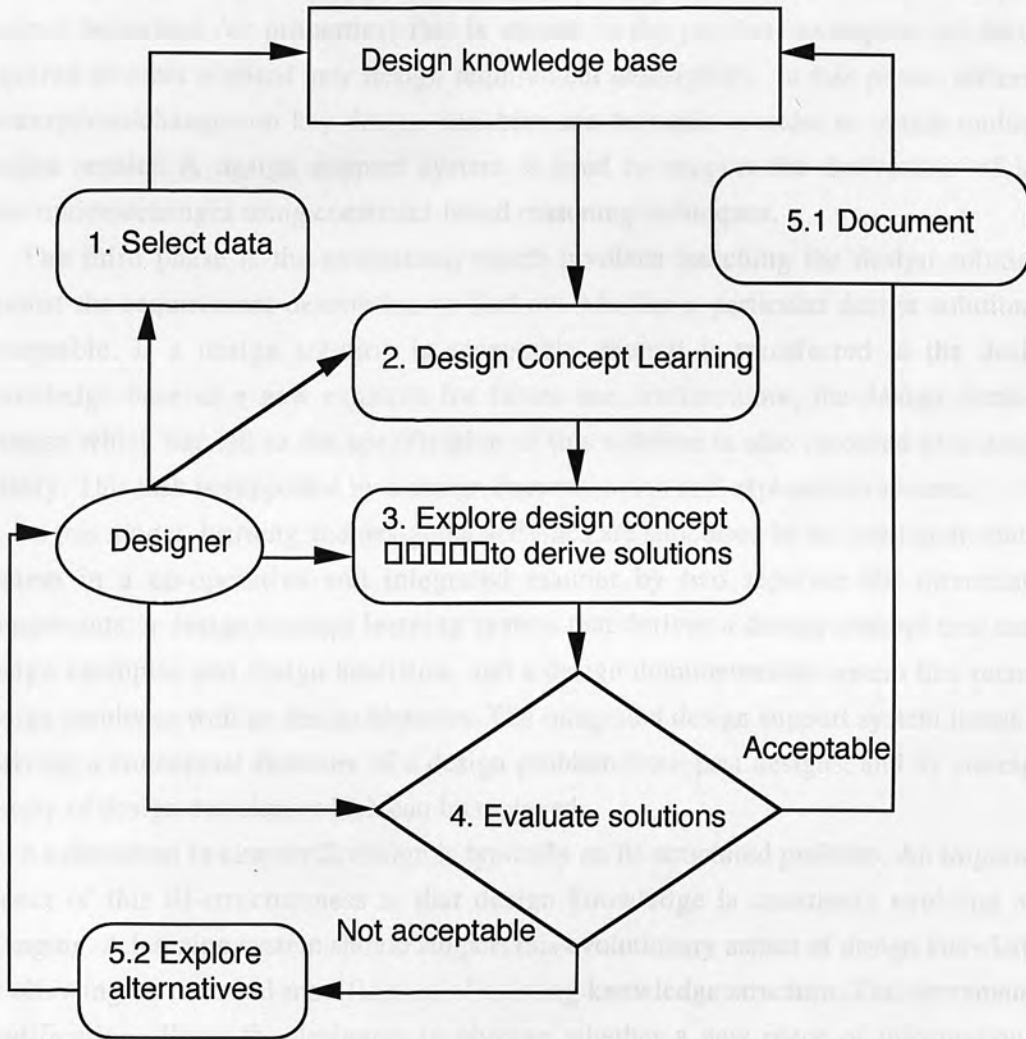


Figure 4.2: A Learning Model for Design Support

The first phase is inductive in nature. In this phase, previous design examples and input data are classified and characterised to such an extent that the structure of the design problem can be identified and constrained. The result of this learning phase is a *design concept tree* that is incrementally developed from design examples and design heuristics. The design heuristic knowledge as the *background knowledge* specifies the levels of concept to be learned whilst the design examples are used to actually build the design concept tree along these concept levels. Each node in this design concept tree represents an important feature demonstrated by the examples classified under it. The design concept tree as a whole provides a well organised knowledge structure in which

implicit knowledge embodied in the original examples is made explicit. From this design concept tree the structure of the design problem can be retrieved and explained.

The second phase is of a more deductive nature: the induced design problem structure is further analysed by making plausible changes to parts of it in an attempt to explore desired behaviour (or properties) that is unseen in the previous examples and that is required to meet a stated new design requirement description. In this phase, different assumptions/changes on key design variables can be made in order to obtain multiple design results. A design support system is used to support the derivations of any assumptions/changes using constraint-based reasoning techniques.

The third phase is the evaluation, which involves matching the design solutions against the requirement description to find out whether a particular design solution is acceptable. If a design solution is acceptable, then it is transferred to the design knowledge base as a new example for future use. Furthermore, the design decision process which has led to the specification of this solution is also recorded as a design history. This task is supported by a design documentation and explanation system.

In this model, learning and designing activities are supported by an intelligent control system in a co-operative and integrated manner by two separate but interrelated components: a design concept learning system that derives a design concept tree using design examples and design heuristics, and a design documentation system that records design results as well as design histories. The integrated design support system learns by deriving a conceptual structure of a design problem from past designs, and by storing a history of design decisions which can be replayed.

As discussed in chapter 2, design is typically an ill-structured problem. An important aspect of this ill-structure is that design knowledge is constantly evolving and changing. A learning system should support this evolutionary aspect of design knowledge by allowing incremental modification of existing knowledge structure. This incremental modification allows the designers to observe whether a new piece of information is related to an existing knowledge structure, or whether the retraction of a piece of information in the existing knowledge structure results in fundamental change. In this aspect, an incremental learning strategy is preferred to a non-incremental strategy.

Both supervised and unsupervised learning can be used to support design applications [Reich *et al* 1993]. Supervised learning is more suitable for supporting design evaluation or rule induction. In this model, inductive learning is used as a way of synthesising the basic design concept structures from input data. Therefore an unsupervised learning approach is adopted to develop a design concept tree.

In this model, a design support system as a constraint-based reasoning tool can also directly manipulate the structure of a design problem or the product data model of an artefact model if it already exists in the design knowledge base, in which case, the learning step can be skipped. This incremental learning model for design is intended for a

class of design problems with the following characteristics:

- when the structure of a design problem is initially unknown, and can only be derived indirectly from past design examples;
- when part of the design objectives is to derive a design problem structure that highlights the common features of the examples (a new design is a modification of this derived structure); and
- when there exists domain knowledge for processing examples, and there are certain criteria for evaluating an adequate description of a design problem structure.

Most design tasks in the conceptual design stage fall into this class of design problem in that in the conceptual design stage, little is known about the design problem and its solutions. It is the most crucial stage in the design process when facts are established, knowledge is gathered, constraints are identified, problems are realised and the solutions are found. Inductive learning techniques play an important role in supporting these tasks by forming organised knowledge structure from different sources (data, domain concepts and design heuristics) that can be explored to derive useful design solutions.

This learning model for design forms the basis for the development of a knowledge-based design support system architecture to be presented in chapter 6.

4.4.2 A Design Concept Learning System

A design concept learning system has been developed by the author as an integral part of a knowledge based design support system architecture based on the learning model for design presented in section 4.4.1.

This design concept learning adopts an unsupervised learning approach for design synthesis tasks and integrates both non-incremental and incremental learning strategies to support the task in the early stage of the design process of developing a concept structure from *unclassified* design examples. It utilises design heuristic knowledge as the *background knowledge* during the learning process. The basic structure of this design concept learning system is illustrated as grey shadowed boxes in Figure 4.3.

The input data is a set of instances representing examples. Each instance has a number of slots representing either single attributes of either nominal or numeric values, or structured instances. The design concept learning system deals with both simple data structures (in the case where all the object instances have single value slots) and complex data structures (in the case where instances are linked to each other).

It integrates three learning strategies for the development of a design concept tree in

which conceptual descriptions of design problem structure can be retrieved and explored further. The basic criteria for evaluating such a design concept tree are:

- It should have some general features that commonly exist in the examples, it should highlight the similarity of design examples in terms of both structure and property.
- It should be well organised for exploring new design solutions.
- It should be incrementally updated.
- It should provide multiple conceptual solutions based on different viewpoints.

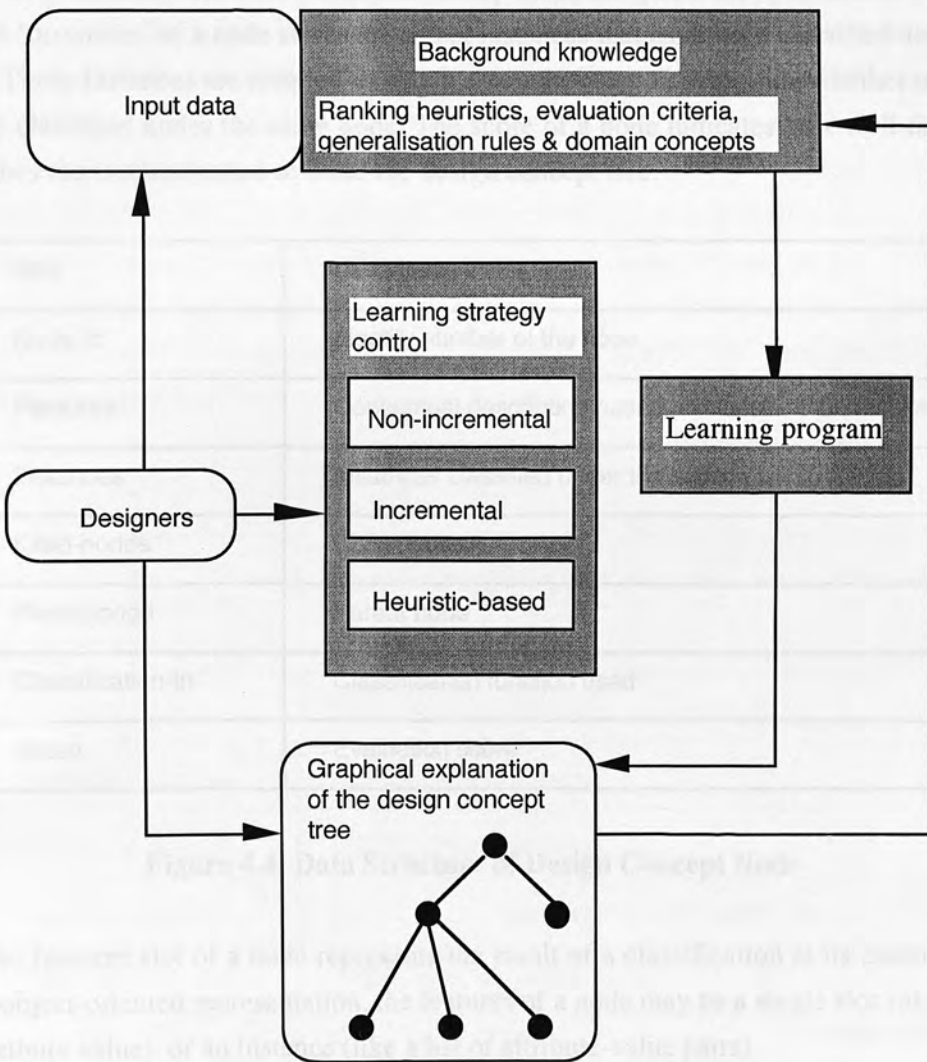


Figure 4.3: Design Concept Learning System

4.4.2.1 Data Structure

A frame-based representation is used to develop the design concept learning system, the task of which is to build a design concept tree from a set of instances representing examples. The data structure of a node in this design concept tree is illustrated in Figure 4.4.

Each node created in the design concept tree has a unique integer identity which is stored in the slot named 'Node-id'. This allows an easy access to all the nodes in the design concept tree and allows each node to be easily identified in a graphical representation of the design concept tree.

Each node in the design concept tree represents a class of features abstracted from a subset of instances, whilst the whole tree incorporates the whole set of instances. The slot named 'Instances' of a node stores the actual instances that have been classified under the node. These instances are recalled to match a new instance to determine whether or not it can be classified under the same node. The score of a node indicates how well the node describes the instances used to build the design concept tree.

Slot	Description
Node-id	Identity number of the node
Features	Conceptual description based on classification results
Instances	Instances classified under the node
Child-nodes	Child nodes
Parent-node	Parent node
Classification-fn	Classification function used
Score	Evaluation score

Figure 4.4: Data Structure of Design Concept Node

The features slot of a node represents the result of a classification at its parent level. In an object-oriented representation, the features of a node may be a single slot value (like an attribute value), or an instance (like a list of attribute-value pairs).

The classification function stored in the slot called *classification-fn* can be either:

1. a function that is pre-set based on the background knowledge (a hierarchical concept

structure that is already known), or

2. a general function that compares the similarity of the instances based on a chosen certain similarity measure.

In the first case, the task of the design concept learning system is to build a hierarchical structure that classifies and characterises a set of unclassified examples given an abstract hierarchical relation. In this case the classification of the design concept tree is both data driven (by instances) and knowledge driven (an abstract hierarchical relation that is used to determine the classification functions used at each level of the design concept tree).

In the second case, the design concept tree is developed using a similarity measure selected by the designers. Three similarity measures have been implemented in this design concept learning system. These are single linkage, complete linkage and group average linkage.

All nodes in the design concept tree are linked through a *parent-node* and *child-nodes* relationship. The result of the learning is a hierarchical structure in which each node has a concept description associated with a subset of instances, as well as a performance evaluation score indicating the quality of that concept description. Nodes higher in the hierarchy represent more general (or more abstract) concepts. The data structure of the design concept tree is illustrated in Figure 4.5.

Slot	Description
All-instances	All instances to be classified in the design concept tree
Current-instance	Current instance
Top-node	Top node in the design concept tree
Similarity-measure	Similarity measure used
Distance-threshold	Distance threshold value
Learning strategy	One of divisive, agglomerative or heuristic approaches
Node-counter	The counter for generating node names

Figure 4.5: Data Structure of Design Concept Tree

The design concept tree stores all the instances to be classified. The slot named

Current-instance stores the instance that is being classified. The design concept tree contains a top node through which all other nodes in the design concept tree can be accessed. Different similarity measures can be specified in the slot named 'Similarity-measure'. A distance-threshold slot stores a threshold value that is to be used by the design concept learning system to decide whether two instances are similar enough to warrant a generalisation. The learning strategy is specified in the slot named 'Learning-strategy'.

4.4.2.2 Learning Strategies

Three different learning strategies are integrated within the implemented design concept learning system:

1. non-incremental approach;
2. incremental and divisive approach; and
3. heuristic-based approach.

The first (non-incremental) approach is suitable for learning from design examples which have numerical attributes. This approach is similar to that of an agglomerative clustering approach but an improved algorithm is used based on an ordered list of distance between pairs of examples. The basic idea is to calculate the distance pairs of examples only once, and order the distance between pairs in a list. The algorithm then merges pairs of examples in this list into concept nodes and stops when all the examples have been covered.

	E1	E2	E3	E4	E5
E1	0				
E2	2	0			
E3	6	5	0		
E4	10	9	4	0	
E5	9	8	5	3	0

Figure 4.6: Distance Matrix

As an example of the operation of this algorithm based on a *single linkage* distance measure, the method is applied to the distance matrix of five examples as illustrated in Figure 4.6. This matrix can be re-organised into an ordered list of distances between pairs as illustrated in Figure 4.7.

Distance	Example pairs	Rank
2	E1 E2	1
3	E4 E5	2
4	E3 E4	3
5	E2 E3	4
5	E3 E5	4
6	E1 E3	5
8	E2 E5	6
9	E2 E4	7
9	E1 E5	7
10	E1 E4	8

Figure 4.7: Ordered List of Distance Pairs

The first pair in the ordered list will be picked and merged into a node (node 1). The second pair in the ordered list is then picked up. Since there is no overlapping between the second pair and the node already created, a second node (node 2) containing E4 and E5 is created. The third pair in the ordered list is then picked up. Because this pair overlaps with node 2, a node will be created to contain the difference between the instances in node 2 and the pair. So the newly created node (node 3) contains E3. Node 2 and node 3 are then merged into a new node 4. By now all the instances have been covered by the current tree, so the process stops. A top node is then created to contain node 4 and node 1. The result of clustering is shown in Figure 4.8.

This algorithm is suitable for a Lisp execution because it is based on *List* operations. It is more efficient than a matrix-based agglomerative algorithm described in [Everitt 1981]. In order to test this, the same example set as shown in Figure 4:6 was presented to the above discussed algorithm and the standard agglomerative algorithm described in chapter 3. This test showed that the running time of this lisp-based algorithm

is two thirds of that of the standard agglomerative algorithm and both algorithms generated the same results shown in Figure 4.8. This means that in an application that involves a large number of examples, the lisp-based algorithm can have an advantage in terms of running speed.

The second (incremental) approach builds a design concept tree in a divisive manner similar to that of UNIMEM. It starts the learning process by creating a top node containing the first instance from a ranked training set. When a new example is supplied, the system searches the current design concept tree in a *depth-first* manner for the most specific node under which the new example can be classified (again this is determined using a specified distance measure). If no such node is found the new example is simply stored at the top node. Otherwise the examples stored in the most specific are retrieved and compared with the new one. If any of these examples is considered to be similar enough to the new one, according to the pre-defined criteria (a threshold value, for example), then a generalisation takes place. A generalisation involves creating a new child node using the two examples that are considered to be similar enough to make the generalisation. If no generalisation can be made, the new example is simply stored in the node with other examples already in the system.

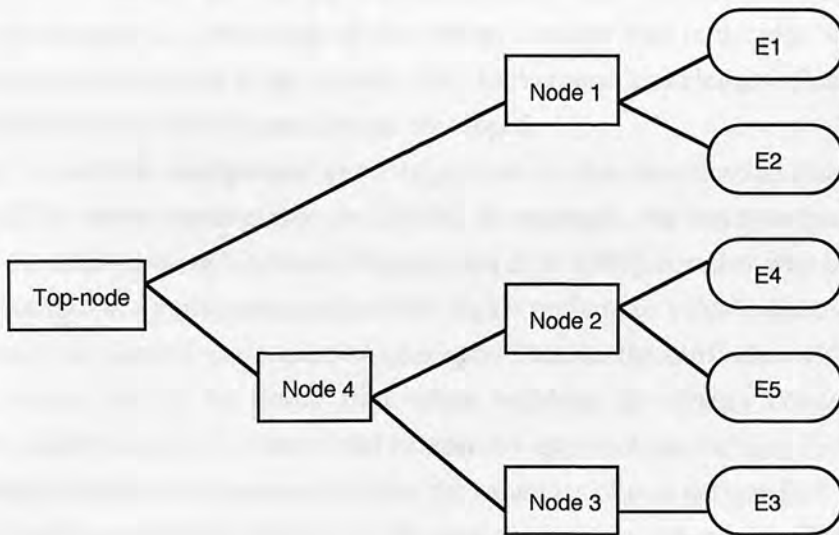


Figure 4.8: Clustering Results

However, in this approach, a threshold value is needed for the system to determine whether two examples are similar enough to warrant a generalisation. To demonstrate this, the same example with the distance matrix shown in Figure 4.6 is used here. In this divisive approach, when the first example, E1, is presented to the system, a top node is created which stores E1. When the second example, E2, is presented the distance between E1 and E2 is calculated. Suppose the threshold value for generalisation is 5. Because the

distance between E1 and E1 is 2, a generalised node containing E1 and E2 (node 1) is created under the top node.

When the third example E3 is presented, it cannot be classified into node 1 because its distance to neither of E1 and E2 is smaller than the distance between E1 and E2. Therefore E3 is stored at the top node. When example E4 is presented, it cannot be classified under node 1 because its distance to E1 (10) and E2 (9) are greater than the distance between E1 and E2 (2). However, its distance to E3 stored at the top node is 4 (smaller than the generalisation threshold value 5), so a generalised node containing E3 and E4 (node 2) is created under the top node.

When the last example E5 is presented it is first classified under node 2 because its distances to E3 (5) and E4 (4) are equal to or less than the generalisation threshold value. Furthermore, because E5 is more similar to E4 than to E3, a further generalisation takes place, creating a sub-node containing E4 and E5 (node 3) and a sub-node containing E3 (node 4) under node 2. This produces the same result as that of an agglomerative approach shown in Figure 4.8 with the exception that the nodes are numbered differently because of its divisive nature.

The third (heuristic-based) approach uses design heuristic knowledge as background knowledge to sort instances through a pre-defined conceptual structure. In this approach, the levels of concept, i.e., the depth of the design concept tree is decided using the domain concepts already held in the system. This background knowledge influences the way in which the design concept tree is to be developed.

One way to use this background knowledge is to fix the classification functions at each level of the design concept tree. In EPAM, for example, the test function at each level of the discrimination net is fixed [Feigenbaum *et al* 1984]. Another way is to give each slot of an instance a preference value with higher preference value indicating that it should be used to classify more general concepts. That is, the attributes with higher preference values are to be tested first when building the design concept tree. Alternatively, Duffy and Kerr's customised perspective approach can be used for the user to decide which attribute is to be used to group the examples of past designs first and then decide which of the resulting groups is to be used to generate sub-groups [Duffy *et al* 1993].

The heuristic-based learning approach has been used in the domain of small-molecule drug design because domain knowledge can be utilised as *background knowledge*. An example will be presented in chapter 7.

4.4.2.3 Generalisation Rules

When a new instance is classified under a node the features of the node need to be generalised. The rule that can be used to generalise a numerical attribute is called the

closing interval rule [Dietterich *et al* 1981]. This rule states that the expression [$\mathbf{a} = a_x$, $a_1 < a_x < a_n$] is less general than the expression [$\mathbf{a} = a_1, \dots, \text{or } a_n$], where \mathbf{a} is an attribute. This rule allows a generalisation to occur within the interval between a_1 and a_n .

The second rule deals with attributes with yes/no (or 1/0) values. This rule is used only in situations where yes/no values can be logically OR-combined to indicate the existence of a particular property among a number of instances. For example, if a property has yes/no values then this rule can be used to answer the question 'do any of the instances have this property at all?'. This rule can be considered as a domain dependent one and it means that if the value of a property in any of the instances is yes, then the result of generalisation is yes.

4.4.2.4 Implementation

The above described design concept learning system has been implemented using an object-oriented representation within a Lisp-based environment. In this implementation, the user can choose one of three learning strategies discussed in Section 4.5.2. For each of these strategies, three different similarity measures, i.e., single linkage, complete linkage and group average linkage have been implemented. These three linkages are described in section 3.2.4 in chapter 3.

A control panel is used in a graphical user interface for the user to choose, add or delete instances. A graphical display window at the right hand side of the screen layout of the graphical user interface displays the structure of the design concept tree as it is being constructed. The user can click on any node in the design concept tree to view the concept description and instances associated with it. The software of the design concept learning system is described in Appendix B.

The design concept learning system deals with both simple data structures (in the case where all the object instances have single value slots) and complex data structures (in the case where instances are linked to each other). The result of the learning is a hierarchically structured design concept tree in which each node has a concept description associated with a subset of instances, as well as a performance evaluation score indicating the quality of that concept description. Nodes higher in the hierarchy represent more general (or more abstract) concepts. The learning process is graphically displayed as the design concept tree is being built.

The design concept learning system is intended to support design tasks. Three different learning strategies are therefore integrated within the implemented design concept learning system for these purposes: 1) a non-incremental approach; 2) an incremental and divisive approach; and 3) a heuristic-based approach. The first two strategies utilise existing clustering techniques [Everitt 1981 and Lebowitz 1987]. The third strategy is developed by the author to address conceptual design problems. The

integration of these three different learning strategies provides a flexible computational environment for testing the issue of integrating inductive learning techniques in knowledge-based design support systems.

The first strategy adopts an improved agglomerative algorithm to cluster design examples into the design concept tree. In this approach, attributes of examples can be selected to form clusters, each of which can be classified again using another set of attributes, thus forming the so-called nested clusters reflecting designers' views similar to the multi-perspective approach developed by Duffy and Kerr [Duffy *et al* 1993]. The second strategy builds a design concept tree in an incremental and divisive approach so that examples can be added to or deleted from the design concept tree, thus allowing the designers to observe the influence of particular examples to the features represented by the design concept tree; The third (heuristic-based) approach uses design heuristic knowledge as background knowledge to sort design examples through a pre-defined or user specified conceptual structure. In this approach, the levels of concept represented by the design concept tree is influenced using the domain concepts and their relations already held in the system as *background knowledge*, thus providing a way for assessing the relevance of a set of design examples to an existing design concept already known to the system or a design concept suggested by a designer.

Summary

This chapter has reviewed a number of design systems utilising inductive learning techniques and discussed the ways in which inductive inference can be utilised in a knowledge-based design support system architecture to support:

1. the acquisition and formulation of design knowledge in the conceptual design stage;
2. the development of a design concept learning system that supports conceptual design tasks; and
3. the accumulation of design knowledge through maintaining design history.

A design concept learning system has been implemented in a Lisp environment utilising three different learning strategies in a unified representation and framework. This design concept learning system is intended to be used as an integral part of a knowledge-based design system architecture. This architecture is different from the systems that have been reviewed in this chapter and in chapter 2 in that it is an integrated system that provides a knowledge-based framework within which a design concept learning system is used as a part of the system's support to conceptual design. It also

provides other facilities to support the exploration and maintenance of multiple contexts of design. It uses a design documentation system to record the design history that can be replayed.

By way of supporting the derivation of useful conceptual design solutions, the exploration and management of these conceptual design solutions, and the documentation of the design results as well as the history of the exploration process, this architecture provides computer-based support to the above three tasks within one integrated computer environment. The design concept learning system provides support for the first and second tasks while a design documentation system provides support to the third task.

The application and evaluation of this design concept learning system and the architecture within and beyond the domain of small molecule drug design will be discussed in chapter 6, 7 and 8.

The rational drug design approach is to design a drug based upon an understanding of the relationship between the structure and the pharmacological activity of the drug at the molecular level. This approach is termed *rational drug discovery* and is based upon the idea that the pharmacological activity of a drug is a direct consequence of its binding to a target receptor molecule. When a drug binds to a receptor, some biological change takes place as shown in Figure 5.1.

Although there are still no theoretical methods for designing a drug from first principles (King *et al* 1993 and Keefe *et al* 1991), the use of computer-based technologies in drug design has become an important factor for the success in the rational drug design approach because this approach involves analysing the complex chemical and biological relationships between the structures and pharmacological activities of large numbers of molecules (Hunter 1993 and Huang *et al* 1993).

Computer-based design support techniques need to be tested in real design applications. The problem in drug design is used in this thesis to test the knowledge-based design support system architecture and the design concept learning system. This requires a detailed analysis of the nature and the complexity of the design tasks in this domain, and an explanation of why knowledge based design techniques, including inductive learning techniques, can be used to support drug design.

In this chapter, the basic concepts in drug design, the analysis of drug design tasks, and the representation drug design knowledge are presented first. This gives a background for describing how the architecture as presented in chapter 6 can be used to support drug design tasks, and how the design concept learning system described in chapter 4 can support a key task in drug design, i.e., the generation of a pharmacophore description from molecule examples (the concept of a pharmacophore and a pharmacophore description are to be described in Section 5.2).

Chapter 5

Knowledge-Based Support of Drug Design

Traditional drug discovery has relied upon the serendipitous method of screening a large set of existing candidate substances to look for those with the desired biological effect. Modifications are made to molecules that pass the screening process in order to increase desirable properties and decrease undesirable properties. The new molecules are then tested and evaluated again. This cycle of modifying and testing is repeated several times until a molecule that satisfies the drug design requirement is obtained.

This screening method is, however, increasingly seen by the drug designers as a waste of research resources [Floyd 1990]. Consequently, researchers have been searching for more rational techniques that allow drugs to be designed with a particular target in mind.

The rational drug design approach is to design a drug based upon an understanding of the relationship between the structure and the pharmacological activity of the drug at the molecular level. This approach is termed *rational drug discovery* and is based upon the idea that the pharmacological activity of a drug is a direct consequence of its binding to a target receptor molecule. When a drug binds to a receptor, some biological change takes place as shown in Figure 5.1.

Although there are still no theoretical methods for designing a drug from first principles [King *et al* 1993 and Koile *et al* 1991], the use of computer-based technologies in drug design has become an important factor for the success in the rational drug design approach because this approach involves analysing the complex chemical and biological relationships between the structures and pharmacological activities of large numbers of molecules [Hunter 1993 and Huang *et al* 1993].

Computer-based design support techniques need to be tested in real design applications. The problem in drug design is used in this thesis to test the knowledge-based design support system architecture and the design concept learning system. This requires a detailed analysis of the nature and the complexity of the design tasks in this domain, and an explanation of why knowledge-based design techniques, including inductive learning techniques, can be used to support drug design.

In this chapter, the basic concepts in drug design, the analysis of drug design tasks, and the representation drug design knowledge are presented first. This gives a background for describing how the architecture to be presented in chapter 6 can be used to support drug design tasks, and how the design concept learning system described in chapter 4 can support a key task in drug design, i.e., the generation of a pharmacophore description from molecule examples (the concept of a molecule and a pharmacophore description are to be described in Section 5.2).

5.1 The Metaphor of Lock and Keys

Drugs that 'fit' better to a receptor will bind more strongly, which will improve pharmacological activity. The popular metaphor of *lock-and-key* can be used here: the drug '*key*' is specifically designed to fit and to operate the receptor '*lock*'.

In rational drug design, for example, if the molecular structure and geometry of a receptor is known, for example, through X-ray crystallography, then the drug designers can understand the receptor's requirements explicitly, and are therefore able to use the so-called *direct approach* to design molecules that fit the requirements. This approach, however, can hardly be adopted by the majority of drug designers because:

- the geometry of most pharmacologically interesting receptors has not been characterised at an atomic level; and
- the process of obtaining the molecular structure for a single receptor type may take many years [Floyd 1990 and Hodgkin 1991].

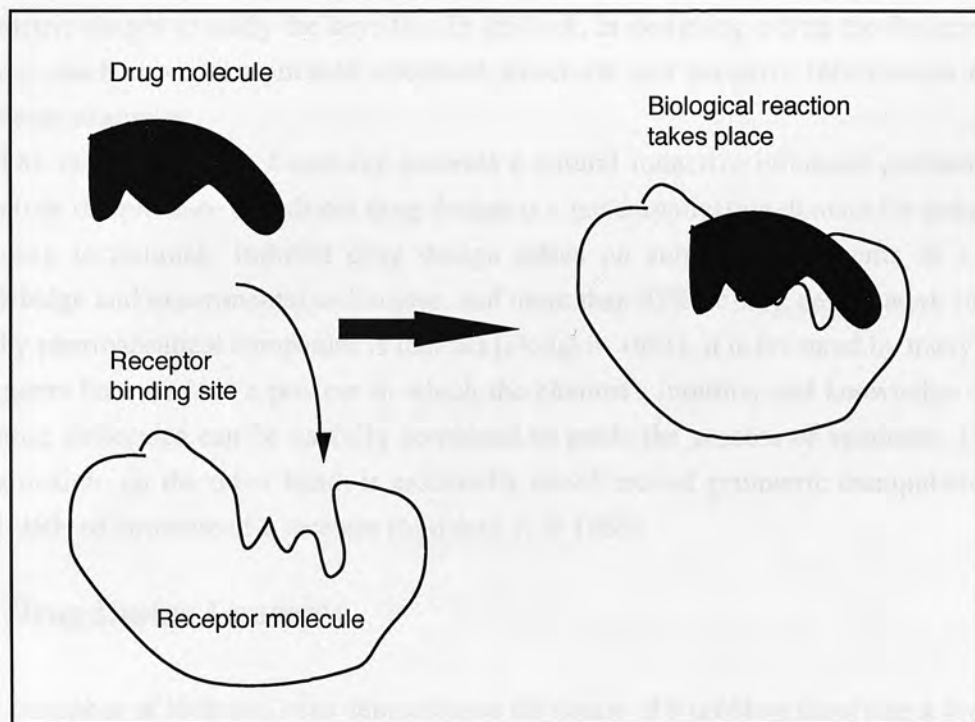


Figure 5.1: Molecule Binding

This situation is like designing a key to turn a lock, without knowing the shape of the lock inside. Imagine a designer is given a lock (without the information of its inner structure), and a set of keys that partially turn the lock to some varying degrees, and is

asked to design a key which can better operate the lock. A natural approach would be to analyse the common features of the keys first. Having figured out what basic key shape contributes to partially turning the lock, the designer might then want to make a new key by modifying these basic shapes to turn the lock more effectively (not just partially).

Drug designers have similar problems to the lock-and key situation, i.e., designing a drug (key) without necessarily the detailed knowledge of an acceptor (lock). The majority of drug designers often have to take a so-called *indirect approach* to drug design [Hunter 1993 and Marshall *et al* 1986].

In indirect drug design, the nature and size of a receptor binding-site often have to be inferred from the molecules that the receptor mostly readily accepts. This approach is analogous to attempting to infer the configuration of the inside of a lock from an examination of the keys that best fit it. It is a process of identifying a structure from examples, and then improving that structure by making conservative changes. This structure is a description of required receptor binding features called the *pharmacophore*. Generating a pharmacophore from a set of existing molecules is one of the major tasks in indirect drug design.

While in the lock-and-key problem a designer might only need to infer about geometric shapes to study the keys that fit the lock, in designing a drug the designer has to use much more complicated chemical structural and property information about molecule examples.

The metaphor of *lock-and-key* presents a natural inductive inference problem and therefore the problem of indirect drug design is a good application domain for inductive learning techniques. Indirect drug design relies on substantial amounts of expert knowledge and experimental techniques, and more than 95% of drug design work carried out by pharmaceutical companies is indirect [Hodgkin 1991]. It is favoured by many drug designers because it is a process in which the chemist's intuition and knowledge about existing molecules can be usefully combined to guide the process of synthesis. Direct drug design, on the other hand, is essentially based around geometric manipulation of well-defined structure of a receptor [Marshall *et al* 1986].

5.2 Drug Design Concepts

The metaphor of lock-and keys demonstrates the nature of a problem involving a form of inductive inference, abstracting features from known examples. However, it is necessary to explain the concepts in this domain in more detail before identifying what computer-based support can be provided to support the tasks in this domain.

In indirect drug design, molecules are used as examples to derive a pharmacophore that highlights their binding features to a particular receptor. Each molecule can be viewed by drug designers at different levels from different views. In a computer-based

system, these levels and views need to be properly represented before any inferences can be made. This section describes and explains the concepts in this domain and their representations.

5.2.1 Molecules

A molecule is represented by its structure and some other attributes such as its *activity*, *assay type* and its structure in terms of atom connectivity. A molecule is traditionally represented as a two-dimensional structure as shown on the left hand side in Figure 5.2. The right hand side in Figure 5.2 shows the attributes in bold font of a molecule.

A computational representation called a *Smiles string* has become a major way of representing molecular structures in a computer system. A Smiles string denotes the structure of a molecule structure using a string [Weininger 1989], and it represents the two-dimensional structure of a molecule in a convenient and reliable way which enables it to be easily manipulated by a computer program.

For example, the Smiles string of the molecule shown in Figure 5.2 is "COc1ccc(COCC(O)Cc2ccnc2)cc1". A Smiles string also shows the substructures such as *rings*, *branches* or *chains* of a molecule. There is a set of rules which determines how Smiles string notation can be used to depict the two-dimensional picture of a molecule. The rules for representing molecular structures using the notion of a Smile string are omitted here but given in Appendix A.

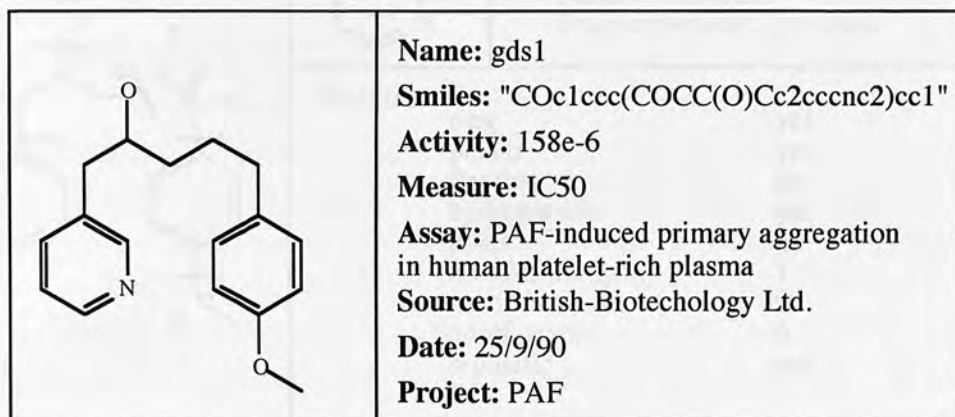


Figure 5.2: Structure and Attributes of a Molecule

5.2.2 Molecular Component

A molecular component is the smallest unit of a chemical structure containing *recognisable chemical functions*. These functions are represented using self-contained

units with a physical and chemical identity and properties, such as *numbers-of-atoms*, *planar*, *polar*, *flexibility*, *hydrophobic*, *ring*, *no-of-h-acceptors*, *no-of-h-donors*, *aromatic*. Partitioning a molecule into constituent molecular components to derive their physical and chemical properties is essentially how drug designers deal with the chemical structures within the molecule. The properties of a molecule are largely derivable from the properties of the molecular components joined together to form the molecule.

A set of rules exists for partitioning a molecule into molecular components and these rules are domain independent and applicable to any organic structure. This task can be supported by a computer system using a rule-based approach by recognising the so-called *isolating bonds* which mark the boundaries between atoms of different classes.

Figure 5.3 illustrates the molecular components of a molecule partitioned by isolating bonds. In Figure 5.3, the left hand side shows the partition and the right hand side illustrates a particular component called *pyridine* with its attributes and values. For example, this component is number (1), it is connected to another molecular component (2), its Smiles string is "c1cccn1" and its property model class has pyridine. In molecular biology, the properties of molecular components are classified by their property models.

The actual rules obtained from drug design experts for partitioning molecules into molecular components are explained in Appendix A.

<p>— Isolating bond</p>		<p>Component number: 1 Connected_to: 2 Smiles: "c1cccn1" Property-model: pyridine</p>																		
	<p>Pyridine</p>	<table> <tr><td>ring</td><td>yes</td></tr> <tr><td>planar</td><td>yes</td></tr> <tr><td>flexible</td><td>no</td></tr> <tr><td>hydrophobic</td><td>yes</td></tr> <tr><td>polar</td><td>no</td></tr> <tr><td>no-of-h-acceptors</td><td>1</td></tr> <tr><td>no-of-h-donors</td><td>0</td></tr> <tr><td>no-of-atoms</td><td>6</td></tr> <tr><td>aromatic</td><td>yes</td></tr> </table>	ring	yes	planar	yes	flexible	no	hydrophobic	yes	polar	no	no-of-h-acceptors	1	no-of-h-donors	0	no-of-atoms	6	aromatic	yes
ring	yes																			
planar	yes																			
flexible	no																			
hydrophobic	yes																			
polar	no																			
no-of-h-acceptors	1																			
no-of-h-donors	0																			
no-of-atoms	6																			
aromatic	yes																			

Figure 5.3: Component Partition

5.2.3 Molecular Fragment

A molecular component has identifiable physical and chemical properties. However, it is not necessarily the basic unit for describing how a molecule binds to a receptor. The representation of a *molecular fragment* was introduced by the drug designers involved in

the Castlemaine project in order to represent the part of a molecule that describes the molecule's binding features.

A molecular fragment is *a small collection of molecular components recognisable by a receptor in a co-operative manner*. A fragment has a primary property as the major receptor binding feature, and a number of secondary properties describing the fragment's remaining chemical properties.

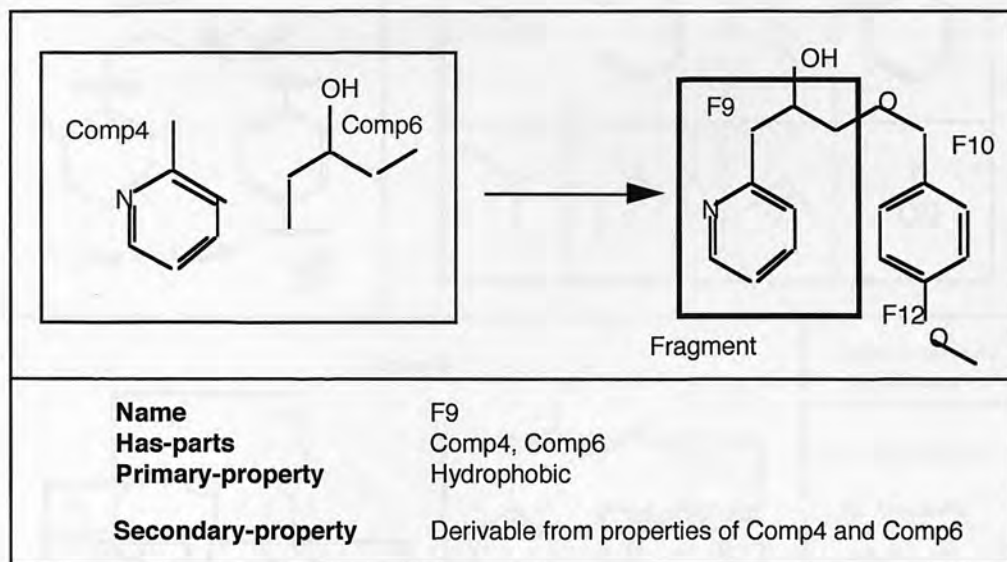


Figure 5.4: Molecular Fragment

For example, the two molecular components (comp4 and comp6) in Figure 5.4 can be combined to form one molecular fragment (F9). This molecular fragment demonstrates a *hydrophobic* primary property among other secondary properties when binding to a target receptor in a co-operative manner. By co-operative manner, is meant that it is advantageous to combine proximal molecular components that are close but not necessarily directly connected in the molecule because they may act jointly at the receptor [Hodgkin 1991]. Two classes of molecular fragments can be formed based on this assumption: 'contiguous' fragments where all the constituent molecular fragments are directly connected and 'disjoint' fragments where one of the constituent molecular components is not directly connected to the rest of the components. This means that molecular fragments may overlap within a molecule, and that one molecular component may contribute to more than one fragment.

Not every set of components within a molecule makes a legal fragment. For example, 12 molecular fragments can be derived from a molecule with 8 molecular components as illustrated in Figure 5.5. Figure 5.5.a shows an original molecule with all its isolating bonds identified using the rules given in Appendix A. Figure 5.5.b lists all the 8

components that are separated by those isolating bonds. Figure 5.5.c illustrates all the fragments that can be assembled from these 8 components. In Figure 5.5.c, all the fragments are illustrated with closed thick lines and named (such as F1, F2 etc.). The primary properties of the fragments are listed on the right hand side.

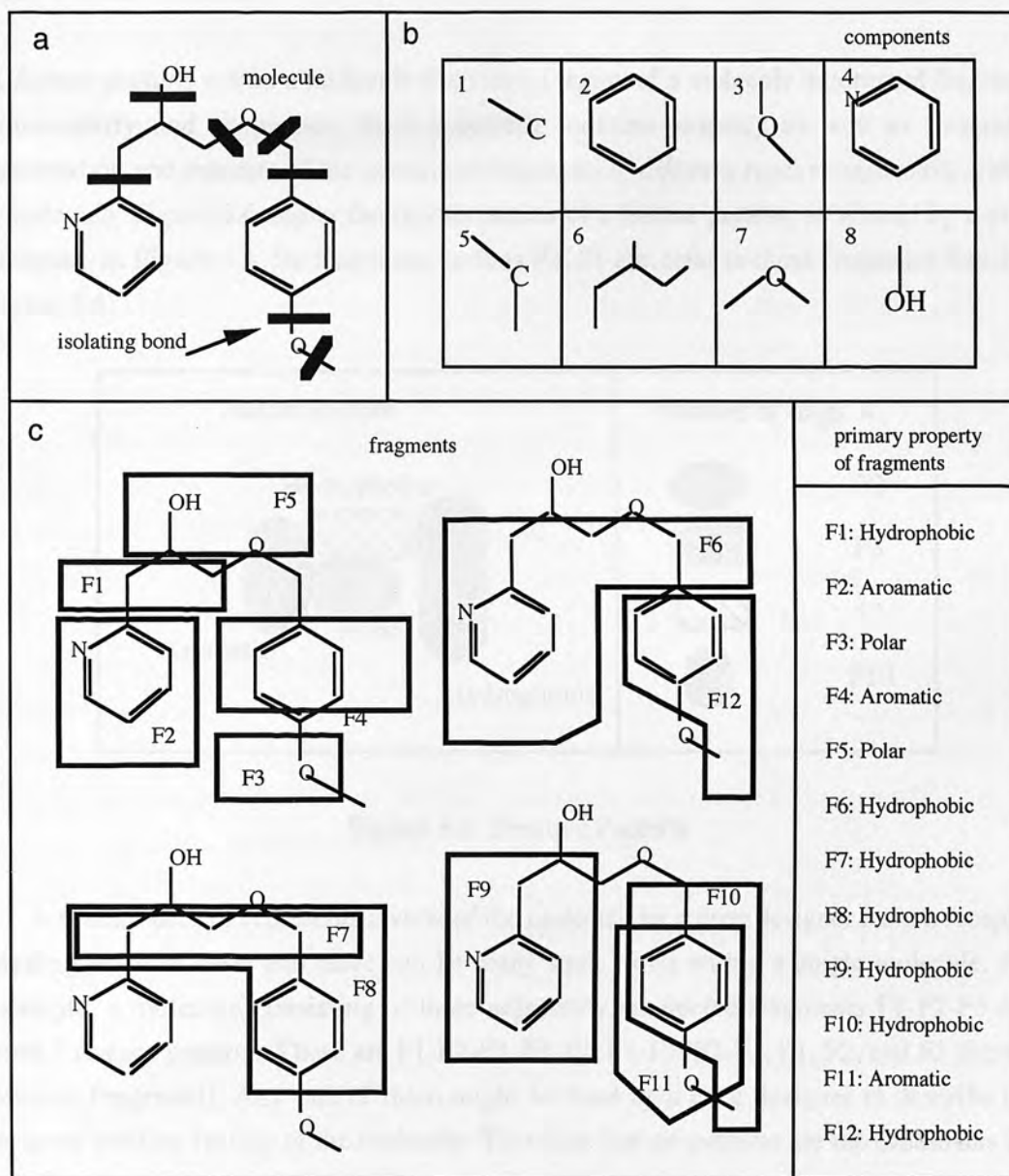


Figure 5.5: From Molecule to Fragments

Rules have been provided by the drug designers in the domain to assemble molecular fragments from molecular components [Hodgkin 1991]. Molecular fragments are formed from one or more molecular components, with each component typically contributing to more than one fragment. In general, molecular components with similar properties may

combine to form molecular fragments. Molecular fragment assembly occurs by considering each molecular component in turn to be a seed to which other components can be added according to a set of rules. These rules are explained in Appendix A.

5.2.4 Feature Pattern

A *feature pattern* within a molecule describes a region of a molecule in terms of fragment connectivity and properties. Such a pattern contains property as well as structural information and consists of the *connected fragments of different types* recognisable within a molecule. Figure 5.6 shows the representation of a feature pattern, as viewed by a drug designer. In Figure 5.6, the fragments such as F2, F1 etc. refer to those fragments listed in Figure 5.5.

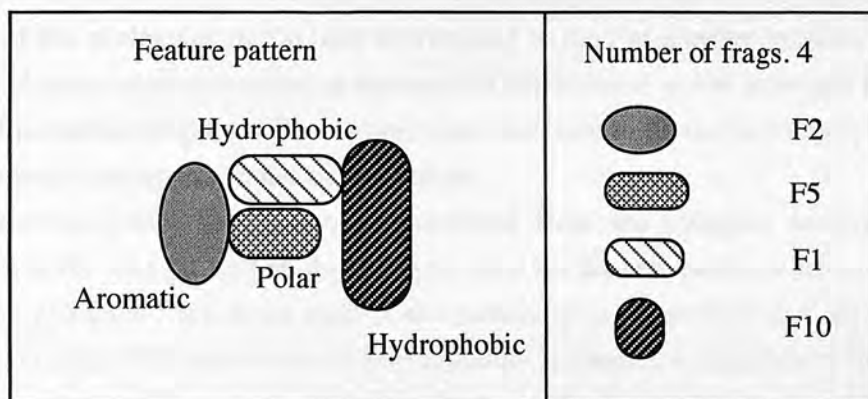


Figure 5.6: Feature Pattern

A feature pattern represents a view of the molecule by a drug designer from a receptor binding point of view and there can be many such views within a single molecule. For example, a molecule consisting of three adjacently connected fragments F1-F2-F3 can form 7 feature patterns. These are F1-F2-F3, F1-F2, F1-F3, F2-F3, F1, F2, and F3 (here F denotes Fragment). Any one of them might be used by a drug designer to describe the receptor binding feature of the molecule. Therefore feature patterns are the candidates for generating a pharmacophore description.

A feature pattern can be described by the primary properties of its fragments. If, for example, F1 has a primary property of polar (P) and F2 has a primary property of hydrophobic (H), then the feature pattern of F1-F2 can be classified as having a *general pattern* of P-H, which conceptually means that 'it is a pattern with a polar fragment on the left and a hydrophobic fragment on the right'. Here F1-F2 is regarded as a specific pattern while P-H is regarded as a general pattern (general in terms of primary property). If, for example, there is another feature pattern, say, F3-F4 with the primary property of F3

being *polar* and the primary property of F4 being *hydrophobic*, then F3-F4 can be considered as being in the same class as F1-F2 because they both have the same general pattern, i.e., P-H. The only difference between the two is the ranges of the secondary properties of their fragments.

5.2.5 Pharmacophore Description

A selected set of molecules used for deriving a pharmacophore description is called a *lead set*. It is defined as a set of molecules which are regarded by the drug designer as highly relevant to the drug being designed.

While a feature pattern is a region within a single molecule described by fragment connectivity and properties, a pharmacophore description is a feature pattern common to all or most of the molecules in a lead set. It is the complete assembly of properties of all or most of the molecules in the lead set required to bind to a target receptor. In other words, a pharmacophore description represents a hypothetical model about the shape and chemical properties required of an abstract new molecule to fit the target receptor site in the same way as those molecules in the lead set.

A pharmacophore description is abstracted from the common features of the molecules in the lead set, and used as a requirement for the new molecule being designed. It must be extracted from all or most of the molecules in the lead set that are known to bind to a receptor. The basic idea of pharmacophore generation is therefore to find out the similarities among all or most of the molecules in the lead set in terms of important receptor binding features, and to describe them in a way in which these features can be easily retrieved and further explored for design purposes.

A molecule has multiple feature patterns as a result of fragments overlapping (see Figure 5.5). Therefore many candidate pharmacophore descriptions may be derived from one lead set, but only those which are abstracted from all or most of the molecules are regarded as being useful.

As illustrated in Figure 5.7, a pharmacophore description contains the following information:

- the feature pattern of the pharmacophore;
- the number of features it has;
- the primary properties of those features;
- the fragments which compose the pharmacophore description; and

- the constitution of the fragments in the feature pattern.

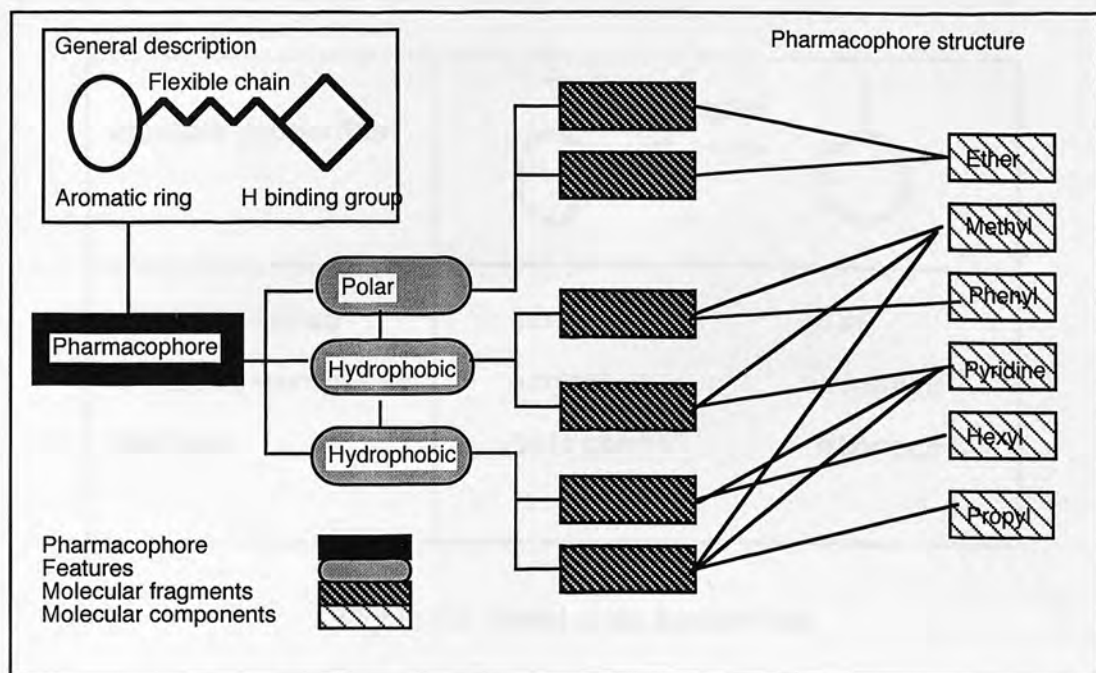


Figure 5.7: Pharmacophore description

Deriving a pharmacophore description from a lead set is the main task in the indirect drug design process that can benefit from the support of inductive learning techniques. This will be fully described in chapter 7.

5.2.6 Isosteres

Isosteres are molecular fragments which have similar chemical characteristics but different structures. When an isostere is looked upon as one of a pair of isosteres, an important assumption can be made: that is, it is possible to substitute one structure for another whilst retaining a given profile of desired properties [Hodgkin 1991]. When part of a molecule is substituted by an isostere, the overall activity of the molecule is likely to change. An example of an isostere pair is illustrated in Figure 5.8. The two fragments that form this isostere pair have the same primary property but slightly different structure and secondary properties.

Specifying isostere replacements is a major way of improving a drug. In doing so a drug designer needs to have a clear notion of which properties are important for binding and in which context those properties could be relevant. This is helped by the pharmacophore description derived from the molecules in a lead set. Pre-stored isostere

pairs in a library can be searched by a computer system for an isostere pair to modify the specification of a new molecule.


Isostere properties		
Part-of-drug	gds5	gds6
Primary-property	aromatic	aromatic
Smiles	<code>"c1ccncc1"</code>	<code>"c1ccccc1"</code>

Figure 5.8: Model of the Isostere Pair

Isostere replacement is done by conservatively replacing component structures within a selected molecule with isosteres, i.e., different components which have similar chemical characteristics. By "conservatively", is meant that there are gradations in the extent of change in making an isosteric replacement [Hodgkin 1991]. An isosteric replacement is exemplified by a pair of molecules and is identified as such when a number of criteria are met:

- when the molecules differ by a single structural change;
- when the molecules are of similar activity; and
- when the molecules show good activities.

The simplest isosteric replacements are univalent single atoms. These are conservative substitutions with a high probability of not destroying the molecule's activity but a low probability of making a significant improvement to the activity. These can be characterised as *low-risk-low-gain* substitutions [Thornber 1979]. The simpler modifications are quick and easy to do. With a small substitution, it is possible to attribute the effects of the change more precisely. Identification of isosteres in a series of molecules can take place once the pharmacophore description has been generated. In a computer-based drug design system, a number of well-known isosteres taken from

relevant literature can be used to build a library to support the task of isostere replacement.

5.3 Supporting Drug Design Using an Inductive Learning Approach

A drug design project starts with a generally stated goal, i.e., to design a new molecule which binds to a particular target receptor, or to design a new molecule which has the similar but improved biological effect to that of a series of existing molecules. To identify a pharmacophore from a set of molecules is a key task in the indirect drug design approach. It can be supported using inductive learning techniques because it involves an induction of common features observed from molecule examples.

From a design point of view, to discover a pharmacophore description is to define an initial design problem structure within which key design parameters and features are defined and constrained first. The design task can then be focused on modifying parts of a molecule, using the pharmacophore description, to improve its pharmacological activity.

In the absence of a detailed receptor structure, it is only possible to use a pharmacophore description by analysing the features of a number of existing molecules which have similar biological behaviours to the drug being designed [Smithers *et al* 1993a, 1993b]. The derivation of a pharmacophore description from a lead set of molecules can be seen as a learning problem. Here the task of learning is to identify and structure the most common and the most characteristic features of the given molecules. A feature of this learning task is that it must utilise the existing domain concepts and knowledge to derive a pharmacophore description that is meaningful for the design of a new drug. In other words, it should utilise background knowledge during the course of learning.

The design of a new drug is modelled as a process involving two separate but closely-related activities: the inductive learning of the structure of a design problem (pharmacophore description); and the exploration of the design problem structure to obtain new solutions (specification of novel molecules through isostere replacement). Figure 5.9 illustrates a mapping from the incremental learning model discussed in chapter 4 through to the small-molecule drug design process.

To complete the whole loop of the indirect drug design process based on this model, AI-based techniques are needed for the following three different tasks:

1. the pre-processing task that transfers original molecules into object instances suitable for the design concept learning system;
2. the learning task that derives a pharmacophore description from a set of molecule examples; and

3. the design task that identifies a new molecule based on the pharmacophore description using the isostere replacement method.

The application of the design concept learning system discussed in chapter 4 for the first two tasks is to be presented in chapter 7. The issue of supporting the third task is discussed in chapter 8.

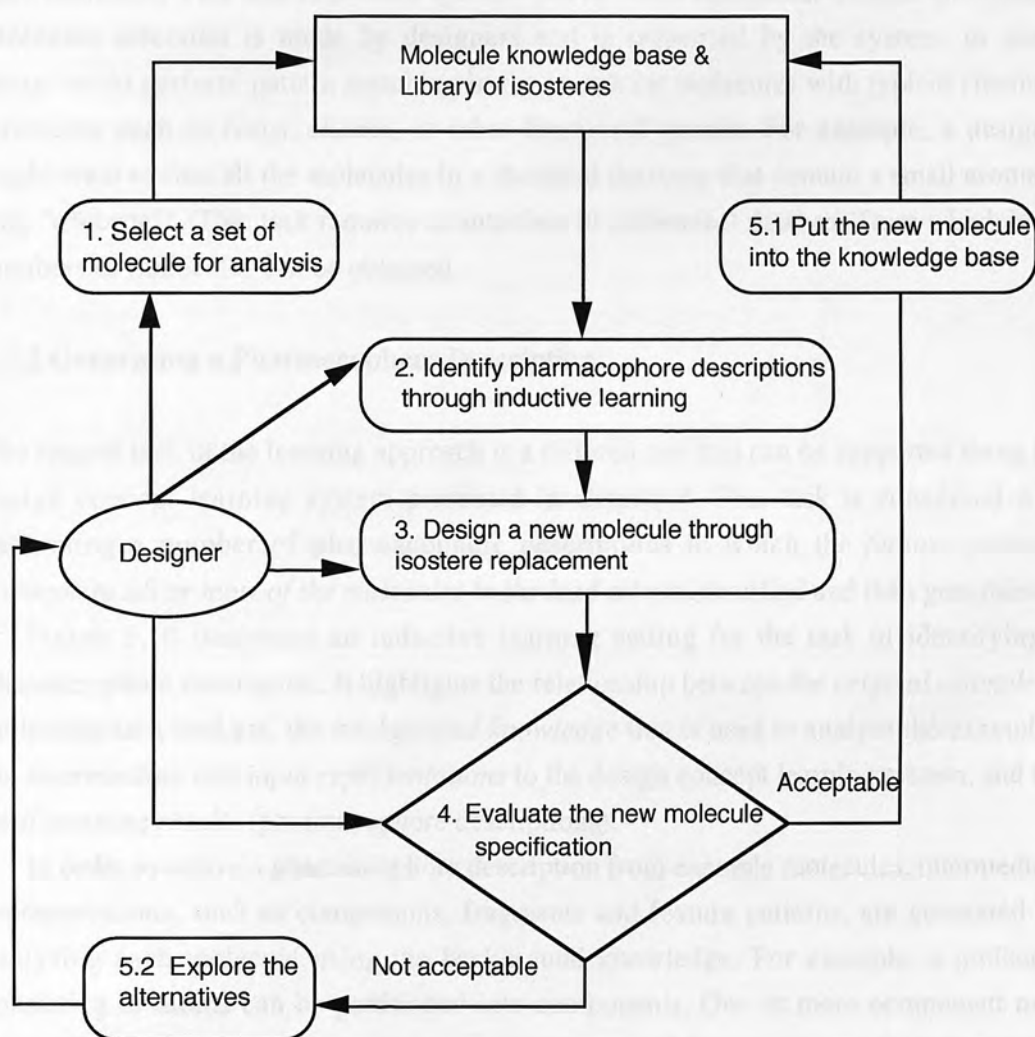


Figure 5.9: Model of Indirect Drug Design Process

5.3.1 Data Preparation

The past design experiences of designers may suggest that molecules fall into different groups, with each group having a distinct binding mode. Each of these groups is regarded as a series. The assumption is that the molecules placed within a series interact with the same receptor in the same way. The nature of a series may not be known and may have to

be discovered. One of a designer's important decisions is to decide which molecules to include in the study at this stage; another is which characteristics should be chosen to divide the molecules into series.

The first task in the learning approach involves selecting of molecules from a database, partitioning each molecule into molecular components, assembling molecular components into molecular fragments and the identification of feature patterns within each molecule. This task is domain specific and involves substantial domain knowledge. Molecule selection is made by designers and is supported by the system, to allow designers to perform pattern matching in the search for molecules with typical chemical structures such as *rings*, *chains*, or other functional groups. For example, a designer might want to find all the molecules in a chemical database that contain a small aromatic ring "c*cccc*". This task requires an interface to a chemical database from which large numbers of molecules can be obtained.

5.3.2 Generating a Pharmacophore Description

The second task in the learning approach is a difficult one that can be supported using the design concept learning system presented in chapter 4. This task is concerned with generating a number of pharmacophore descriptions in which the *feature patterns common to all or most of the molecules in the lead set* are identified and then generalised.

Figure 5.10 illustrates an inductive learning setting for the task of identifying a pharmacophore description. It highlights the relationship between the *original example* of molecules in a lead set, the *background knowledge* that is used to analyse the examples, the *intermediate and input representations* to the design concept learning system, and the *final learning results* (pharmacophore descriptions).

In order to derive a pharmacophore description from example molecules, intermediate representations, such as components, fragments and feature patterns, are generated by analysing each molecule using the background knowledge. For example, a molecule consisting of atoms can be partitioned into components. One or more component may form a fragment. A feature pattern is an arrangement of fragments, and feature patterns are used to describe the pharmacophore.

In Figure 5.10 the dashed lines between feature patterns and molecules indicate that a set of feature patterns derived from a molecule is an abstract view of how the molecule might bind to a receptor. How to support the task of generating pharmacophore descriptions from a set of molecule examples using a design concept learning system will be fully discussed in chapter 7. The result of this learning process is a design concept tree from which multiple pharmacophore descriptions can be retrieved.

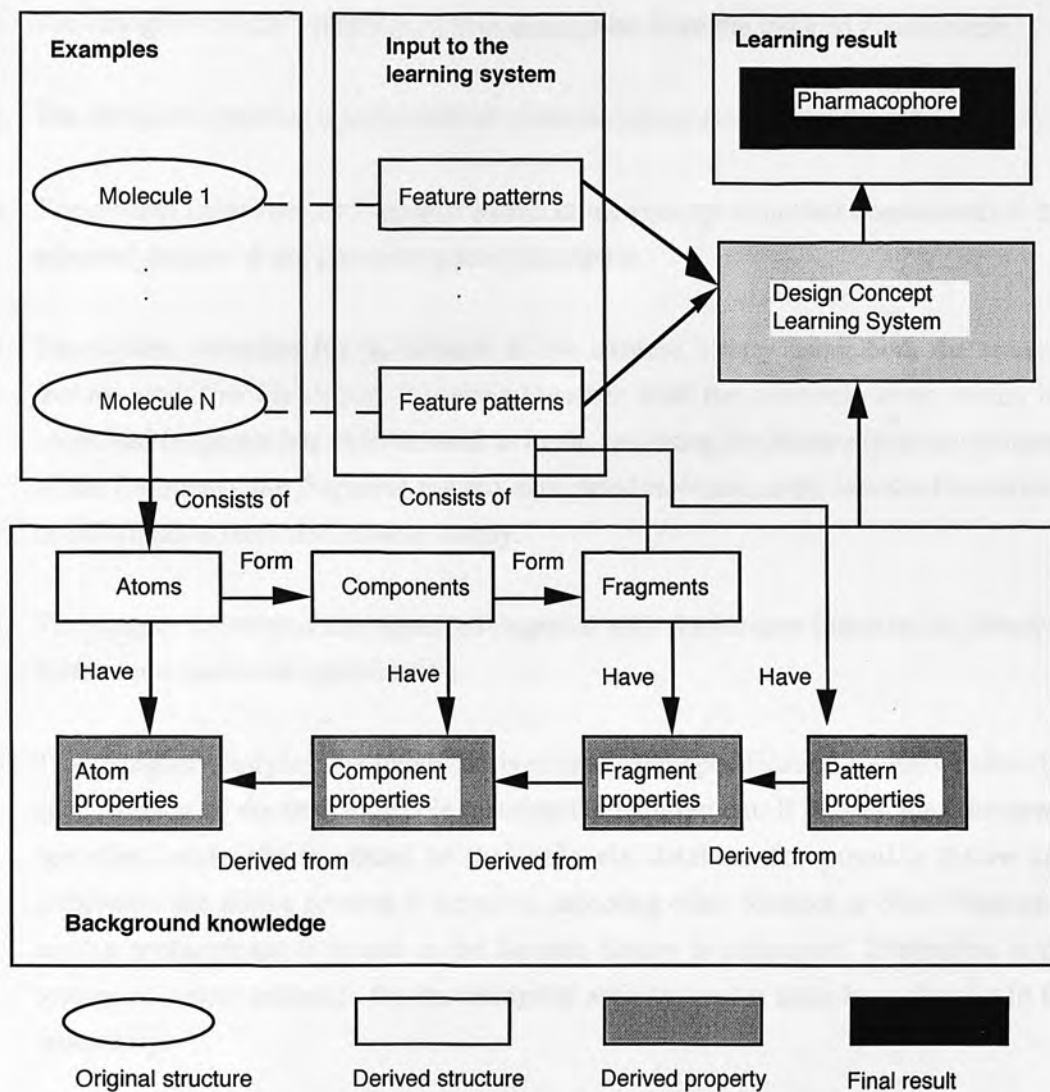


Figure 5.10: A Learning Setting for Drug Design

5.3.3 Designing a New Molecule via Isostere Replacement

The third task, which involves designing a new molecule, is achieved using a pharmacophore description to guide the specification of a new molecule that has the necessary properties and structure. In this stage, the pharmacophore description is used to guide the search for an *isostere* to replace a part of any molecule in the lead set, resulting in a synthetic candidate molecule being produced. This new molecule will bind to the receptor in a similar way to those molecules used to generate the pharmacophore description, but with different ranges of secondary properties and hopefully a higher activity.

The task of suggesting a synthetic candidate molecule is supported by a library of isosteres and is performed in five steps based on the induced design concept tree.

1. The designer selects a pharmacophore description from the induced concept tree.
2. The designer specifies a *feature* in the pharmacophore description for optimisation.
3. The system identifies the fragment stored in the concept node that corresponds to the selected *feature* of the pharmacophore description.
4. The system searches for an isostere in the isostere library using both the *selected feature* and the *identified fragment* together with the molecule from which the identified fragment has been derived as input, i.e., using the *feature* (primary property of the fragment), the *fragment* and the *associated molecule* as the indexes for retrieval of information from the isostere library.
5. The system substitutes the identified fragment with the isostere found in the library to form a new molecule specification.
6. The designer analyses the newly derived molecule specification to see whether the specification of the new molecule matches the requirement. If it does, then the newly specified molecule is added to the molecule database for possible future use. Otherwise the above process is repeated, selecting other features or other fragments, until a replacement is found or the isostere library is exhausted. Evaluation is the testing of a new molecule for its biological activity, and is done by a chemist in the laboratory.

Summary

The early stage of design involves analysing the characteristics of the design problem: parameterising and constraining a model of the design problem. In pharmaceutical design, this analysis is largely evidence-driven and is based on a theoretical understanding of how chemical properties are distributed and how they contribute differently to the binding of a drug molecule to its receptor.

One of the major activities of indirect drug design is the generation of a pharmacophore description from example molecules. A pharmacophore is, by definition, the total assembly of properties of a set of molecules required to bind to a given receptor. Pharmacophore generation is achieved by recognising feature patterns of fragments which are common to all or most of the molecules in the lead set. The derived pharmacophore description can be seen as a hypothesis about the arrangement of chemical properties of a required receptor for a new molecule to achieve the desired biological activity. As such it integrates across the feature patterns found for individual

molecules, identifying the common properties and the relationships between these properties.

This chapter has described the basic concepts, knowledge representation and task analysis in indirect drug design. The task of generating a pharmacophore description has been identified as an interesting and challenging problem for the development of a design concept learning system that can be integrated into a knowledge-based design support system architecture. This design concept learning scheme can be used to derive initial design problem structures from design examples, thus providing support to the problem at the early stage of the design process when such a problem structure is unavailable.

A software system has been implemented to support indirect drug design. Figure 5.11 shows the interface for selecting active molecules for analysis, and Figure 5.12 shows the interface for viewing the identified fragments (it highlights all the polar fragments that can be identified from a molecule called GDS1).

Chapters 1, 2, 3 and 4 have given general discussions and descriptions of knowledge-based design support and inductive learning. The remaining chapters will describe the software systems that have been implemented. Chapters 6 and 7 will discuss the exploration of multiple design solutions in indirect drug design using a knowledge-based design support system architecture, and identification of a pharmacophore description using the design concept learning system discussed in chapter 4.



Figure 5.11: Identifying Active Molecules

KBS for Drug Design

GDS1	GDS2	GDS3	GDS4
GDS5	GDS6	GDS7	GDS8
GDS9	GDS10	GDS11	GDS12

Compound name: GDS1

smiles: c1ccc(O)ccc2ccccc12

activity: 158e-6

measure: IC50

assay: PAF-induced primary aggregation in human platelet-rich plasma

source: EP0264114

date: 25/09/90

project: PAF

ring: (Antagonist Furydylseries)

Figure 5.11: Interface for Molecule Selection

Operation

Castlemaine Graphic Interface

Fragmentation

Control

HYDROPHOBIC

POLAR

AROMATIC

GDS1

GDS2

GDS3

GDS4

GDS5

GDS6

GDS7

GDS8

GDS9

GDS10

GDS11

GDS1-FRAG-1

GDS1-FRAG-2

GDS1-FRAG-3

GDS1-FRAG-4

GDS1-FRAG-5

GDS1-FRAG-6

GDS1-FRAG-7

GDS1-FRAG-8

GDS1-FRAG-9

Fragment Properties

Compound Name: GDS1

FRAGMENT	>FRAGV-FRAGFEFT ^o	AGLIENT-FRAGFEFT ^o	HSE-FRAGS
GDS1-FRAG-10	4-HROPHOBIC	TOO-LONG	GDS1-COMP-3 GDS1-COMP-2
GDS1-FRAG-9	4-AROMATIC	TOO-LONG	GDS1-COMP-5 GDS1-COMP-7
GDS1-FRAG-8	4-HROPHOBIC	TOO-LONG	GDS1-COMP-5 GDS1-COMP-3 GDS1-COMP-5
GDS1-FRAG-7	4-HROPHOBIC	TOO-LONG	GDS1-COMP-5 GDS1-COMP-5 GDS1-COMP-2
GDS1-FRAG-6	>CLAR	TOO-LONG	GDS1-COMP-4 GDS1-COMP-3
GDS1-FRAG-5	4-HROPHOBIC	TOO-LONG	GDS1-COMP-2 GDS1-COMP-3 GDS1-COMP-1
GDS1-FRAG-4	4-HROPHOBIC	TOO-LONG	GDS1-COMP-1 GDS1-COMP-2
GDS1-FRAG-3	>CLAR	TOO-LONG	GDS1-COMP-7
GDS1-FRAG-2	4-AROMATIC	TOO-LONG	GDS1-COMP-5
GDS1-FRAG-1	4-AROMATIC	TOO-LONG	GDS1-COMP-1

Figure 5.12: Identify Fragments

Chapter 6

An Architecture for Intelligent Design Support

Future design is likely to be supported by knowledge-based design support system tools in the same way that expert system applications are being supported by expert system development tools at present. Design exploration as an intelligent behaviour can be supported by an integrated system that represents and manipulates design knowledge. The *knowledge-based design support system architecture* described in this chapter is a software architecture which reflects components or sub-systems of a computer system and their interaction for intelligent design support. It is a computational environment that provides general design support functions. These functions include mainly:

- management of a design knowledge base,
- control of design knowledge sources,
- creation and maintenance of multiple design contexts,
- documentation of design history, and
- graphical explanation of design results.

The integration of a blackboard control system and an Assumption-based Truth Maintenance System (ATMS) forms the core of this architecture which is referred to as the ATMB architecture. This architecture has been implemented using a Lisp-based tool¹. This chapter describes the ATMB architecture and its implementation in terms of *knowledge representation, integration of a blackboard control system and an ATMS, context management, design documentation, and application.*

6.1 An Overview of the Architecture

The ATMB architecture, as illustrated in Figure 6.1, consists mainly of *a design knowledge base, a design concept learning system, a truth maintained blackboard control system, a design context management system, a design documentation system, and a graphical user interface.*

¹ This tool is described in Appendix C.

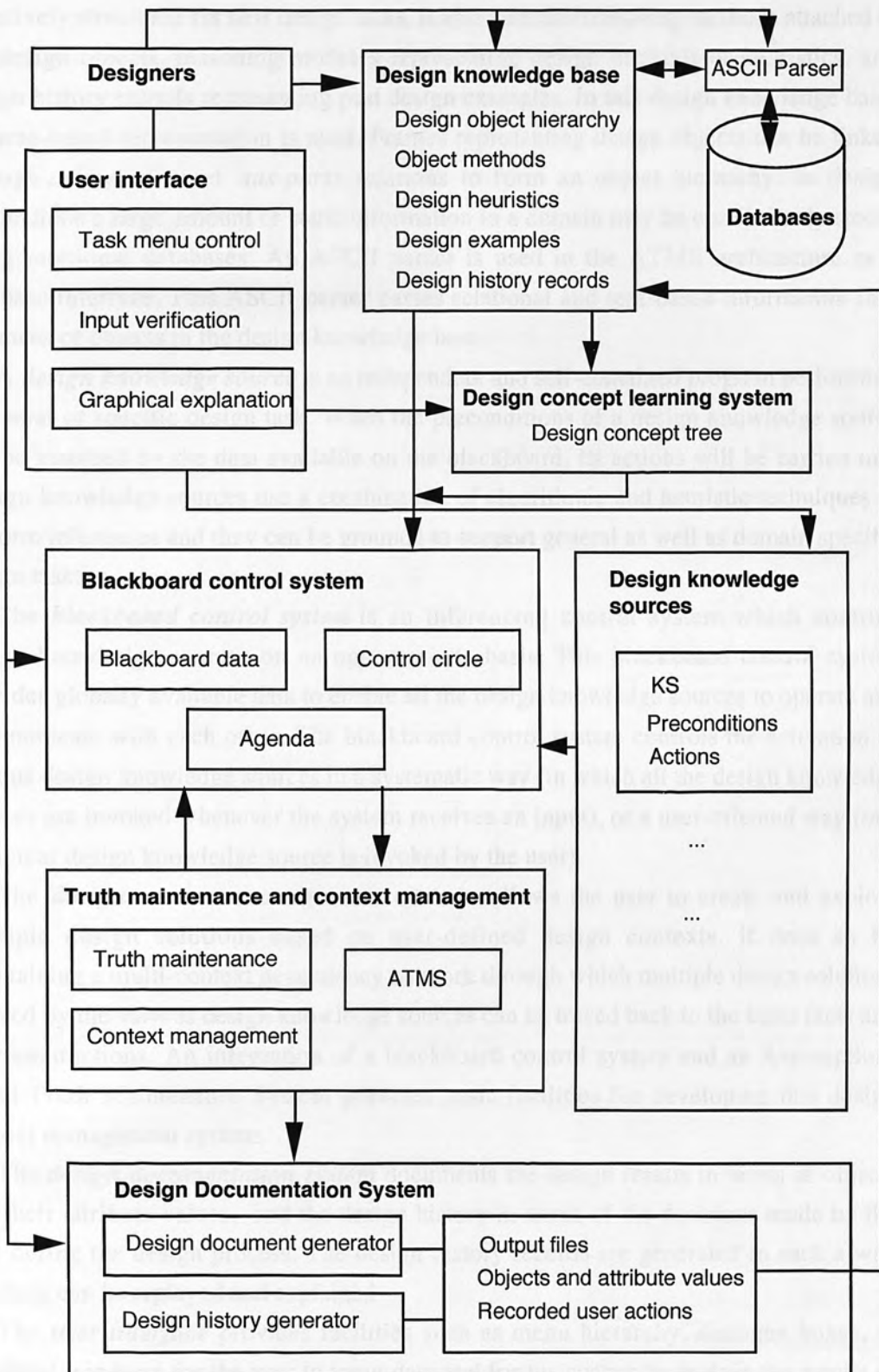


Figure 6.1: The Architecture

The *design knowledge base* is a collection of design object classes which can be

selectively structured for new design tasks. It also contains reasoning methods attached to the design objects, reasoning modules representing design methods or heuristics, and design history records representing past design examples. In this design knowledge base, a frame-based representation is used. Frames representing design objects can be linked through *a-kind-of* and *has-parts* relations to form an object hierarchy. In design applications a large amount of static information in a domain may be conveniently stored using relational databases. An ASCII parser is used in the ATMB architecture as a database interface. This ASCII parser parses relational and text-based information into instances of objects in the design knowledge base.

A *design knowledge source* is an independent and self-contained program performing a general or specific design task. When the preconditions of a design knowledge source can be matched by the data available on the blackboard, its actions will be carried out. Design knowledge sources use a combination of algorithmic and heuristic techniques to perform inferences and they can be grouped to support general as well as domain specific design tasks.

The *blackboard control system* is an inferencing control system which controls design knowledge sources on an opportunistic basis. This blackboard control system provides globally available data to enable all the design knowledge sources to operate and communicate with each other. The blackboard control system controls the activation of various design knowledge sources in a systematic way (in which all the design knowledge sources are invoked whenever the system receives an input), or a user-oriented way (one particular design knowledge source is invoked by the user).

The *design context management system* allows the user to create and explore multiple design solutions based on user-defined design contexts. It does so by maintaining a multi-context dependency network through which multiple design solutions derived by the various design knowledge sources can be traced back to the basic facts and user assumptions. An integration of a blackboard control system and an Assumption-based Truth Maintenance System provides basic facilities for developing this design context management system.

The *design documentation system* documents the design results in terms of objects and their attribute values, and the design history in terms of the decisions made by the user during the design process. The design history records are generated in such a way that they can be replayed and explained.

The *user interface* provides facilities such as menu hierarchy, dialogue boxes, or graphical windows for the user to input data and for the system to explain the results or the status of the system. Any user input is verified first by the user interface and then translated into the internal system operations. A graphical explanation system is included in the user interface to explain the current status of the system and any new knowledge generated during the course of a design.

As an integral part of the architecture, the *design concept learning system* is treated as conceptual support system that derives a design concept tree from design exempts. When the concept or the product model of an artefact already exists in the design knowledge base, the design tasks can be carried out by loading these concepts and product models from the design knowledge base. When such a concept or product model does not exist, the design concept learning system is used first to derive a basic design solution structure from design examples.

The ATMB architecture as a whole provides a basic framework as well as a software kernel for developing knowledge-based design systems.

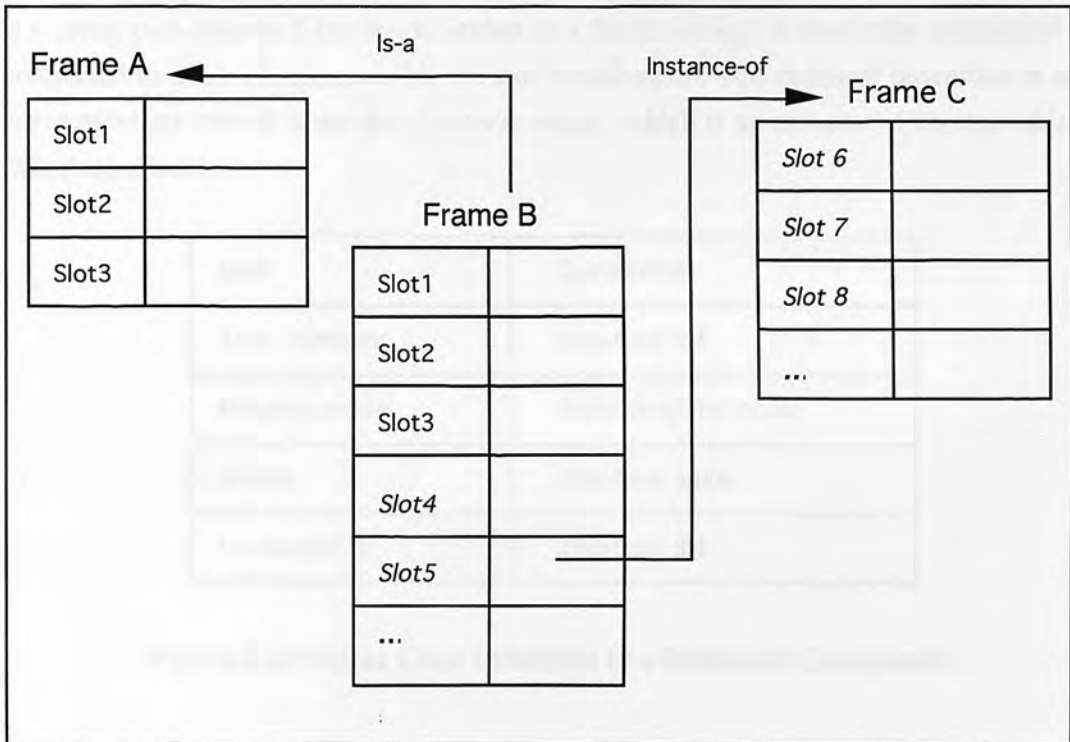


Figure 6.2: Frame-based Representation

6.2 Knowledge Representation

A frame-based representation is used to represent design objects and their relations in the design knowledge base of the ATMB architecture. Information in a frame-based representation is structured in a network representation called the *lattice*. The lattice is composed of *frames* and *instances*. A frame is used as a template to define a class of objects. The attributes of an object class are described by *slots*. Instances of a frame can be created to represent specific objects. An instance has the same slots as the associated frame, but also holds values in the slots. In a frame-based representation, the *is-a* relation

is created by class inheritance whilst and the *has-parts* relation is created by linking a slot to an instance of another class.

For example, as illustrated in Figure 6.2, frame B inherits slot 1, slot 2, and slot 3 from frame A because it can be defined as a *kind-of* frame A, i.e., a sub-class of frame A. Frame B has some local slots such as slot 4 and slot 5. Slot 5 demonstrates a typical *has-parts* relation because its value is an instance of frame C.

Figure 6.3 shows the definition of an object representing a specific molecular component. In this definition, the slot *atom-numbers* refers to the atom numbers of the component used in a graphical user interface for depicting the structure of the molecule. The slot *Smiles* is used to store the computational representation of the component which is a string (see chapter 5 for the definition of a *Smiles string*). A molecular component is connected to other components via the slot *connected-to*. The chemical properties of the component are stored in the slot *Property-model*, which is an instance of another object called *mc-model*.

Slot	Constraints
Atom-numbers	Lisp-type: list
Property-model	Instance-of mc-model
Smiles	Lisp-type: string
Connected-to	Lisp-type: list

Figure 6.3: Object Class Definition of a Molecular Component

In a frame-based representation, frames, instances, and slots are referenced by symbols. A slot has the identity of *instance-name* and *slot-name*, as illustrated in Figure 6.4. A slot is a basic data unit for defining design variables or parameters. In order to create and maintain multiple contexts within this representation scheme, an assertion is used to link a slot of an instance with an ATMS node. In a Lisp system this assertion has the form of

(Slot-ID, ATMS-node-ID)

In an Assumption-based Truth Maintenance System (ATMS), the same piece of data may have different versions in different ATMS contexts. A slot may therefore appear in more than one place and this can be explained through its ATMS-node-ID from which the context of the slot is to be retrieved. The justification and the context of a slot is accessed

through its ATMS-node-ID and the actual data is accessed through its slot-ID. In this way, a frame-based representation can be extended to deal with justified reasoning and multiple contexts. In the ATMB architecture an assertion base is used to store these links. Additional functions are provided for access to a piece of data either via its AIMS-node-ID or its Slot-ID. This assertion-based linkage between a frame-based representation and the ATMS enables the computation in the ATMS to be separated from the actual data, and for any piece of data derived by the design knowledge sources to be explained either in object terms or ATMS terms.

Data type	Identity
Instance	(Class-name Instance-name)
Slot	(Instance-name Slot-name)
ATMB data-unit	<div style="text-align: center;"> (Slot-ID ATMS-node-ID) </div> <div style="display: flex; justify-content: space-around; align-items: center;"> Datum ← └─┬─▶ (Justification Context) </div>

Figure 6.4: Basic Data Connection Unit for the ATMS

6.3 The Control System

6.3.1 The Design Knowledge Source

Design knowledge sources represent inferential knowledge that performs general and domain specific design tasks. In the ATMB architecture, design knowledge sources differ in size but they all have the same format. As illustrated in Figure 6.5, each design knowledge source has a *name* slot, a *checking-fn* slot for checking its preconditions, an *active-p* slot to indicate whether it is active or not (a design knowledge source in the ATMB architecture can be explicitly activated or deactivated by the user), a *consequents-fn* slot to store the functions that actually carry out the actions of the design knowledge source, a *rule-sets* slot to store the rules if the action of the design knowledge source is to be carried out by a set of rules, and a *preconditions* slot to store the antecedents by which the inferred results are to be justified.

The preconditions of each design knowledge source are statements for the following three checks: there must be data in the lattice that the knowledge source can work on; the data must be in the current context; and the proposed actions of the design knowledge source should not have already been done. Whenever a design knowledge source is invoked, a precondition checker associated with that design knowledge source will first

carry out these checks. If these checks are passed, the design knowledge source places a Knowledge Source Activation Record (KSAR) on the blackboard agenda.

Slot	Description
Name	Name of the design knowledge source
Active-p	Status of the design knowledge source (yes/no)
Preconditions	Preconditions of the design knowledge source
Checking-fn	Precondition checking function
Consequents-fn	Action functions
Rulesets	Rulesets associated with the design knowledge source

Figure 6.5: Definition of Design Knowledge Source

Design knowledge sources can be explicitly enabled or disabled by the users. Only enabled design knowledge sources are invoked by the blackboard control system. Disabled design knowledge sources are temporarily out of use until such a time they are explicitly enabled by the users. This allows the resource of the system to be explicitly allocated to some of the design knowledge sources that the designer may consider to be more relevant to the current design problem.

Design knowledge sources can also be ordered in the ATMB architecture using their priority values. Theoretically, design knowledge sources do not necessarily have any priority values because they rely only on the matching of their preconditions against the available data on the blackboard to produce a bid for work. However, ordered design knowledge sources can speed up the inference process when the number of the design knowledge sources is large and when some of them perform a design task in sequence.

6.3.2 Blackboard Data Structure and Control

The blackboard itself is defined as an object structure whose definition as illustrated in Figure 6.6. This means that multiple blackboards can be created. In Figure 6.6, the gray shadowed boxes represent the components within the ATMB architecture that are different from the existing systems as reviewed in chapter 2 (Section 2.2.2). These components are necessary for the integration of the blackboard and the ATMS (see Section 6.4 and 6.5 for more detailed discussions on integration and design context management).

A *blackboard agenda* is a place in the blackboard system where the KSARs generated by design knowledge sources are stored. The blackboard control strategy consists of a two-phase (matching and firing) control cycle.

In the matching phase, each design knowledge source in the blackboard system is invoked in turn. If the preconditions of any design knowledge source can be matched by the data already on the blackboard, this design knowledge source generates a KSAR and places it onto the blackboard agenda.

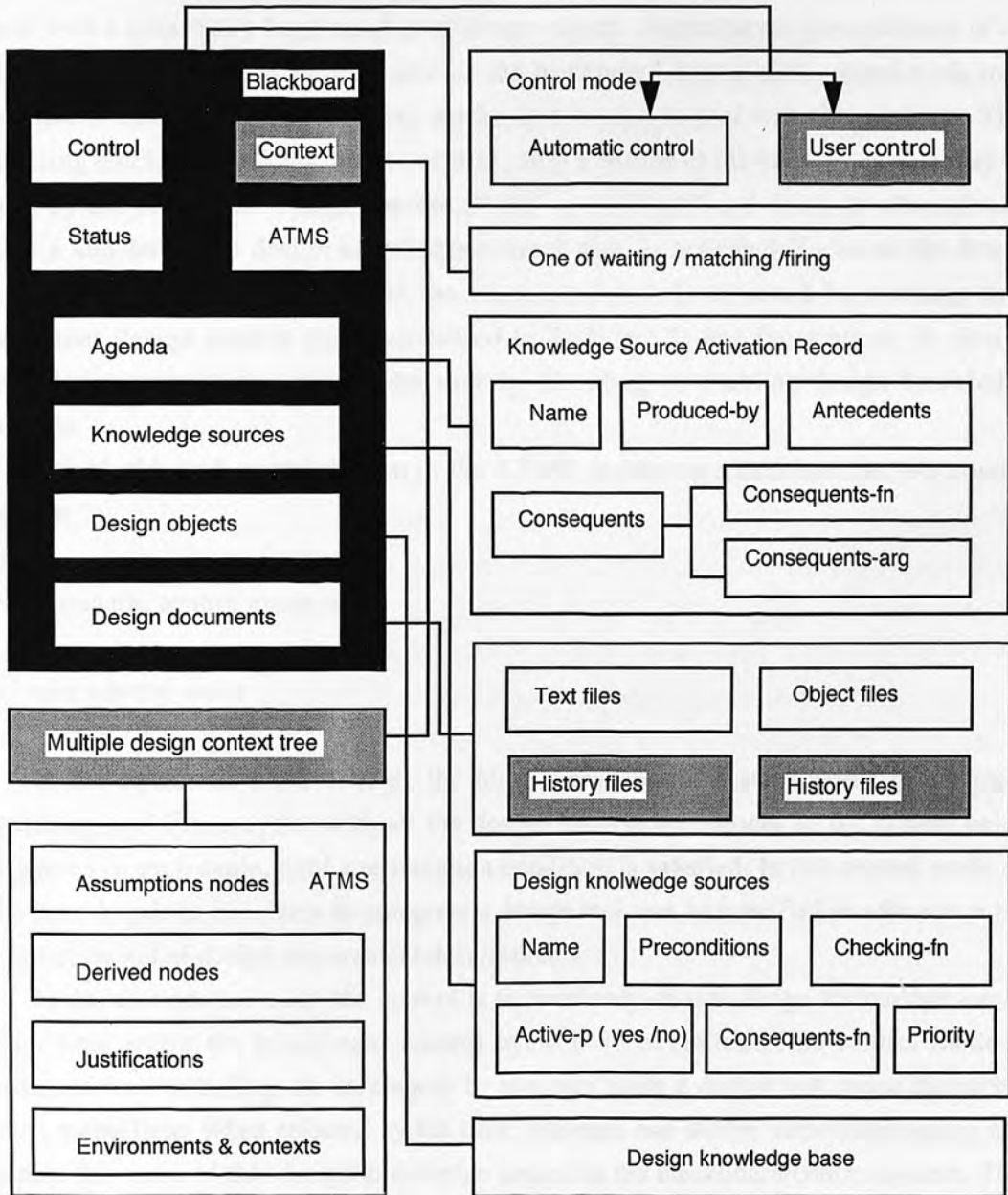


Figure 6.6: The Data Structure of the Blackboard

The firing phase starts when the matching phase is finished, i.e., when no more KSARs can be generated by any design knowledge sources. In this phase, the KSAR at the top of the blackboard agenda is executed, resulting in a particular design action being carried out. The execution of any KSAR may result in the blackboard data being changed, in which case the design knowledge sources are triggered again to propose new KSARs. This blackboard control cycle stops only when the agenda becomes empty. This is the termination condition of the blackboard control.

In a knowledge-based design application, the blackboard control system may have to deal with a potentially large number of design objects. Matching the preconditions of all the design knowledge sources against all the blackboard data at each control cycle may become a slow process. A focusing mechanism is used to deal with this problem. This focusing mechanism allows that at one time, only a portion of the blackboard data may be seen by the design knowledge sources (focus on the blackboard data), or alternatively, only a sub-set of the design knowledge sources may be activated (focus on the design knowledge sources). Focusing on the blackboard data is achieved by working on a particular design context (to be discussed in Section 6.5) and focusing on the design knowledge sources is done by the user by disabling or enabling design knowledge sources.

The blackboard control system in the ATMB architecture therefore has two control modes:

1. automatic control mode, and
2. user control mode.

In the automatic control mode the blackboard control system works in a typical matching-and-firing cycle, with all the design knowledge sources in the system being triggered in each cycle, until a termination condition is satisfied. In this control mode all the user decisions necessary to complete a design task can be specified in advance in the form of an initial design requirement description.

In the user control mode the control is focused only on one design knowledge source each time while the blackboard control cycle is in operation. This control mode is designed for controlling the inferences by the user using a design task menu hierarchy. Each menu item, when selected by the user, activates one design knowledge source and passes the name of this design knowledge source to the blackboard control system. This control mode is used for the designers to monitor the design process step by step.

This blackboard control system has been implemented as a stand-alone system with a graphical interface. The major interface functions of this blackboard control system are described in Appendix B.

6.4 Integration of ATMS and the Blackboard

The generation, maintenance and explanation of design contexts for exploring alternative design solutions are the additional functions of the ATMB architecture compared with other blackboard-based design systems that have been reviewed in this thesis. As discussed in chapter 2, an *assumption-based truth maintenance system* (ATMS) is a system that maintains the consistency of a multi-context justification network based on user assumptions, i.e., the design decisions made by the users. Based on an integration of an ATMS and the blackboard control system discussed above, a design context management system has been developed for creating and maintaining multiple design contexts when the structure of a design problem is being explored by designers.

6.4.1 Ross' ATMS

A version of an ATMS based on de Kleer's work [de Kleer 1986], designed and implemented by Ross [Ross 1989], is integrated with the blackboard control system discussed above.

The reason for choosing Ross's ATMS is that it is implemented in C and therefore would run faster than if implemented in Lisp. In addition, Ross's implementation of the ATMS has the features of simplicity and flexibility suitable for design applications [Banares-Alcantara 1991]. It uses integers to represent nodes and assumption identities. The actual datum in Ross' implementation is not recorded as part of the nodes as the standard ATMS algorithm described by de Kleer [de Kleer 1986]. Therefore this ATMS works faster than the standard ATMS algorithm. The representation of an actual datum in an application is flexible as long as there is a method to link the datum with the ATMS nodes. The datum linking method used in the ATMB architecture for the integration has been explained in section 6.1 (see Figure 6.4).

Ross's ATMS provides seven interface functions as listed in Appendix B. To integrate these C-based functions into a Lisp environment, a *foreign function environment* is created so that the C functions in Ross' ATMS can be called within the Lisp environment of the ATMB architecture as foreign functions. The details for creating such a foreign function environment are omitted here but given in Appendix C.

6.4.2 Integration

A blackboard control system integrated with an ATMS can be called a truth maintained blackboard in that the blackboard has a multi-contextual justification network to support the inference and the exploration of multiple solutions. In such a truth maintained

blackboard system, the ATMS builds a justification network in the form of the *assumption nodes* representing decisions or assumptions made by the designers, and the *derived nodes* representing the results inferred by the design knowledge sources. The ATMS maintains the dynamic growth of this justification network by updating the information associated with each node whenever a piece of new information is derived by the design knowledge sources. This allows all the newly derived information to be justified in the context of basic design decisions (user assumptions), and to be maintained throughout a design session.

As discussed in chapter 2, inconsistencies can be detected by the system or declared by the designers. Any nodes in the system without a justification will not be processed by the design knowledge sources.

In order for the system to maintain the dependency network for the knowledge derived based on user assumptions, a KSAR proposed by any design knowledge source must carry the necessary information for the truth maintenance system to establish the link between a proposed action and its preconditions. This link is used by the truth maintenance system to build the justification for the derived result.

A KSAR, as illustrated in Figure 6.6, is an object instance representing a bid of a design knowledge source to perform a general or a specific design task. A KSAR carries the antecedents of a proposed action via the slot called *antecedents*, i.e., the conditions under which the proposed action is to be carried out, or in other words, the data which justify the proposed actions of the KSAR.

A KSAR also provides arguments, using the bindings established during pattern matching (via the slot called *consequents-args*), for the action functions that actually perform a design task. It also tells the system, via the *consequents-destination* slot, where to put the results generated by the design knowledge source. The *Produced-by* slot links a KSAR with the design knowledge source responsible for proposing it.

Figure 6.7 illustrates the control process of an integrated blackboard and an ATMS in which KSARs are proposed by the design knowledge sources and checked by the system.

- A KSAR is firstly checked for its format by the system. If the format is right, and the new KSAR is not a duplicate of any KSARs already in the agenda, it is placed onto the blackboard agenda. Otherwise the KSAR is ignored.
- Before a KSAR is executed, it is checked again to see whether its preconditions still hold, i.e., to see whether the antecedents of the KSAR remain consistent. The system does this by checking whether the KSAR has a valid set ATMS nodes and assumptions as the justification for the proposed action.
- When a KSAR has been executed, the generated results are placed on the blackboard.

The system creates a new ATMS node for the results and adds its justification to the ATMS database.

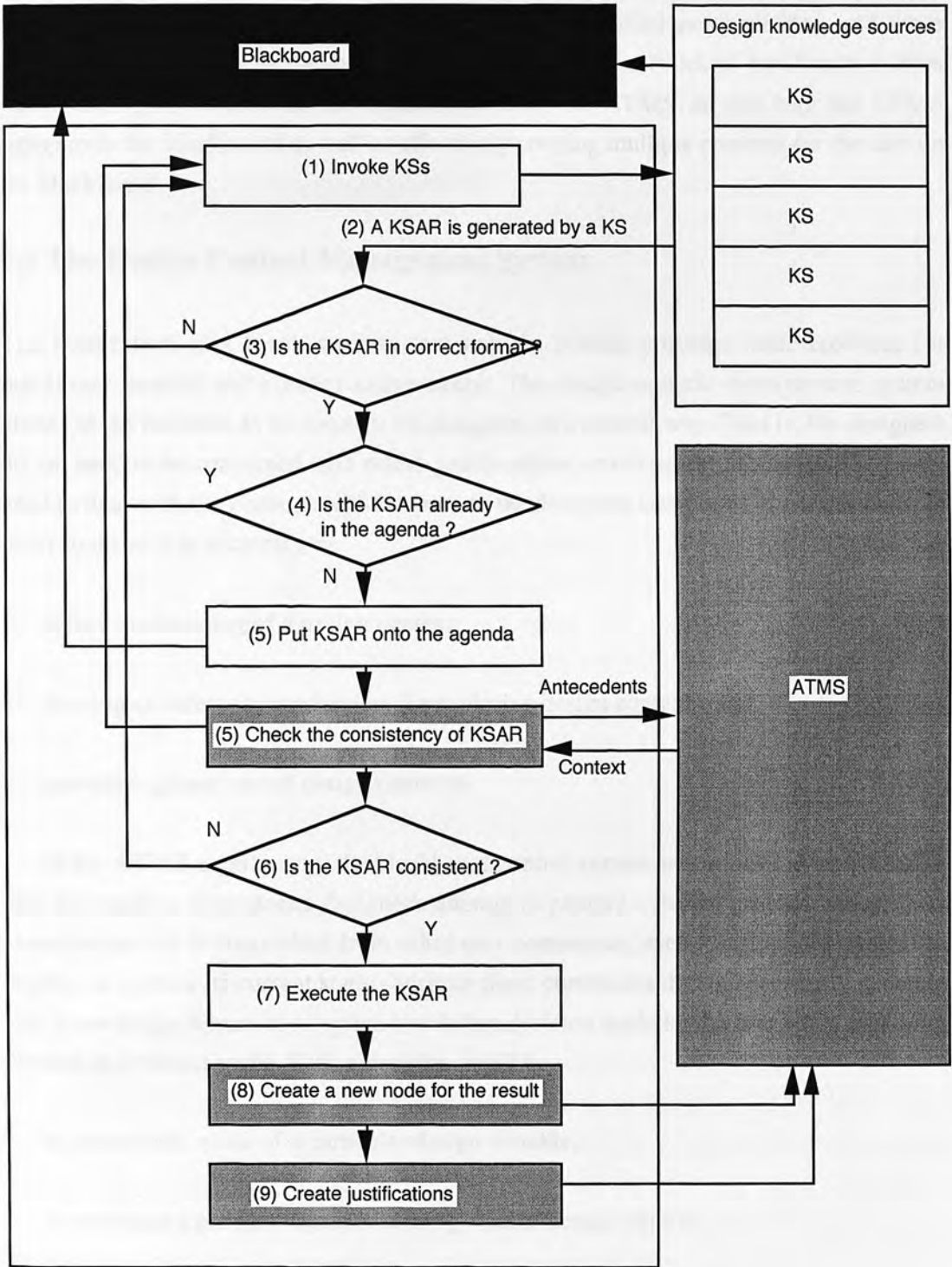


Figure 6.7: Truth Maintained Blackboard Control Process

In Figure 6.7, the grey shadowed boxes represent the components that are absent from

standard blackboard systems. These components provide a basis for the development of a design context management system (see next section). The main difference between a standard blackboard control system and a truth maintained blackboard control system is that in the latter system, the inference is based on justified preconditions, and these preconditions are used as the antecedents to build up the network of justifications from which multiple contexts can be established using the ATMS. In this way the ATMS *safeguards* the blackboard as well as effectively creating multiple contexts for the data on the blackboard.

6.5 The Design Context Management System

The integration of a blackboard system and the ATMS provides basic facilities for intelligent control and context maintenance. The design context management system allows these facilities to be used by the designers in a natural way. That is, the designers do not need to be concerned with nodes, justifications, environment, labels etc. They only need to deal with the *contexts* which represent the designers *viewpoints* or *perspectives*. In order to do so it is necessary to:

- define the meaning of a design context;
- develop an inference mechanism for exploring design contexts; and
- provide explanations of design contexts.

In the ATMB architecture, the blackboard control system acts on user assumptions. A user assumption represents a designer's attempt to explore a design problem space. User assumptions are distinguished from other user commands, such as asking the system to display or explain its current status, because these commands do not necessarily generate new knowledge. A *user assumption* is a design decision made by the user while exploring the design problem space. Such a decision might be

- to assume the value of a particular design variable,
- to construct a product data model using a set of design objects,
- to choose a particular design method or a control parameter, or
- to add, delete or modify design objects.

A *design context* is a design solution (a complete set of design variable values) or a partial design solution (a sub-set of design variable values) based on consistent user assumptions (or decisions). A design solution or a partial design solution can therefore be justified in a context consisting of:

1. The design data (design objects);
2. The design methods (design knowledge sources) used to manipulate the data; and
3. The design variable values assumed by the designer.

A design method can be interpreted here as a set of constraints, or a set of rules that constrains the design variables and dependent design parameters. The design is started by loading a set of basic design objects from the design knowledge base to form a conceptual structure of the design problem. If the existing design objects available in the design knowledge base cannot be directly used to build such a structure, a *design concept learning system* is used to derive it from design examples.

The ATMB architecture, as an inferencing control system, can make inferences from any values of the design variables assumed by the designers, by propagating them throughout the constraint set, to derive the values of other design variables. Alternative design solutions are obtained by creating design contexts by members of a design team who may want to make different decisions on the choices of either the design data, the design methods, or the design variable values.

The role of the *design context management system* is to enable the designers to create contexts and to switch from one to another. It allows the designers to concentrate on the conceptual aspects of multiple design solution exploration without having to worry about the details of the implementation. To do so, the design context management system in the ATMB architecture uses the concepts of *design route* and *design branch*. A new *design route* means loading a different set of design data to start the design whilst a new *design branch* can be created under a design route or a design branch for making different choices on either design variable values or design methods. A design branch inherits the design decisions made in its parent design route or branch. Multiple design routes and design branches created by the designers form a *design context tree*. For example, as illustrated in Figure 6.8, design branch 2 inherits the method set, constraint set and evaluation criteria of design route 1, but focuses on a smaller set of objects (object set 2) taken from object set 1 to explore.

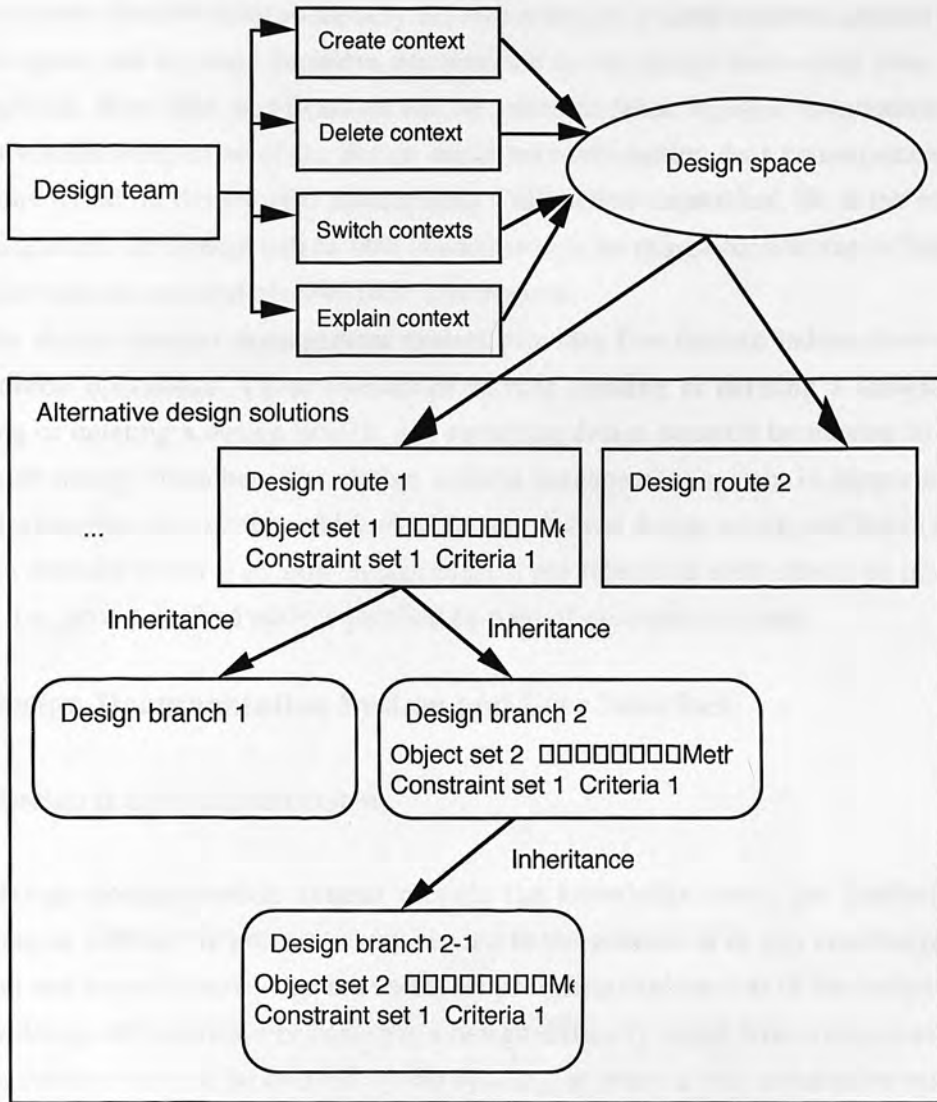


Figure 6.8: Exploring Design Solutions in Different Contexts

Anyone involved in the design process may want to start work at any point in the design process either from the beginning, or by inheriting some routes and branches of the existing design context tree created by others to explore them further in his or her own contexts. The creation of the design context tree is recorded as the history of the design, which can be saved for replay purposes or learning purposes (the recording of design history will be further discussed in the next section).

At the end of a design session, there may be more than one design route or branch, each of which represents a different design context justified by a set of different design data, design methods, and design variable values. With the design context management system, the ATMB architecture is well placed to explain the contexts of individual solutions using their justifications. The designers who created these design solutions can

then compare them in order to identify the best solution. If some common ground can be agreed upon, that is, some decisions are accepted by the design team while some others are rejected, then false justifications can be added to those rejected assumptions. This means withdrawing some of the design decisions made earlier. As a consequence, those solutions based on the rejected assumptions will become unjustified. So at the end of a design session, the system will be able to sort through the design context tree to find those solutions which are based on confirmed assumptions.

The design context management system provides five domain independent design exploration operations. These operations include creating or deleting a design route, creating or deleting a design branch, and switching design contexts by moving to design routes or design branches. The design context management system is supported by a graphical explanation system which displays any derived design results and their contexts in both domain terms (i.e., how design objects are related to each other), or in ATMS terms (i.e., how a derived node is justified by a set of assumption nodes).

6.6 Design Documentation System and User Interface

6.6.1 Design documentation system

The design documentation system records the knowledge about the artefact being designed as well as the process which has led to the generation of this knowledge. Both product and process knowledge are useful for providing explanations of the design results or any design difficulties. For example, a design difficulty might arise when an expected design solution cannot be derived by the system, or when a user assumption results in some of the constraints in the system being violated. In these situations the designers need to find out the source of the difficulty. The design documentation system provides necessary explanations of any chosen aspects of the current knowledge held in the system. A *design document* in the ATMB architecture contains complete and consistent sets of design objects and the values for all object attributes together with their justifications.

The design documentation system of the ATMB architecture is illustrated in Figure 6.9. Three different files are generated to form a design document. The first is a text-based description of design results and their justifications, each in a different context; the second is an object file, in which all the blackboard objects created during a design session are recorded; the third is a history record, in which all the user decisions made during a design session are recorded in the order in which they are created.

In a computer-based design system, repeating a particular design session or restoring an interrupted design session is useful, especially when the design project lasts for a long time, or is carried out by a group of designers. The design documentation system

produces a design history record for the designers to review the details of previous design sessions and, if necessary to replay a design history record.

A *design history record* is a sequence of decisions made by the designers during a design session. Such a record does not keep all the information representing the state of the system when it is recorded. Instead it records all the decisions that led to that state. In particular a history record is stored in such a format that it can be understood by the designers and it can be replayed.

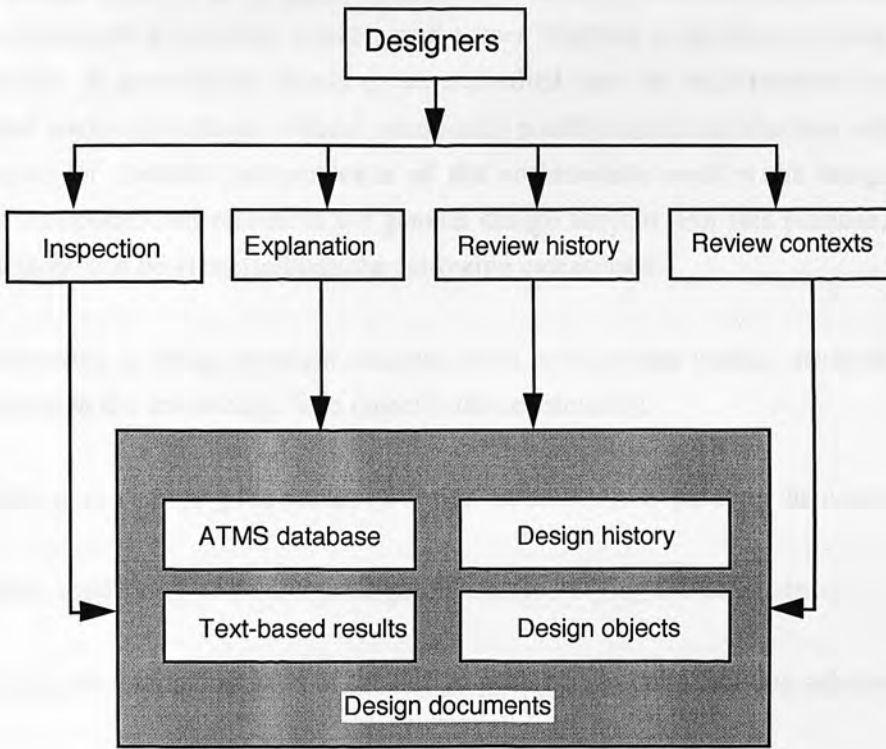


Figure 6.9: Design Documentation System

The traceability of a long term design project can be maintained by repeating any previous design session by loading its design history files. In this way, the knowledge of an experienced designer can be monitored and recorded, allowing design knowledge to be accumulated.

6.6.2 User Interface

The users of the ATMB architecture are not expected to be familiar with the terminology of blackboard and ATMS. The content of the blackboard must be presented graphically to the users in domain terms; particularly when a number of redesigning tasks have been carried out it is important to explain how results have been derived. A graphical

explanation system is used to explain the current state of the system in terms of object relations or ATMS justifications. The explanation given by the system in terms of ATMS justification is only necessary for debugging purposes and is not expected to be shown to the users. In addition, the designers can browse or inspect the objects on the blackboard, or ask the system to list all the solved variables or unsolved variables.

From a system integration point of view, the architecture is a self-contained and domain independent system, which supports the representation and maintenance of various design objects. In an application, this architecture needs to be connected with domain dependent knowledge sources and a user interface to perform domain specific design tasks. A mechanism needs to be embodied into the architecture for an easy integration with applications without necessarily modifying the architecture software. A high degree of domain independence of the architecture enables the integration of different computational resources for general design support. For this purpose, the user design actions can be classified into the following categories:

- Constructing a design problem structure (or a product data model) using the objects available in the knowledge base (specify the constraints);
- Assuming or changing the values of design variables (manipulating the constraints);
- Adding, modifying or deleting design objects (modifying the constraints);
- Focusing on interesting parts of the design problem structure (solving sub-problems);
- Selecting and documenting design solutions (evaluating the results).

User actions in the first four categories are likely to create new knowledge and thus to trigger the blackboard control cycle. In a blackboard control system, the control system triggers knowledge sources whenever something is entered onto the blackboard. The role of an interface is *verify* user actions and then *decompose* them into executable internal ATMS and blackboard system operations. The verified user actions are then subsequently recorded in a file that can be reloaded to replay a design session. This file contains a sequence of Lisp assertions representing user actions taken in a particular design session in the form of (*process-user-activity* :*function* :*arg-list*). Here *process-user-activity* is a macro which calls the *function* with the *arg-list*. In this way, a previous recorded design session can be fully replayed or partly replayed step by step.

6.7 Application

The ATMB architecture has been successfully integrated in the Castlemaine drug design support system. The Castlemaine drug design support system controlled the design process using the ATMB architecture in a graphical window called the *main design sheet* which included a *menu hierarchy*, a *design environment status panel*, a *multiple design context exploration panel*, a *design result display panel* and a *molecular graphics display window*. The tasks in the menu hierarchy are connected to the ATMB architecture through the user control mode. Appendix C lists the main functions implemented as part of the ATMB architecture for reuse purposes.

Using the ATMB architecture, a typical drug design session may involve the following steps:

1. A designer starts a design session by selecting a set of molecules from a database. This selection is based on his or her own judgement on what molecules in the database are most relevant to the drug to be designed. The system passes molecules from a text-based data file, translates them into object instances, and then depicts their molecular structures in the molecular graphics display window (See Figure 6.10).
2. The designer then starts to analyse the molecules by selecting *analysis-full* from the menu hierarchy. The system then asks the designer to make some assumptions about: (1) what molecules to analyse; (2) what set of fragmentation rules to use and (3) the number of molecules to be used to generate the initial pharmacophore description. This is illustrated in Figure 6.11.
3. The result of the analysis is a pharmacophore description (See Figure 6.11). In the design result display panel named 'pharmacophores', each box represents a primary property feature of the pharmacophore description.
4. A designer might next want to explore, for example, alternative rules for fragmentation; in which case, a new design branch can be created (see Figure 6.11) and this new branch can be given a name, say *design2*. In this new design branch the designer can use a different set of fragmentation rules, resulting in another pharmacophore description being derived. The system then creates a new context by inheriting all the assumptions already made in its parent design context (*design1*), and asking the designer to change the fragmentation rule set. The designer can also create a new design route by loading a different set of molecules causing a new design route to be created. In this way, designers can examine the difference between using

different fragmentation rules or data sets.

5. To design a new drug, a lead molecule is first selected in the lead set; for example, in Figure 6.15, a molecule called *burmimide* is selected .
6. A primary property feature of the pharmacophore is selected by the designer (who decides which part of the lead molecule is to be improved although the system can make initial suggestions). This is performed in the design result panel by clicking onto the relevant part of the pharmacophore icon.
7. The system is then asked to search its isostere library for possible isostere replacement suggestions (see the window called "Choose the best isostere" in Figure 6.12). In this example, the system finds two isosteres and suggests that a fragment in *burmimide* can be substituted by a preferable isostere shown in a black box. This suggestion is based on (1) finding all the isosteres that match the primary property of the fragment to be improved); and (2) comparing the closeness of their secondary properties to that of the fragment to be improved, and finding the most appropriate one.
8. The designer can accept the system's suggestion or make his or her own choice. Once the suggestion is confirmed or the designer's own choice is made, the system carries out the substitution, resulting in a new molecule being produced. This new molecule is put back into the molecule knowledge base for possible reuse or re-analysis.

By using the graphical explanation system, a designer can view the derived pharmacophore structure. For way of example, Figure 6.13 shows the explanation of the structure of a pharmacophore. It shows how a pharmacophore is described by its primary property feature; how each feature is derived from fragments found in the lead set; what the components of each fragment are; and what the properties of each component are etc. In ATMS terms, each node in the ATMS database can be explained by its antecedents and consequences. For example, in Figure 6.14, node 21 (a component) is justified by node 4 (a molecule) and node 17 (a lead set selection decision), or node 4 (the same molecule) and node 59 (another lead set selection decision). Unless both lead set selection decisions, or the molecule itself, are modified, node 21 holds consistently in two contexts $\{\{4\ 17\}\ \{4\ 59\}\}$. Furthermore node 21 justifies nodes 36 and 65, each of which represents a fragment node derived using a different set of fragmentation rules.

Summary

The effective integration of a blackboard control system and an assumption-based truth

maintenance system provides a unique mechanism for design context exploration and context management. A way of modelling design context has been presented and a way of exploring contexts using the concepts of design route and design branch has been implemented.

The ATMB architecture described in this chapter has been implemented as a self-contained knowledge-based design support system kernel using an object-oriented knowledge representation that is easy to interact with domain specific design knowledge sources. The architecture forms the core of a software kernel with facilities in the control of design processes, the exploration and management of multiple contexts of design, the graphical explanation of design results, and the documentation of the design results and decision process as design history.

The successful conclusion of the Castlemaine project in which the ATMB architecture was used to build a knowledge-based drug design support system has demonstrated that it is capable of controlling the design exploration process opportunistically and maintaining multiple design contexts [Smithers *et al* 1992].

The ATMB architecture as a knowledge-based design support tool supports design exploration in the following way:

1. The architecture is initialised by a designer by loading a number of design objects and design knowledge sources from a design knowledge base. The instantiation of the design objects effectively represents a particular part of the design space selected to be explored in more detail.
2. The designer might then start exploring the possible solutions to this particular design problem by choosing values to assign to design variables appearing in the objects and constraints to see what the consequences of such values, or combinations of values, are. This can be done by selecting a design task either through a design requirement statement window, or alternatively, through a design task menu hierarchy.
3. Any design task selected by a designer is first verified by an interface and then passed to the truth maintained blackboard control system.
4. Design knowledge sources are subsequently informed and asked if they can infer anything from these new items, together with anything already on the blackboard. Any design knowledge source can notify the blackboard control system of what it can do and what it requires from the blackboard to carry out its inference. A design knowledge source does this by proposing a KSAR which is to be placed onto the blackboard agenda, where the KSARs from all the involved design knowledge sources are queued. Justification for the proposed actions are contained in the KSAR.

5. The blackboard control system starts processing the KSARs when all the design knowledge sources become silent, i.e., when no more KSARs can be produced by any design knowledge sources. The blackboard control system executes the KSARs in the blackboard agenda one by one. The system performs inferences and maintains the justifications and the contexts of the newly derived information until further decisions need to be made.
6. The designer is then able to look at the results by displaying the interesting parts of the contents of the blackboard data, and asking the system to explain what it has done, via the graphical explanation system in order to decide what to do next. This might involve making further assumptions by creating new design routes or new design branches. In this way designers can explore alternative design solutions by making different decisions.
7. At the end of a design session, all the derived results can be graphically explained and compared with one another in their relevant design contexts.

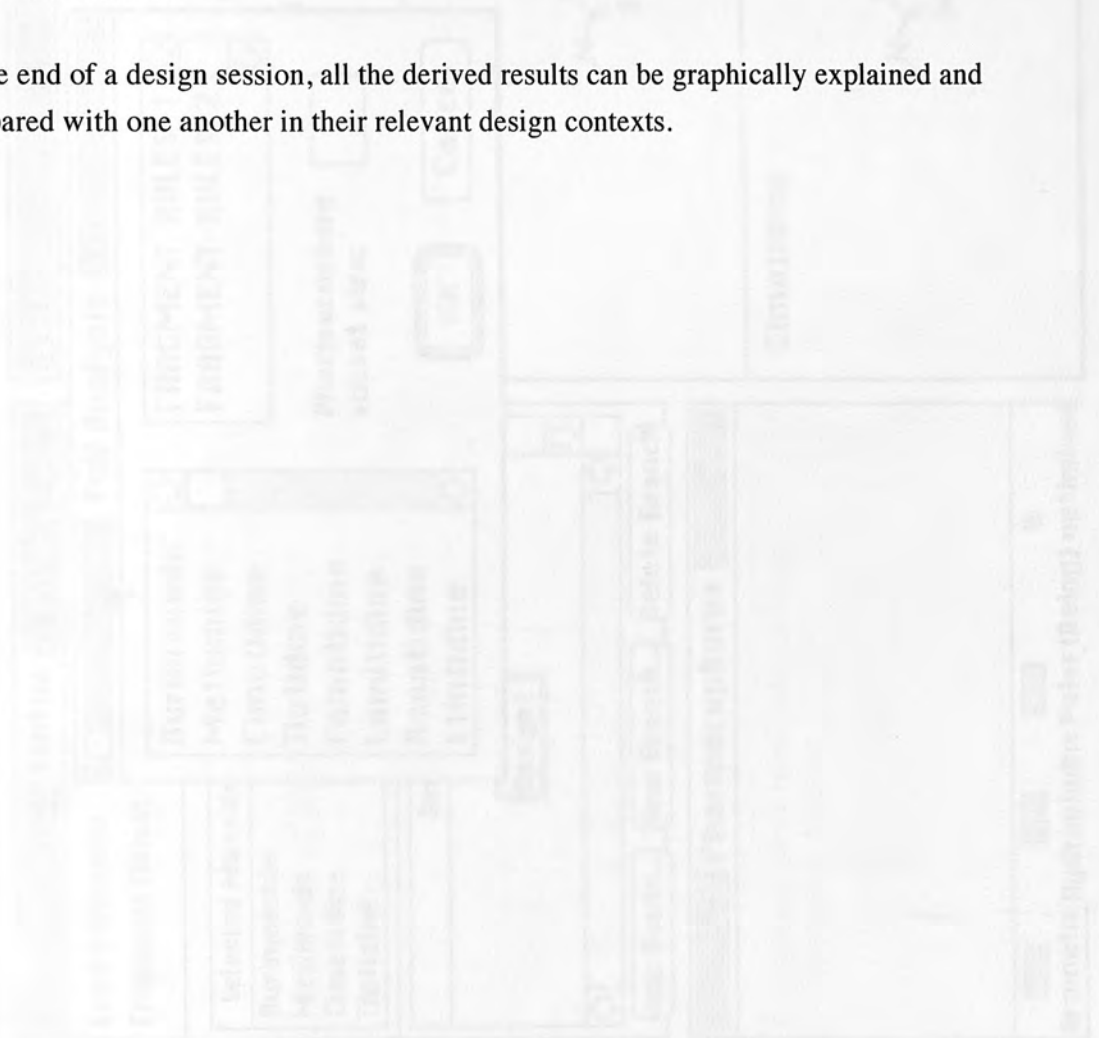


Figure 6.10: Making Decisions on How and What

Status

Lead Molecule:

Fragment Rules:

Selected Molecule:

Burimamide
Metiamide
Cimetidine
Tiotidine

Full Analysis


FRAGMENT-RULES-1

FRAGMENT-RULES-2


Pharmacophore subset size:

Molecules

7.80E-6




9.20E-7



Pharmacophores

7.90E-7

Cimetidine



Pharmacophores

0

Aromatic Hydrophobic Polar (Being) optimised

Figure 6.10: Making Decisions on Data and Method

<p>Status Unspecified</p> <p>Lead Molecule: FRAGMENT-RULES-2</p> <p>Fragment Rules: FRAGMENT-RULES-2</p> <p>Select as Lead</p>	
<p>Selected Molecules</p> <p>Lupitidine</p> <p>Famotidine</p> <p>Tiotidine</p> <p>Donetidine</p>	<p>Design Routes</p> <p>Design1 — Design2</p> <p>New Route... New Branch... Delete Branch</p>
<p>Pharmacophores</p> <p>Pharmacophore hypothesised from: Famotidine, Icotidine</p> <p>Aromatic Hydrophobic, Polar (Being) optimised 0</p>	

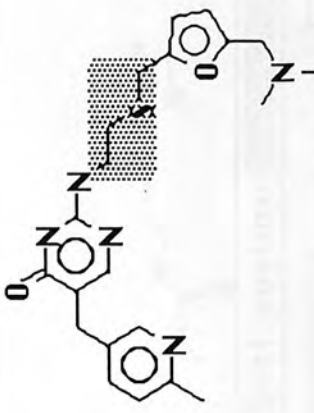


<p>Molecules</p> <p>Lupitidine 1.60E-8</p> 	<p>Famotidine 1.70E-8</p> 	<p>Tiotidine 1.80E-8</p> 
--	--	--

Figure 6.11: Creating a New Design Branch

Status Burimamide

Lead Molecule: Burimamide


Fragment Rules: FRAGMENT-RULES-1

Selected Molecules

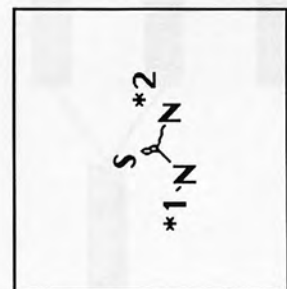
- Burimamide
- Metiamide
- Cimetidine
- Tiactidine

Molecules

Burimamide 7.80E-6



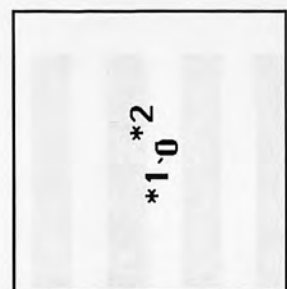
Fragment in Burimamide



*1.N*2

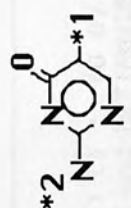
Choose best isostere

Isostere



*1.O*2

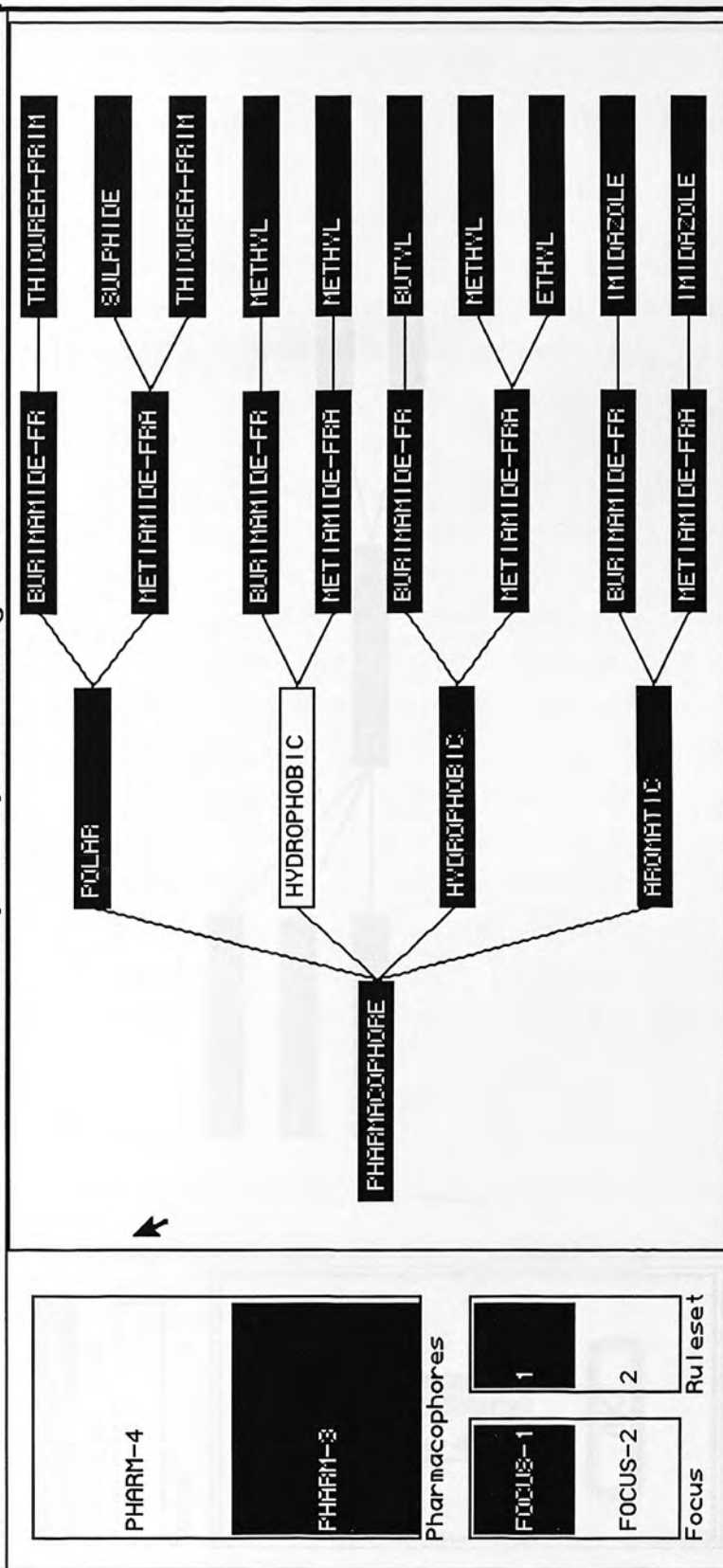
Appropriate Isosteres



*2.N*1

*1.O*2

Figure 6. 12: Designing New Drug Using an Isostere Library

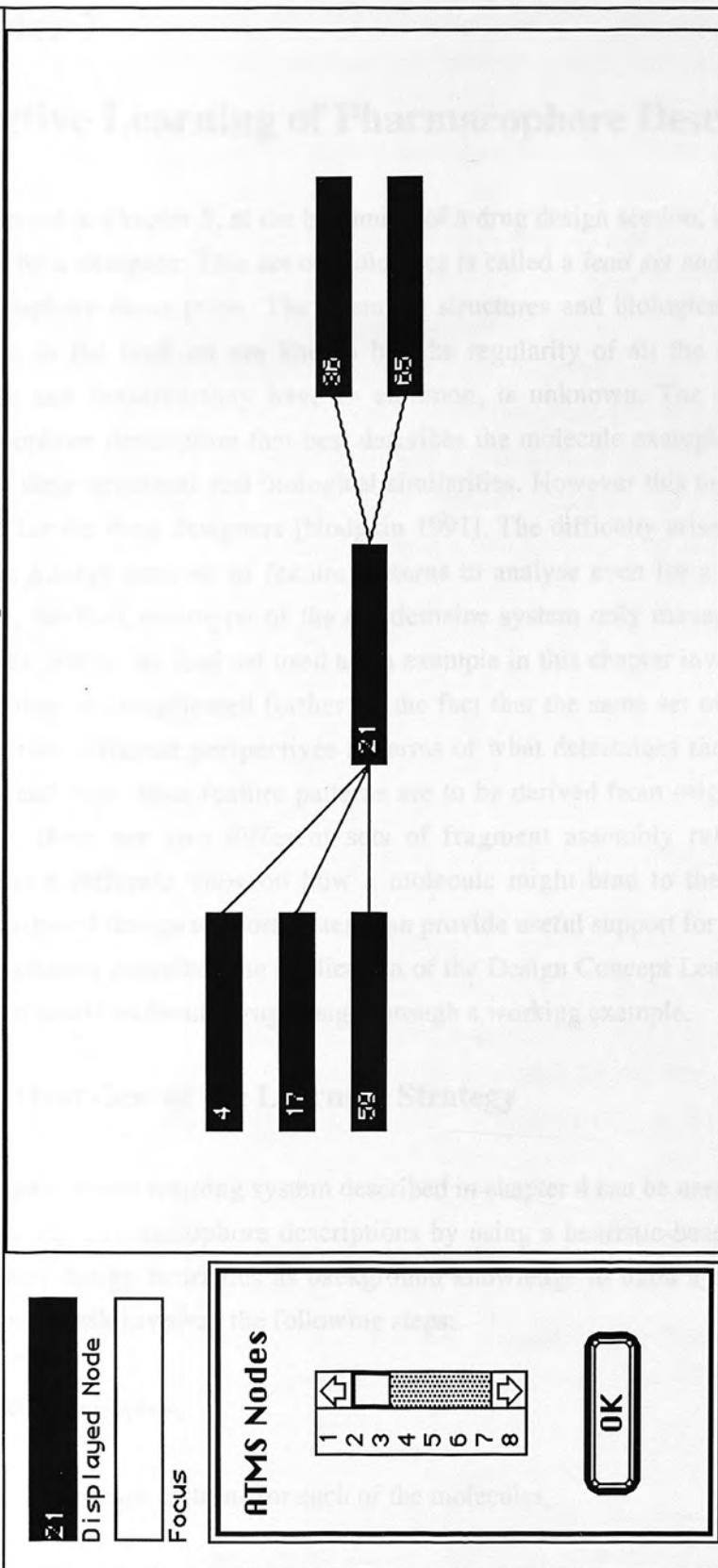


Feature: PHARM-3-FEATURE-3

Primary Property: HYDROPHOBIC
 Property Ranges: <(0 0) (0 0) (1 1) (0 0) (0 0) (0 0) (0 0) (0 0) UNUSED>
 Has Parts: BURIMAMIDE-FRAG-5 and METIAMIDE-FRAG-3.
 Connected To: PHARM-3-FEATURE-4.

Figure 6.13: Graphical Explanation of Domain Concepts

ATMB Graphic Explanation System



Node Number: 21 Type: derived Object ID: <CIMETIDINE . MOL-COMPONENTS>

Antecedents: (4 17 59)

Consequents: (36 65)

Label:

<4 <CIMETIDINE . SMILES-STRING>> <17 <FOCUS-1 . SELECTED-MOLECULES>>,
 <4 <CIMETIDINE . SMILES-STRING>> <59 <FOCUS-2 . SELECTED-MOLECULES>>.>

Figure 6.14: Graphical Explanation Justifications

Chapter 7

Inductive Learning of Pharmacophore Descriptions

As discussed in chapter 5, at the beginning of a drug design session, a set of molecules is selected by a designer. This set of molecules is called a *lead set* and is used to derive a pharmacophore description. The chemical structures and biological activities of each molecule in the lead set are known but the regularity of all the molecules, i.e., the structure and features they have in common, is unknown. The task is to derive a pharmacophore description that best describes the molecule examples in the lead set in terms of their structural and biological similarities. However this task has proved to be difficult for the drug designers [Hodgkin 1991]. The difficulty arises from the fact that there are a large number of feature patterns to analyse even for a small lead set. For example, the first prototype of the Castlemaine system only managed to work with 6 molecules, whilst the lead set used as an example in this chapter involves 37 molecules. The problem is complicated further by the fact that the same set of molecules may be viewed from different perspectives in terms of what determines the interesting feature patterns and how these feature patterns are to be derived from original molecules. For example, there are two different sets of fragment assembly rules, each of which represents a different view on how a molecule might bind to the target receptor. A computer-based design support system can provide useful support for this difficult task.

This chapter describes the application of the Design Concept Learning System in the domain of small-molecule drug design through a working example.

7.1 An Overview of the Learning Strategy

The design concept learning system described in chapter 4 can be used to support the task of generating pharmacophore descriptions by using a heuristic-based learning strategy that utilises design heuristics as background knowledge to build a design concept tree. The learning task involves the following steps:

1. Ranking examples,
2. Deriving feature patterns for each of the molecules,
3. Building a design concept tree using the feature patterns in each molecule, and
4. Evaluating the pharmacophore descriptions in the design concept tree.

The purpose of ranking molecules is to decide the order in which the molecule examples are to be supplied to the design concept learning system incrementally. This task is performed using the design heuristic knowledge about what molecules are considered by the drug designers to be more important than the others. The ranking procedure results in a *ranked lead set*.

The reason for deriving feature patterns from original molecule examples is to pre-process the input data using the knowledge that is already known in the domain. This data pre-processing task transfers the representation of the original molecules to a form suitable for the design concept learning system to derive pharmacophore descriptions useful for design purposes, i.e., connected object instances. This task involves *component partition*, *fragment assembly* and *feature pattern identification*.

The major learning task is to use the feature patterns identified from the original molecules in a lead set to develop a design concept tree incrementally until all the example molecules can be classified.

The design concept tree represents a knowledge structure in which the similarity of the molecules in the lead set has been represented. Because one molecule example typically has more than one feature pattern as a result of fragment overlapping, one molecule may be classified in the design concept tree under more than one route. This means that multiple pharmacophore descriptions can be obtained from the design concept tree. The task of evaluation is to compare the qualities of pharmacophore descriptions that can be retrieved from the design concept tree.

In the design concept tree, each pharmacophore description has a scoring value indicating its quality in terms of how well it reflects the lead set and how useful it is for design purposes. This is done by calculating a quantitative score for each of the concept nodes in the design concept tree using design heuristics. In this way, the pharmacophore descriptions in the design concept tree can be compared with each other with respect to these design heuristics.

7.2 Background Knowledge

The definition of a pharmacophore description given in chapter 5 provides a pre-defined structure on the input data (the lead set of molecules) from which the pharmacophore descriptions are to be inductively derived. That is, the construction of the design concept tree is not purely data driven. It is rather influenced by the background knowledge, which indicates that a pharmacophore description must describe the common features of all or most of the molecules in the ranked lead set, and that a pharmacophore is to be described by the feature pattern that all or most of the molecules in the lead set have in common. Here the structural and chemical relationships between the feature patterns and their

original molecules is background knowledge while the identification of the feature patterns common to all or most of the molecules is induction.

The construction of a design concept tree from which multiple pharmacophore descriptions can be obtained is the task for the design concept learning system. It is the combination of inductive inference and design heuristics as background knowledge that makes a concept in the design concept tree meaningful for this design task. The background knowledge is incorporated into the design concept learning system to specify the abstract levels of the design concept tree at which the molecule examples are to be classified.

Figure 7.1 illustrates the levels of a pharmacophore concept that are used in the development of the design concept tree as background knowledge, with the top one indicating the highest level of abstraction. For example, in a design concept tree developed on the basis of this background knowledge, the top node represents all the molecule examples in a lead set, indicating that anything under this node is derived from a particular lead set. A child node of the top node represents a unique structural description. In this particular application, this structural description is the number of the fragments in the feature patterns of the molecules that have been classified under the node. Therefore any particular node at this level can describe something like "*this pharmacophore has a feature pattern consisting of certain number of fragments*".

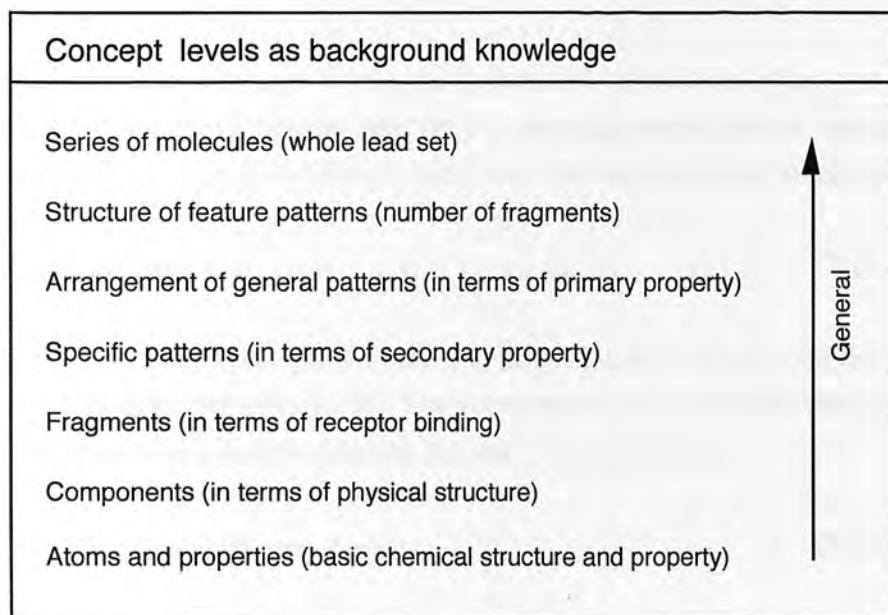


Figure 7.1: Level of Concepts

At the third level, any particular node describes a pharmacophore in terms of its primary property arrangement. For example, a node at the third level can describe

something like "a pharmacophore has certain number of fragments **and** an aromatic ring on the left, a flexible chain of hydrophobic nature in the middle, and a hydrophobic binding group on the right". At the next level below this, the nodes are distinguished by the secondary properties of the specific feature patterns that actually constitute a pharmacophore description. Each specific pattern has a number of fragments, each of which consists of one or more components with their associated chemical properties.

This background knowledge in the form of levels of concept indicates how molecules should be classified when building the design concept tree. For example, feature patterns are first classified by their structural differences, i.e., the number of fragments. Feature patterns with the same number of fragments are then further classified depending on their general feature patterns (the primary property of each fragment in the patterns and the way in which they are connected). Each general pattern will have one or more specific feature pattern; each specific pattern in turn has a number of fragments and each fragment is derived from one or more components. In this way the developed design concept tree represents clusters of the original molecules, influenced not only by the molecules themselves, but also by the levels of concept that the drug designer wish to use.

7.3 The Learning Algorithms

7.3.1 Ranking

The design concept learning system uses an incremental approach, it is therefore useful to construct an initial design concept tree that can be easily generalised or specialised. In general, the ranking value of the *i*th example E_i in a training set can be decided by

$$R_i = \sum w_j * a_j \quad (7.1)$$

where w_j is a weighting value given by the designer to the *j*th attribute A_j of the molecule E_i , and a_j is a normalised value for A_j . The normalisation for a desirable attribute and an undesirable attribute can be calculated by 7.2 and 7.3 respectively:

$$a_j = (A_j - A_{min}) / (A_{max} - A_{min}) \quad (7.2)$$

$$a_j = (A_{max} - A_j) / (A_{max} - A_{min}) \quad (7.3)$$

In the application of small-molecule drug design, the ranking attributes are selected on the basis of the following three design considerations:

- high activity;

- structural variation; and
- number of fragments.

Molecules with high activity are regarded by the drug designers as being the best descriptors of the requirements of the receptor. The activity of a small molecule is measured by the so called IC_{50} which is the 50 percent concentration in the bloodstream [Hodgkin 1991].

A pharmacophore description should describe the most active molecules in the lead set. Structural variation is measured by the number of molecular fragments of different primary properties a molecule has, with higher values indicating greater variation. The designer hopes to have greater structure variation in order to explore the pharmacophore description later on in the design stage.

Choosing a molecule which has the smallest number of molecular fragments to build the initial design concept tree ensures that it does not contain feature patterns which may not be found in other molecules in the lead set.

7.3.2 Identifying Feature Patterns

A feature pattern is an arrangement of fragments viewed by a drug designer when considering how a molecule might bind to a particular target receptor. The relationship between molecule, component and fragment has been explained in chapter 5 (see in Figure 5.5). As a result of fragment overlapping, multiple feature patterns can be identified from one single molecule.

According to the knowledge provided by the drug designers in this domain [Floyd 1990 and Hodgkin 1991], a feature pattern can be formed in three different ways:

- by adjacent fragments that are directly connected;
- by overlapping fragments that have some part of their structure in common; and
- by disjointed fragments that are not physically connected.

Any arrangement of two or more fragments within any of the above three categories constitutes a feature pattern. A single molecule may therefore give rise to a large number of such feature patterns, but not all of them are necessarily useful in deriving a pharmacophore description. To decide which of these feature patterns are meaningful, it is necessary to use design heuristics. The following two rules have been provided by the

drug designers to restrict the number of feature patterns that can be derived from a molecule:

Rule 1: A feature pattern should contain at least three or more fragments. A feature pattern consisting of only one or two fragments only represents a small portion of an original molecule. Therefore such a feature pattern is not considered to be enough to describe how the molecule might bind to a target receptor.

Rule 2. A feature pattern must contain fragments that are adjacently connected but not overlapping. That is, it must have an *unambiguous structural identity* in terms of fragment connectivity. Feature patterns connected by overlapping or disjoint fragments are considered to be too abstract.

Identifying feature patterns of each molecule example in the lead set is an essential data preparation step for the development of the design concept tree. This data preparation step reduces the complexity in the original data using heuristic design knowledge so that the design concept learning system needs only focus on those feature patterns that are more relevant to the design purposes. An algorithm has been designed to perform this task in a systematic manner. This algorithm uses a synthetic strategy to generate all the possible patterns within a molecule first and then deletes those which do not satisfy the above heuristic rules.

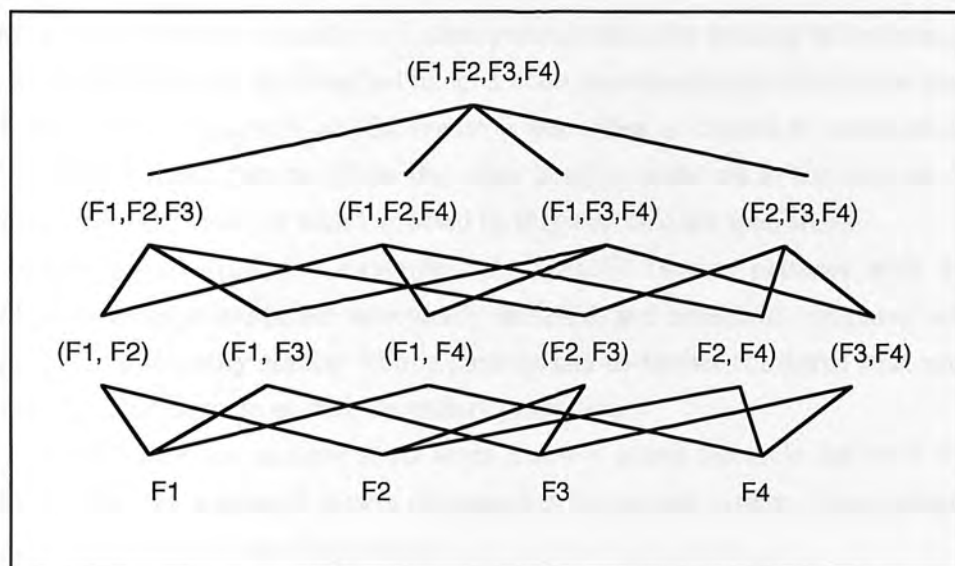


Figure 7.2: Generating Feature Patterns

For example, if a molecule has four fragments (F1, F2, F3, F4), then the following feature patterns as illustrated in Figure 7.2 will be generated first. All the patterns whose list length is less than 2 are then removed (Rule 1). This leaves a subset of {(F1, F2, F3), (F1, F2, F4), (F1, F3, F4), (F2, F3, F4), (F1, F2, F3, F4)}. If F1 and F2, for example, are overlapping fragments, then (F1, F2, F3), (F1, F2, F4), (F1, F2, F3, F4) are removed (Rule 2), leaving two feature patterns (F1, F3, F4) and (F2, F3, F4).

7.3.3 Building the Design Concept Tree

The task of identifying feature patterns that all or most of the molecules in the lead set have in common is performed by building a design concept tree using the design concept learning system described in chapter 4. Because background knowledge about the levels of concept can be used in this application, the heuristic-based learning strategy implemented in the design concept learning system is used. Using this strategy the design concept tree is initialised using the first molecule in the ranked lead set which is called the *lead molecule*. Feature patterns in the lead molecule are first classified using the *background knowledge* about the levels of concept shown in Figure 7.1.

The initial design concept tree provides a basic structure to enable the similarity of molecules in the lead set to be further sorted out. The design concept tree is further generalised when similar feature patterns in other molecules are added to those already stored in the design concept tree. The design concept tree grows as more and more feature patterns of the molecules in the lead set are added. When a new molecule is added, the learning system tries to classify its feature patterns using the existing design concept tree. If a feature pattern can be classified under a node, then the concept description associated with that node is generalised. Otherwise a new class is created to accommodate the unclassified feature pattern. If on the other hand, a molecule in the lead set is to be retreated, then the concept nodes affected by this molecule are specialised.

In this application, for example, two specific feature patterns with different topological arrangements are structurally different and cannot be compared with each other. Two structurally similar feature patterns can be further compared with each other in terms of the closeness of their secondary properties.

The generalisation process starts when a newly added molecule has been classified under a node with a general pattern that matches the general pattern of the molecule. The process involves the following 4 steps:

1. Identify a specific pattern from the newly added molecule that is most similar (in terms of single linkage distance) to the specific patterns of the molecules that are already stored in the node.

2. Store the most similar specific pattern from the newly added molecule in the node.
3. Generalise the concept description of the node by merging the secondary properties of the newly stored specific pattern and that of the specific patterns from the molecule that are already stored in the node. The merging of secondary properties uses the transformation rules discussed in chapter 4 for numerical properties such as *number of atoms* and yes/no properties such as *polar*, *planar* etc.
4. Store the newly generalised concept description in terms of secondary property ranges in the node and delete the old one.

For example, as illustrated in Figure 7.3, suppose a general feature pattern H-P-H is derived from molecule 1 and is associated with a concept node N in the design concept tree. Molecule 1 has a number of specific feature patterns ($S_{11}, S_{12}, S_{13}, S_{14}, \dots, S_{1m}$), each of which consists of a different set of fragments with different secondary properties. Suppose molecule 2 also has a general feature pattern H-P-H and some specific feature patterns ($S_{21}, S_{22}, S_{23}, S_{24}, \dots, S_{2n}$). Since these two molecules have the same general feature pattern, they can be classified under the same node N.

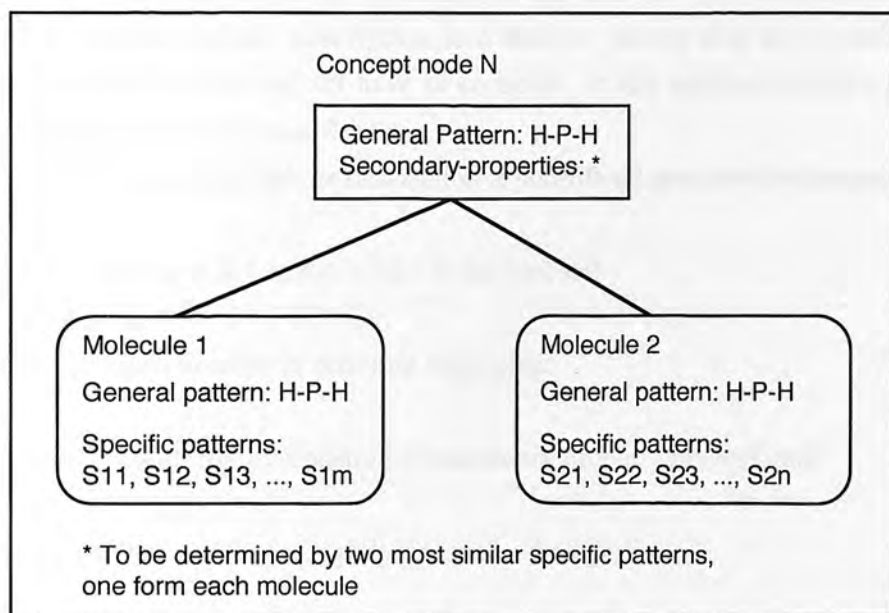


Figure 7.3: Generalisation of Secondary Properties

As a result of classifying two molecules under the same node, the concept description associated with this node needs to be generalised. The generalised concept description will have a general feature pattern (H-P-H) and a set of generalised secondary properties

using the most similar pair of specific feature patterns, one from each molecule that gives the shortest *single linkage distance*. After this generalisation, the concept associated with N1 can be stated as something like "a general feature pattern H-P-H can be found in molecule 1 and molecule 2 with the constituent specific patterns S11 and S21 (suppose S11 and S21 give the shortest single linkage distance), and the secondary property ranges of the concept are determined by the fragments in S11 and S21".

7.3.4 Concept Evaluation

The design concept tree is developed using the feature patterns identified from each molecule in the lead set incrementally. Because one molecule typically has feature patterns which may be substantially different, it will consequently be classified under more than one node, resulting in an overlapping of groups of concepts. In other words, there are multiple pharmacophore descriptions in the design concept tree.

In order to help the designers to know which node in the design concept tree is the best pharmacophore description, it is necessary to have some criteria for evaluating the nodes in the design concept tree.

A general and important criterion is that a concept node is considered better than the others if it covers more molecule examples in the lead set. This is true because by definition a pharmacophore description is a feature pattern that all or most of the molecule examples in the lead set have in common. In this application there are other domain dependent issues to consider too.

For example, a pharmacophore description is considered good by the designers if

- it describes the most active molecules in the lead set;
- it has the greatest number of different fragments;
- it has features with tightly-constrained secondary property ranges¹; and
- it has the greatest number of tightly-constrained properties.

Based on these considerations an additional *scoring scheme* is devised to give each node in the design concept tree a numerical value to measure its *goodness and fit* in terms of the above domain dependent considerations. The node in the design concept tree that scores the highest value can be chosen as the best pharmacophore description.

¹ An attribute value between [1 to 4] is considered tighter than a value between [0 to 5].

The score of a concept node i is calculated using (7.4) - (7.5).

$$Q_i = c_1 * x_{i1} + c_2 * x_{i2} + c_3 * x_{i3} + c_4 * x_{i4} \quad (7.4)$$

$$c_1 + c_2 + c_3 + c_4 = 1 \quad (7.5)$$

where c_1 , c_2 , c_3 , and c_4 are the weights specified by the designer for average activity, structural variation, example occurrence frequency, and secondary property distance respectively, x_{i1} , x_{i2} , x_{i3} , x_{i4} are normalised attribute values representing activity, number of different fragments, number of instances, and secondary property distance respectively.

The first factor x_{i1} gives node i an amount in terms of its pharmacological activity. The second factor x_{i2} measures the structural variation by counting the number of different fragments that have been classified under node i , which indicates that a larger number of different fragments within a pharmacophore description is a better data set for structure-activity relationship analysis in design. The third factor x_{i3} is the base rate at which examples are classified under node i compared with the total number of molecules in the lead set, with a larger value indicating that the node represents more of the training examples. The fourth factor x_{i4} is the sum of the distances calculated when generalising the concept description stored in node i . This factor gives a measure of the closeness of feature patterns classified under the node in terms of their secondary properties.

7.4 A Working Example

This section presents a working example of identifying pharmacophore descriptions to demonstrate how the learning approach works in this complex design problem.

In this example, 37 molecules in a series named GDS [Floyd 1990 and Hodgkin 1991] are used. Some of these molecules are shown in Figure 7.4, in which each molecule is shown with its two-dimensional chemical structure, its internal representation (Smiles string) and its activity value. The input to the design concept learning system is a selected lead set of molecules, and the output of the learning process is a design concept tree from which different levels of concept about the lead set molecules can be obtained.

For each molecule in the lead set, component partition and fragment assembly are performed first. This results in the original examples being transferred into instances of object class *drug* with their molecular components and molecular fragments identified. For example, after component partition and fragment assembly, the representation of the molecule named gds3 is shown in Figure 7.5.

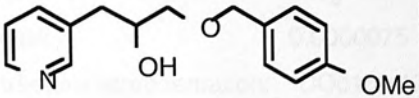
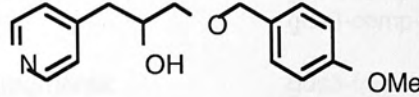
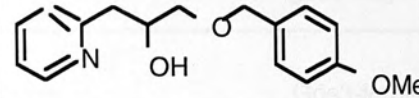
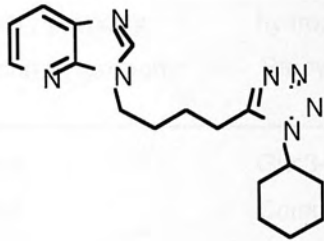
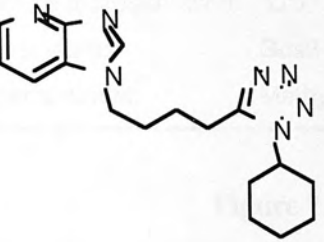
No.	Molecules	Structural representation	Activity
1		<chem>COc1ccc(COCC(O)Cc2cccnc2)cc1</chem>	158
2		<chem>COc1ccc(COCC(O)Cc2ccncc2)cc1</chem>	750
3		<chem>COc1ccc(COCC(O)Cc2ccccc2)cc1</chem>	166
		
36		<chem>c12cccc1ncn2CCCCc3nnnn3C4CCCC4</chem>	118
37		<chem>c12cccnc1ncn2CCCCc3nnnn3C4CCCC4</chem>	112

Figure 7.4: Molecule Examples

Name:	Gds3
Class:	Drug
Activity:	0.0000075
Structure representation:	"COc1ccc(COCC(O)Cc2cccn2)cc1"
Components:	gds3-comp-8 □gds3-comp-7 □gds3-comp-6 □gds3-comp-5 gds3-comp-4 □gds3-comp-3 □gds3-comp-2 □gds3-comp-1
Fragments:	gds3-frag-12 gds3-frag-11 gds3-frag-10 gds3-frag-9 gds3-frag-8 gds3-frag-7 gds3-frag-6 gds3-frag-5 gds3-frag-4 gds3-frag-3 gds3-frag-2 gds3-frag-1
Name:	Gds3-frag-1
Class:	Fragment
Has-parts:	Gds3-comp-6
Primary-property:	hydrophobic
Secondary-property:	Propyl
Name:	Gds3-comp-1
Class:	Component
Atom-number:	0
Structure representation:	"C*7"
Connected-to:	Gds3-comp-3
Property-model:	Methyl

Figure 7.5: Input Data Structure

In order to decide which molecule in the lead set to use to initialise the concept tree, all the molecules are first ranked using the ranking algorithm described in Section 7.3.1. Figure 7.6 lists the ranking results for the 37 molecules in the lead set.

The first molecule in the ranked lead set, named *gds3*, is used as the *lead molecule* to create an initial design concept tree. All the feature patterns in the lead molecule are then identified using the algorithm described in Section 7.3.2. For *gds3* there are a total of 29 specific patterns which can be classified into 20 classes (general patterns).

Molecule	Ranking value	Concept Explanation	Activity	Number of fragments
6083	0.9523571428571428		0.0000075	12
6089	0.8562224352828379		0.00000525	10
60814	0.5506222023010547		0.0000046	9
60829	0.5394534925206136		0.000005	8
60818	0.5188590604026846		0.00000215	13
6082	0.4672195539645254		0.0000021	12
60821	0.4451677852348994		0.000005	6
60822	0.4451677852348994		0.000005	6
6081	0.4204544582938845		0.00000158	12
60836	0.403744966442953		0.00000087	13
60812	0.3918197507190796		0.00000231	10
6088	0.3711351869606904		0.00000208	10
60817	0.3704697986577181		0.0000005	13
60810	0.3333633748801534		0.00000166	10
6086	0.3107363374880153		0.00000086	12
60815	0.2853211888782359		0.00000165	9
60819	0.2823571428571429		0.00000005	12
60813	0.261039309683605		0.00000138	9
6087	0.249271322694151		0.00000071	10
60811	0.2380345158197507		0.0000006	10
6084	0.2317392138063279		0.00000053	10
60816	0.2088782355581016		0.00000008	9
60833	0.189093009587728		0.00000058	9
60834	0.1630124640460211		0.00000029	9
60836	0.1487670182166826		0.00000118	7
60837	0.1433710450623202		0.00000112	7
60820	0.1214093959731544		0.00000014	6
6085	0.1172905081495685		0.00000083	7
60830	9.30086285492768E-2		0.00000056	7
60823	8.723489932885908E-2		0.00000102	6
60832	7.232406519654842E-2		0.00000033	7
60831	4.984084372003834E-2		0.00000008	7
60825	3.867114092959732E-2		0.00000048	6
60827	3.417449664429531E-2		0.00000043	6
60824	1.348993288590604E-2		0.0000002	6
60843	1.248993288590604E-2		0.0000002	6

Figure 7.6: Molecule Ranking Result

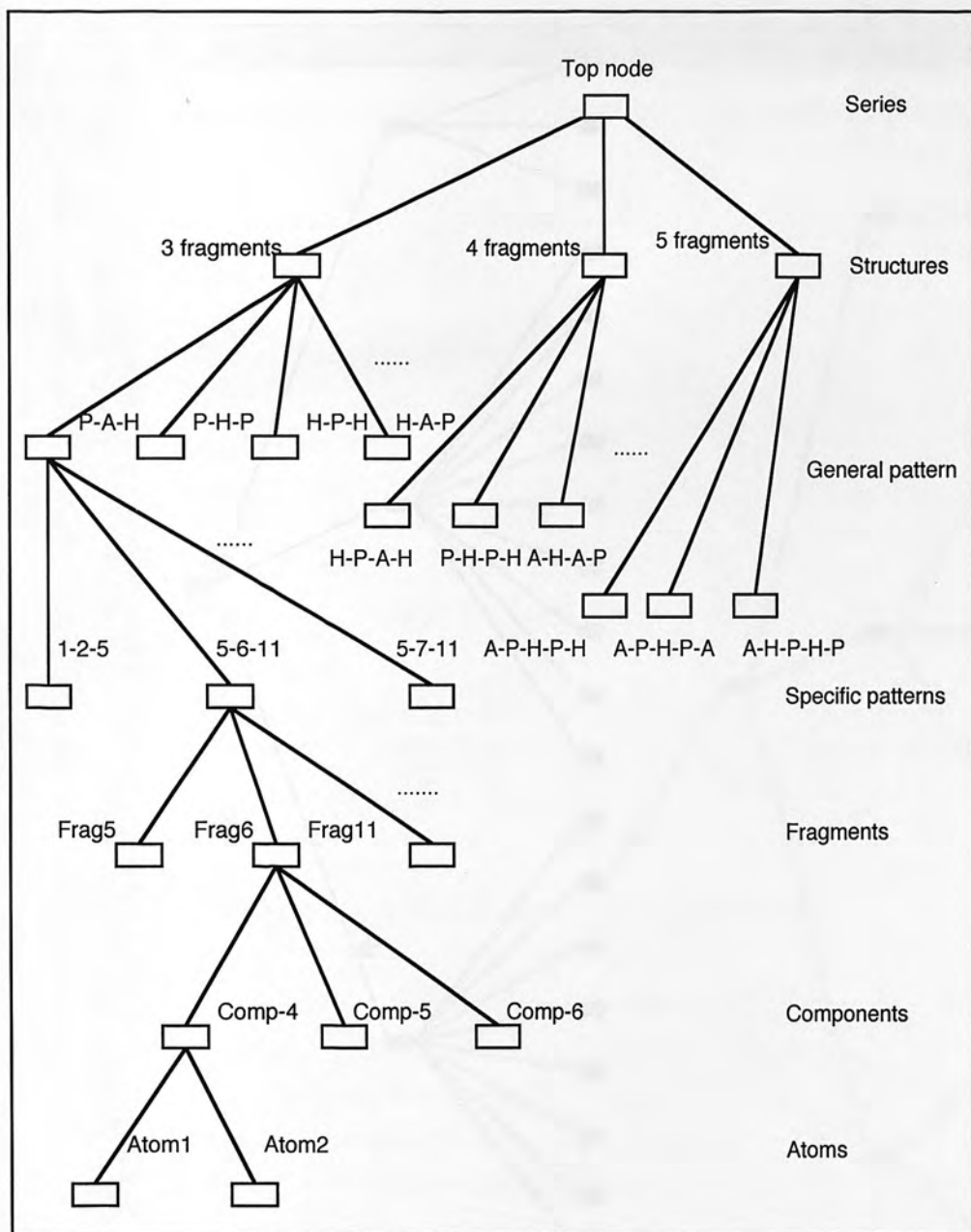


Figure 7.7: Initial Design Concept Tree

Figure 7.7 illustrates the initial design concept tree created using the feature patterns in the first molecule (gds3) in which each node has an initial concept description associated with it. For example, the node labelled as "3 fragments" indicates that any instances classified under it have *three-fragment* patterns whilst the node labelled P-A-H indicates that it have a general feature pattern consisting of three fragments whose primary properties are polar (P), aromatic (A) and hydrophobic (H) respectively. Each general pattern has a number of specific patterns.

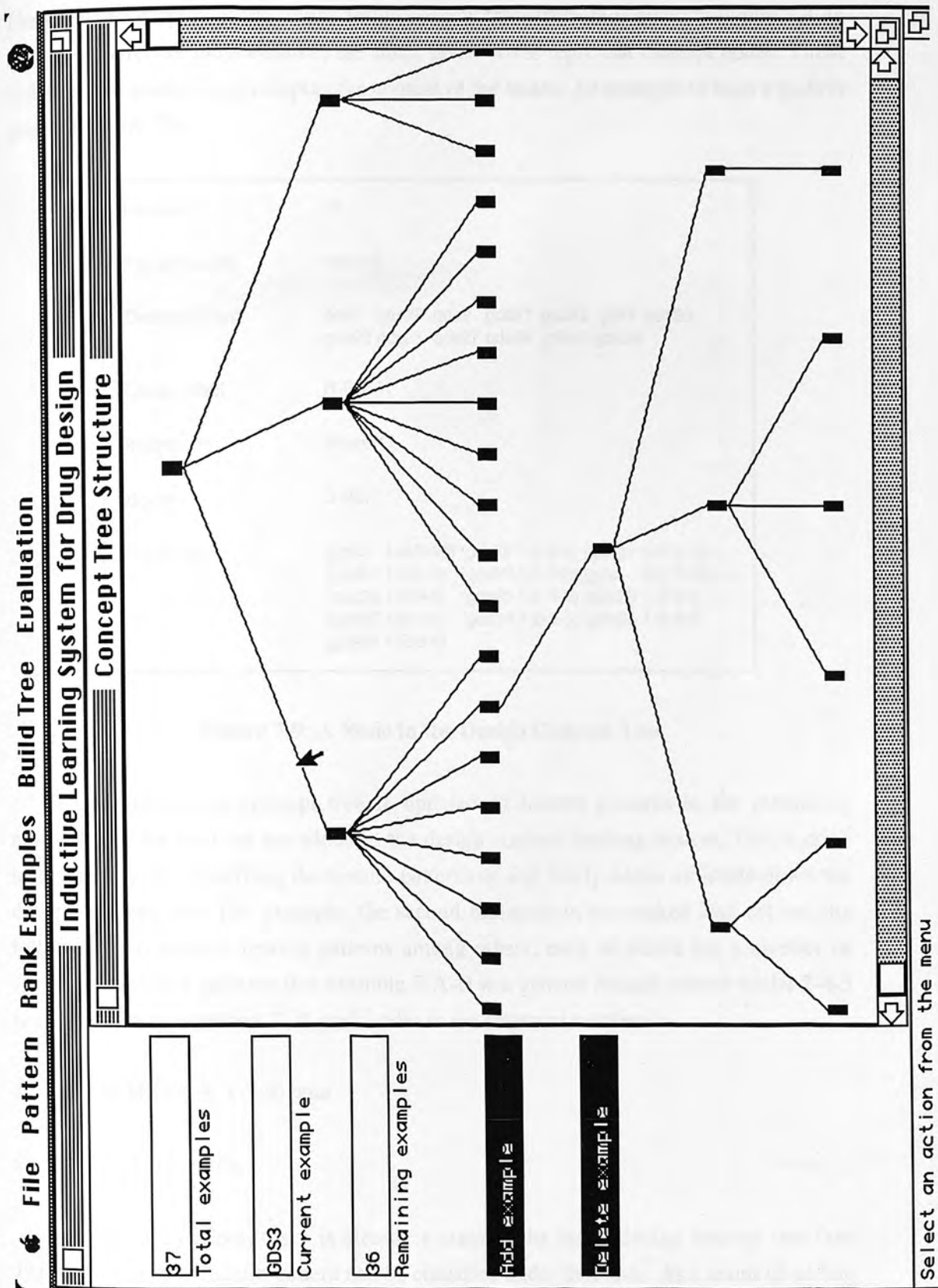


Figure 7.8: Display of the Initial Design Concept Tree

The graphical display of the initial design concept tree structure is given in Figure 7.8. In the concept tree display window, the small black boxes represent concept nodes. These nodes can be clicked on to display the content of the nodes. An example of such a node is given in Figure 7.9.

Node-id	14
Parent-node	Node-2
Derived-from	gds3 gds29 gds2 gds21 gds22 gds1 gds20 gds23 gds25 gds27 gds24 gds26 gds28
Class-label	H-P-A-H
Instance	Pharm11
Score	0.303
Patterns	(gds3 1-2-5-10) (gds29 1-2-5-8) (gds2 1-2-5-10) (gds21 1-2-3-5) (gds22 1-2-3-5) (gds1 1-2-5-10) (gds20 1-2-3-5) (gds23 1-2-3-5) (gds25 1-2-3-5) (gds27 1-2-3-5) (gds24 1-2-3-5) (gds26 1-2-3-5) (gds28 1-2-3-5)

Figure 7.9: A Node in the Design Concept Tree

The initial design concept tree is updated as feature patterns in the remaining molecules in the lead set are added to the design concept learning system. This is done incrementally by classifying the feature patterns of any newly added molecule down the design concept tree. For example, the second molecule in the ranked lead set has the following two general feature patterns among others, each of which has a number of specific associated patterns (for example P-A-H is a general feature pattern whilst 2-4-5 is a special feature pattern, 2, 4, and 5 refer to the fragment numbers):

1. P-A-H (2-4-5, 1-7-9), and
2. H-A-H (2-3-10).

For the first pattern, there is already a class in the initial design concept tree (see Figure 7.7), so this feature pattern can be classified under that node. As a result of adding a new feature pattern into a node, the concept associated with that node (the initial pharmacophore description) needs to be generalised using the method described in Section 7.3.3 (see Figure 7.3).

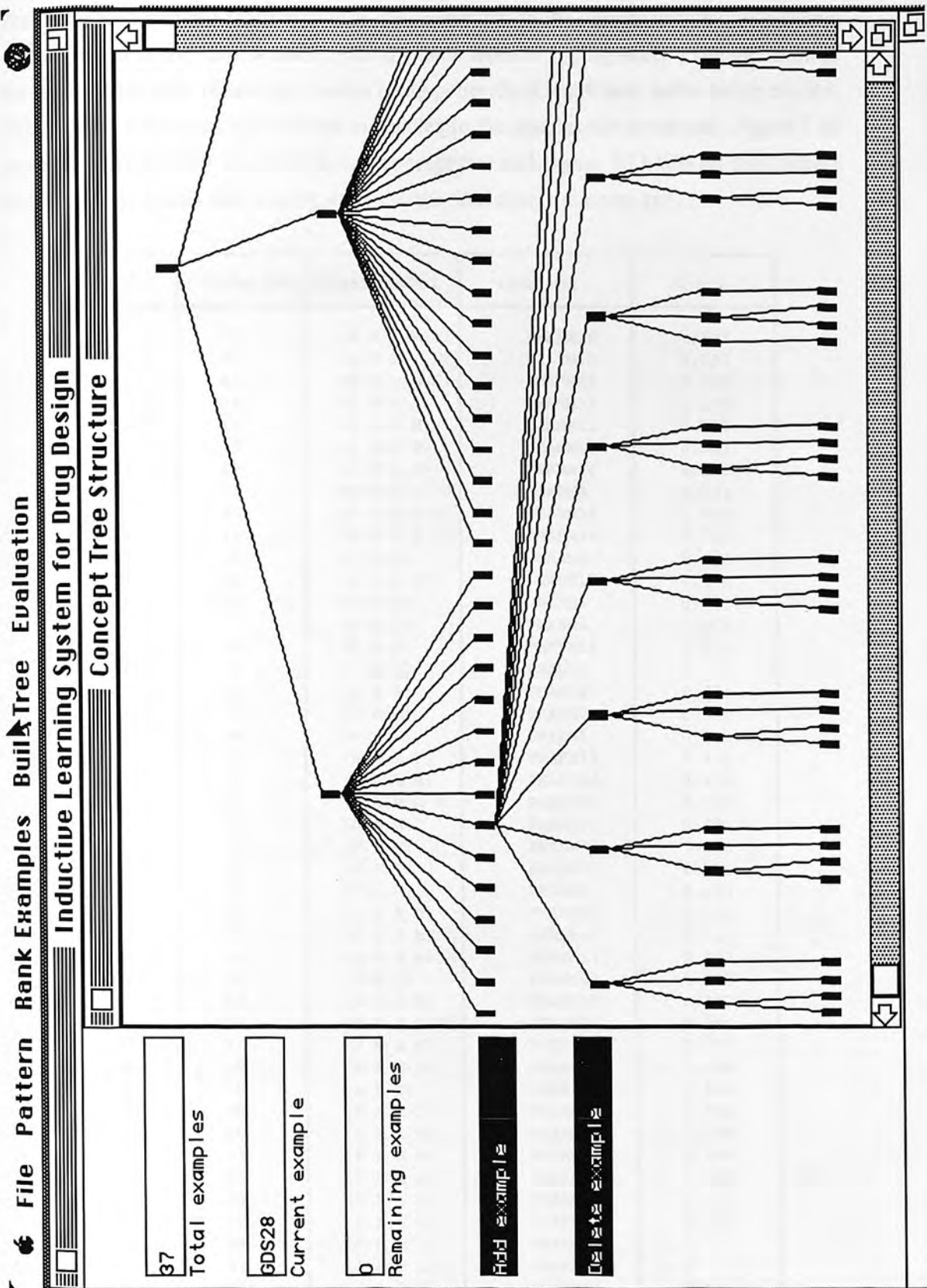


Figure 7.10: The Display of the Final Design Concept Tree

For the second general feature pattern, no class exists in the current design concept tree. So the system creates a new node under the node labelled "3 fragments". The inclusion of the second molecule results in 8 nodes being generalised and 9 new nodes being created. This process continues until all the molecules in the lead set are processed. Figure 7.10 shows the structure of the final design concept tree and Figure 7.11 lists all the concept nodes, concept labels and scoring values in the final design concept tree.

Rank	Node no.	Class label	Concept	Score
1	22	(A P P H A)	PHARM19	0.643
2	21	(A P H P H)	PHARM18	0.592
3	42	(H H P H A)	PHARM40	0.589
4	16	(P H H A)	PHARM13	0.560
5	15	(P H P H)	PHARM12	0.550
6	19	(P H H P)	PHARM16	0.541
7	17	(A H A P)	PHARM14	0.540
8	11	(H A P)	PHARM8	0.531
9	23	(A P P H H)	PHARM20	0.509
10	41	(H H P H P)	PHARM39	0.500
11	9	(H P H)	PHARM6	0.471
12	18	(A P H P)	PHARM15	0.468
13	6	(P H P)	PHARM3	0.455
14	7	(H H P)	PHARM4	0.453
15	24	(H A H)	PHARM21	0.411
16	5	(P H A)	PHARM2	0.397
17	46	(A H A)	PHARM43	0.384
18	10	(H H A)	PHARM7	0.372
19	4	(P A H)	PHARM1	0.372
20	40	(A H H P)	PHARM38	0.361
21	46	(H A A H)	PHARM44	0.358
22	31	(P A P A H)	PHARM28	0.357
23	25	(A A H)	PHARM22	0.356
24	37	(P P H)	PHARM34	0.351
25	33	(H P A)	PHARM30	0.335
26	8	(P H H)	PHARM5	0.333
27	28	(P A A H)	PHARM25	0.320
28	37	(H P P H)	PHARM35	0.312
29	14	(H P A H)	PHARM11	0.303
30	26	(A H P)	PHARM23	0.302
31	39	(H A P H)	PHARM37	0.302
32	32	(P A P H H)	PHARM29	0.300
33	27	(P H A H)	PHARM24	0.296
34	29	(A P H H)	PHARM26	0.289
35	38	(A P H)	PHARM36	0.289
36	30	(P A H P)	PHARM27	0.288
37	20	(A P A H)	PHARM17	0.258
38	13	(P P H H)	PHARM10	0.200
39	35	(H P H A)	PHARM32	0.181
40	34	(H P H P)	PHARM31	0.181
41	12	(P P H A)	PHARM9	0.173
42	44	(P H P A)	PHARM42	0
43	47	(A A P H)	PHARM45	0
44	43	(P A P H)	PHARM41	0

Figure 7.11: List of Final Concept Nodes

The final design concept tree gives a complete picture of the example molecules in terms of their similarities in structure and property. Each node in the concept tree has a class label, and a class description. All together these concept descriptions provide multiple solutions to the task of pharmacophore identification.

Figure 7.9 displays the information associated with a particular node (node 14) in the design concept tree. This node stores all the feature patterns that have been used to generalise the pharmacophore description using a nearest neighbour strategy. The concept description, i.e., the pharmacophore description associated with this concept node (node 14), is illustrated in Figure 7.12.

It is necessary to evaluate the relevant importance of the nodes in the design concept tree in order to select the one that is most useful for design purposes. Using the method described in section 7.3.4, each node is given a numerical value indicating the quality of the concept node as listed in Figure 7.13.

This pharmacophore description has four features (features 1 and 2 are hydrophobic, feature 2 is polar and feature 3 is aromatic). Each feature has fragments from those molecule examples that have been used to generalise this pharmacophore description. Each fragment in turn has a number of molecular components, whose chemical properties can be identified through their property models. In this way a pharmacophore description gives a complete picture of which lead set of molecules it is derived from, how many molecules in the lead set are covered by this pharmacophore description, how many fragments the pharmacophore description has, the general features of the pharmacophore in terms of primary property, and what the properties of the feature patterns are.

This concept scoring method attempts to use the information that may be available in the form of a design requirement such as weighting values, to measure the quality of the concepts learned. If, for example, a design requirement is to "find the pharmacophore descriptions that commonly exist in most of the example molecules", the scores of the concept nodes can be re-calculated by setting all the weights, except the one concerning the number of example molecules, to zero. This gives the highest score to the concept node that is derived from the maximum number of example molecules as illustrated in Figure 7.13. In this case node 7 scores the highest value as it can be observed in 36 out of 37 molecules.

The frequency in Figure 7.13 is calculated by dividing the number of molecules containing the feature patterns in the pharmacophore description by the total number of molecules in the lead set. For example, the first pharmacophore description covers 36 molecules whilst the second covers 31 molecules.

File Pattern Rank Examples Build Tree Evaluation
 Inductive Learning System for Drug Design
 Concept Explanation

Name of pharmacophore model PHARM11:
 Pattern: H-F-A-H
 Derived From: GOS3.

H: hydrophobic P: polar A: aromatic

	feature 1	feature 2	feature 3	feature 4
primary-property	(NO)	(NO)	(YES)	(NO)
aromatic	(NO)	(NO)	(NO)	(YES)
flexible	(YES)	(NO)	(NO)	(YES)
hydrophobic	(0 3)	(0 2)	(0 6)	(0 3)
no-of-atoms	(0 0)	(0 2)	(0 1)	(0 0)
no-of-h-bond-donors	(0 0)	(0 1)	(0 0)	(0 0)
planner	(0 0)	(0 0)	(0 0)	(0 0)

Figure 7.12: A Pharmacophore Description

Rank	Node no.	Class label	No. of examples	Frequency	Score
1	7	(H H P)	36	0.972	1.0
2	6	(P H P)	31	0.837	0.857
3	5	(P H A)	31	0.837	0.857
4	4	(P A H)	21	0.567	0.571
5	26	(A H P)	13	0.351	0.342
6	25	(A A H)	13	0.351	0.342
7	14	(H P A H)	13	0.351	0.342
8	37	(P P H)	11	0.297	0.285
9	24	(H A H)	11	0.297	0.285
10	8	(P H H)	11	0.297	0.285
11	35	(H P H A)	11	0.297	0.285
12	34	(H P H P)	11	0.297	0.285
13	20	(A P A H)	11	0.297	0.285
14	39	(A P H)	10	0.270	0.257
15	33	(H P A)	10	0.270	0.257
16	38	(H P P H)	10	0.270	0.257
17	13	(P P H H)	10	0.270	0.257
18	12	(P P H A)	10	0.270	0.257
19	41	(A H H P)	9	0.243	0.228
20	10	(H H A)	8	0.216	0.2
21	40	(H A P H)	8	0.216	0.2
22	29	(A P H H)	8	0.216	0.2
23	28	(P A A H)	8	0.216	0.2
24	27	(P H A H)	8	0.216	0.2
25	32	(P A P H H)	8	0.216	0.2
26	31	(P A P A H)	8	0.216	0.2
27	30	(P A H P)	7	0.189	0.171
28	9	(H P H)	6	0.162	0.142
29	46	(A H A)	5	0.135	0.114
30	47	(H A A H)	5	0.135	0.114
31	11	(H A P)	4	0.108	0.085
32	19	(P H H P)	4	0.108	0.085
33	18	(A P H P)	3	0.081	0.057
34	17	(A H A P)	3	0.081	0.057
35	16	(P H H A)	3	0.081	0.057
36	15	(P H P H)	3	0.081	0.057
37	43	(H H P H A)	3	0.081	0.057
38	42	(H H P H P)	3	0.081	0.057
39	23	(A P P H H)	3	0.081	0.057
40	22	(A P P H A)	3	0.081	0.057
41	21	(A P H P H)	3	0.081	0.057
42	48	(A A P H)	1	0.027	0
43	45	(P H P A)	1	0.027	0
44	44	(P A P H)	1	0.027	0

Figure 7.13: Relationship Between Examples and Concepts

If the number of molecules that a pharmacophore covers is the only criterion for evaluating the appropriateness of the learning results, then the first pharmacophore description in Figure 7.13 can be chosen as the best because it describes the features that all or most of the molecules in the lead set have in common.

The concept nodes illustrated in Figure 7.10 represent the result of considering all the evaluation criteria as equally important (the score is calculated by giving equal weights to all the evaluation factors, i.e., average activity, number of different fragments, closeness of patterns in terms of secondary properties, and the number of examples that contribute to the concept).

If an example molecule is later found irrelevant to the lead set, then a specialisation can be performed by deleting that molecule from the concept tree. This may happen whilst investigating the influence of individual molecules on the pharmacophore description learned. In this case the concept tree will need to be specialised by excluding some of the nodes whose scores are below a certain value threshold. For example, the nodes (node 45, 44, and 48) with the smallest scores in Figure 7.13 contain feature patterns that can be found in only one molecule. Therefore these nodes can be deleted, as they do not represent the majority of the molecule examples in the lead set. If a particular molecule produces many of these *low quality* concepts, it can be concluded that the molecule is significantly different from the majority of the molecules in the lead set.

Summary

In this chapter, an example has been used to demonstrate how the design concept learning system described in chapter 4 can be used to support one of the key tasks in the domain of small molecule drug design.

In this application, the designer starts a drug design session by selecting a set of molecules from a molecule knowledge base. This forms a lead set. The rules about component partitioning, fragment assembly, and feature pattern identification can then be invoked to construct a set of feature patterns for each of the molecules in the lead set. The designer can then select the attributes and define the weights for the ranking procedure. The ranking based on these attributes and weights forms a ranked lead set.

The designer can then start the design concept learning system to classify each molecule example in the ranked lead set using the background knowledge in terms of levels of concept to be developed. This task is performed incrementally. The result of this learning process is a design concept tree from which multiple pharmacophore descriptions can be retrieved.

An evaluation program can then be invoked to calculate a score for each of the nodes in the design concept tree. The design can adjust the weights given to the evaluation program. The designer can also delete some of molecules from the lead set if they are later considered irrelevant to the drug being designed.

The best pharmacophore description selected from the design concept tree is put forward for the specification of a new molecule using the isostere replacement approach

described in chapter 5. When a new molecule is specified it can be put back into the knowledge base for future use.

The design concept learning system and the architecture developed in this thesis has been successfully used to support the problem of small molecule drug design. Chapter 8 will present an evaluation of the architecture and the design concept learning system within and beyond this domain.

During the design process, a strong commitment to design knowledge acquisition and the integrated application of AI techniques to design problem modelling and design problem solving at various stages of the design process. This thesis has addressed the following issues concerning knowledge-based design support and knowledge engineering:

- investigating the role of inductive inference in the design process;
- modelling the early stage design process as learning and exploiting the design problem structure;
- developing a knowledge-based architecture for design support;
- integrating a design concept learning system with the architecture; and
- supporting full design tasks using the implemented architecture and the design concept learning system.

In the course of this research, this thesis developed an integrated software system that implements computer-based design techniques to support early stage design activities:

1. a knowledge-based design support system architecture for knowledge-based design problem modelling; and

2. a design concept learning system for user-guided design support.

The architecture and the design concept learning system have been implemented in a knowledge-based environment and applied to a complex design problem, i.e. small molecule drug design. In this chapter, the integrated techniques and the design concept learning system are evaluated within the domain of small molecule drug design. Their limitations and their applicability beyond this particular domain are then discussed. Finally, some suggestions for further research on the topic of knowledge-based design support are proposed.

Chapter 8

Evaluation and Future Directions

The development of a knowledge-based design support system can benefit from a good understanding of the design process, a strong commitment to design knowledge acquisition, and the integrated application of AI techniques in design product data modelling and design problem solving at various stages of the design process. This thesis has addressed the following issues concerning knowledge-based design support and inductive learning:

- understanding the role of inductive inference in the design process,
- modelling the early stage design process as learning and exploring the design problem structure,
- developing a knowledge-based architecture for design support,
- integrating a design concept learning system with the architecture, and
- supporting real design tasks using the implemented architecture and the design concept learning system.

As a result of the research, this thesis developed an integrated software kernel that provides two computer-based design techniques to support early stage design activities:

1. A knowledge-based design support system architecture for knowledge-based design system development; and
2. A design concept learning system for conceptual design support.

Both the architecture and the design concept learning system have been implemented in a Lisp-based environment and applied to a complex design problem, i.e., small molecule drug design. In this chapter, the integrated architecture and the design concept learning system are evaluated within the domain of small molecule drug design. Their limitations and their applicability beyond this particular domain are then discussed. Finally the directions for further research on the topic of knowledge-based design support and inductive learning are proposed.

8.1 Evaluation of the Architecture

8.1.1 Features of the Architecture

To develop a knowledge-based design support system is a difficult task. The difficulty arises mainly from the fact that the acquisition and representation of domain and design knowledge needs combined efforts from both designers and knowledge engineers. The software development of a knowledge-based design support system is also a lengthy and costly process because various computational techniques necessary for design support are not readily available in an integrated environment.

The ATMB architecture developed in this thesis was fully integrated in the Castlemaine drug design system. Since the Castlemaine project this architecture has been developed by the author as an integrated software kernel that can be used to develop knowledge-based design application systems. This architecture integrates a number of AI-based sub-systems including, mainly, an object-oriented scheme for representing domain and design knowledge, a blackboard system for intelligent control of the design support process, a design context management system for the exploration and maintenance of multiple contexts of design, a design documentation system for recording design results and design history, and a graphical user interface for interacting with the users and for explaining the results generated by the system.

The integration of a blackboard control system and an assumption-based truth maintenance system in the ATMB architecture provides an effective mechanism for the exploration and management of multiple contexts of design, which is absent from many of the design systems such as HOBS, IFe, DICE and IDF [Carter *et al* 1991, Clarke *et al* 1991, Sriram *et al* 1992 and Ball *et al* 1992]. This mechanism is suitable for design support because the design decision process is typically *nonmonotonic*, i.e., some later decisions may contradict the earlier ones.

While some of the design systems reviewed in this thesis provided a blackboard style of control system for design support, the following features are uniquely associated with the architecture developed in this thesis:

- It combines rule-based and object-oriented knowledge representation for the development of design knowledge bases and design knowledge sources.
- It has been designed and implemented as a self-contained AI software kernel with basic and essential components for intelligent design support.

- It provides flexibility for the designer to explore multiple contexts of design making different design decisions (even conflicting decisions). A design context can be created by any member of a design team making assumptions. The design context management system maintains the justifications of any derivations based on these assumptions.
- It has the potential for concurrent engineering design. The knowledge on the blackboard can be shared by members of a design team and explored from different perspectives. Negotiation can be delayed until such time as it is necessary to evaluate a number of alternative design contexts (or solutions). The negotiation decisions can also be added to the system as *assumptions*, resulting in some of the undesirable design results becoming eliminated.
- It provides facilities for documenting design results and design history that can be replayed. Any derived knowledge can be graphically explained using the information kept in the design document.
- It supports conceptual design using a design concept learning system that can derive a basic structure of the design problem from raw data or design examples when such a structure is not readily available as a design prototype.

The ATMB architecture has been tested in the domain of small molecule drug design. It has proved to be effective in this domain and it supported the task of identifying alternative design solutions (pharmacophore descriptions) [Smithers *et al* 1992 and 1993a and 1993b].

8.1.2 Limitations for the Architecture

Definition of a Design Context: In the current implementation of the design context management system, a design context is defined by means of a constraint-based approach to design. That is, a design context is defined from a domain independent basis, i.e., on the basis of the *design data*, the *design method (constraints)*, and the *design variable values*. The difficulty arises from the fact that some issues, such as the question of what forms the context of a design, and how the designers explore design contexts, are not necessarily domain independent issues. A more general design context definition should take into account a wider range of issues such as *the design process*, *the design product data model*, *the manufacturing constraints*, *the materials*, and *the user type* etc. The definition of a design context in the current design context management system needs to be extended in order to support design applications where multiple solutions need to be

explored simultaneously by a team of designers, each of which may have different priorities and concerns.

The Partition of the Blackboard: The architecture developed in this thesis does not provide an explicit partition of the blackboard like other design systems such as HOBS, IFe, DICE and IDF [Carter *et al* 1991, Clarke *et al* 1991, Sriram *et al* 1992 and Ball *et al* 1992]. The current implementation of the ATMB architecture only supports an implicit partition of the information available on the blackboard by means of creating design contexts. It is possible to create more than one blackboard within the current implementation of the architecture because the blackboard itself is an object class. However, the issue concerning the co-operation between different blackboard instances has not been addressed. This may appear to be over restrictive for systems in which different software packages must operate on different data structures.

User Modelling: In the current implementation of the architecture, all user actions are treated as *assumptions* upon which the system's inferences depend. It is therefore unable to deal with different types of users, each of which may be concerned with only one part of an integrated system and may have to carry out domain specific design tasks using the data created by others involved in the design process. The inability of the current architecture to model different types of user restricts its application in concurrent engineering design, where it is customary for different types of users to solve a complex design problem. A further investigation on the issue of user modelling together with the issue of defining a design context is necessary before a more intelligent user interface than the currently implemented one can be developed.

The directions in which these limitations can be overcome will be discussed in section 8.4.

8.2 Evaluation of the Learning Scheme

8.2.1 Features of the Design Concept Learning System

The development of a design concept learning system is motivated by the need to support conceptual design tasks in the early stages of the design process. The incremental learning model for design presented in chapter 4 has identified an inductive learning component within a knowledge-based design support architecture. The features of the implemented design concept learning system can be highlighted as follows:

- It is an integral part of a knowledge-based design support system architecture;

- It operates on structured data representation;
- It provides three different learning strategies (agglomerative, incremental and divisive, and heuristic);
- It utilises design heuristics as *background knowledge* in the learning process in a number of ways (data preparation, example ranking, generalisation, concept evaluation); and
- It graphically displays the design concept tree as it is being built.

8.2.2 Application

A task in the domain of small molecule drug design, i.e., the generation of a pharmacophore description has proved to be difficult [Hodgkin 1991]. The design concept learning system developed in this thesis has provided effective support to this task [Smithers *et al* 1992, 1993a and 1993b].

While many of the current machine learning systems can only deal with small sized applications, the architecture and the design concept learning system developed in this thesis have been tested in a real and complicated design problem. This has demanded considerable research effort first in becoming familiar with the concepts and the problems in the domain, and second in the lengthy knowledge elicitation process during which the data and information in this domain were organised.

In the Castlemaine project, several approaches were investigated to support the task of generating a pharmacophore description. An algorithm used in the first prototype of the Castlemaine system used a systematic method to compare all the attributes of all the fragments in all the molecules in a lead set. This method proved to be computationally too expensive and there is no way to embody any design heuristics into the method. It only managed to process a lead set of up to 6 molecules. Subsequently, this method was considered inapplicable in an evaluation working package of the Castlemaine project [Hodgkin 1991].

The development of a design concept learning system has been partly motivated by the need for a general inductive learning component within a knowledge-based design support system capable of supporting a task of this nature during the early stages of the design process. The idea of using inductive learning techniques to support the task of generating a pharmacophore description was conceived after the method used in the first prototype of the Castlemaine drug design support system had failed to generate satisfactory results. Several approaches were subsequently investigated by way of prototyping.

One approach used a *screening strategy* similar to ID3 to identify a pharmacophore description by first selecting a subset of molecules and then generating pharmacophore hypotheses from this subset [Hodgkin 1991]. The appropriateness of the pharmacophore hypothesis was then tested against the remaining molecules in the lead set. This approach closely modelled the way in which drug designers derive pharmacophore descriptions, and uses design heuristics in the selection of a subset of molecules. It was, however, non-incremental and still did not present a clear picture of how the molecules in a lead set are classified. It did not present any relation regarding the whole lead set, i.e., how the features of the molecules in the lead set were distributed. As a consequence, it could not compare alternatives. Nevertheless, this approach was considered in the evaluation of the Castlemaine project by the drug designers as more appropriate than the method used in the first prototype [Smithers *et al* 1992 and Buck *et al* 1991].

The design concept learning system presented in this thesis represents an attempt to provide a general support to design problems of this nature using inductive learning techniques. It improves the methodologies developed in the Castlemaine project further in the following aspects:

- It models the task of pharmacophore generation as an inductive learning problem and supports this task using an incremental and unsupervised learning strategy. Thus it allows a large number of molecules to be incrementally used to develop a design concept tree. For example, the first method adopted in the Castlemaine system was only able to deal with 6 molecules at most, whilst this incremental learning approach can easily deal with all 37 molecules in the GDS series;
- It explicitly uses the design heuristics as background knowledge in the learning process for preparing data, ranking molecule examples, specifying the levels of design concept at which the molecule examples are to be classified, and for evaluating the learning results. Utilisation of this background knowledge into the design concept learning system enables the design concept learning system to develop a design concept tree that is meaningful for design purposes;
- It provides a better interface for the designers to choose molecules to form a lead set, to add or to delete molecules from the lead set, to observe the differences, or to choose different methods for processing the original data i.e., to choose different sets of rules for the fragmentation of molecules.

In addition, the learning process is constructive in the sense that the final pharmacophore description is derived using some intermediate feature patterns, which are constructed from the original example molecules using domain knowledge that has been

formulated as production rules (see Appendix B). For example, the concepts of component, fragment and feature patterns and the rules to generate them from the original molecule examples are used to transfer the representation of an original molecule into structurally related object instances representing the feature patterns [Smithers *et al* 1993a and 1993b].

The design concept learning system developed in this thesis is intended to be used as a general learning tool with several learning strategies within a knowledge-based design support system architecture. It was first developed by the author as part of an integrated system in the Castlemaine project to support small molecule drug design applications. This integrated system was used by the drug designers involved in the Castlemaine project to support two key areas of drug design: pharmacophore generation and molecular modification by isostere replacement. The methods of generating feature patterns from original molecule examples, ranking molecule examples, generating pharmacophore descriptions using the third learning strategy as discussed in chapter 4 (section 4.2.2.2), and graphically explaining the design concept tree were mainly aimed at supporting the task of pharmacophore generation. A number of molecule sets including eight Beta-agonists and twelve Acyl Cholesterol o-Acyl Transferase (ACAT) inhibitors were taken from the literature in the Castlemaine project to test the integrated system. The test showed that the definition of a pharmacophore and its generation methods provided a useful framework within which (1) to identify isostere relationships in drug series, and (2) to propose candidates for chemical synthesis by making isostere substitutions.

After the Castlemaine project the design concept learning system has been further developed by the author as a general learning component of an AI-based design support system architecture. It is currently being used by the author as a learning component within an Integrated Functional Modelling System. In this domain the design concept learning system is used to cluster abstract conceptual design solutions (in terms of mechanical component configuration and orientation that satisfy a stated input/output requirement) generated by a functional synthesiser [Chakrabarti 1996 and Tang 1996]. Section 8.4 in this chapter will give more discussions on possible application and further development of the architecture and the design concept learning system in the domain of mechanical engineering design.

8.2.3 Limitations

Utilisation of Learning Techniques: The application of inductive learning techniques in this thesis has been confined to a particular problem of identifying a problem structure, or a model of an artefact (a pharmacophore description), from a set of examples. It can only support conceptual design tasks for which design heuristics as background knowledge exist.

The role of inductive inference in knowledge-based design support can also be viewed from a knowledge engineering point of view. That is, a learning sub-system within a knowledge-based design support system architecture should support a wide range of knowledge intensive activities such as *knowledge acquisition, discovering design knowledge from databases, refining design plans or design strategies from recorded design history or from design guideline databases* etc.

Other machine learning techniques such as case-based reasoning, explanation-based learning and genetic algorithms etc. can also support design tasks including conceptual design tasks. These techniques have not been utilised in the current implementation of the design concept learning system.

Learning at the Level of Lead Set: The indirect approach to small molecule drug design is based on the assumption that a pharmacophore description is derived from a set of similar molecules (a lead set) that are thought to bind to a target receptor in basically the same way as the new drug to be designed.

In the current application of the design concept learning system in small molecule drug design, a lead set is selected by the drug designers. This assumes the role of a drug designer in deciding which molecules are important. The design concept learning system is unable to make an automatic selection. To be able to do so it is necessary to define the concept of a *lead set*. Currently a lead set only contains a number of molecules selected by a drug designer. To learn at the level of lead set using the design concept learning system, e.g., to discriminate between different sets of molecules, more domain knowledge is needed to describe the features of the molecules at that level.

Dealing with Multiple Patterns: If an example has multiple interpretations (for example, a molecule contains multiple and overlapping feature patterns), then it will be classified under more than one node. The design concept learning system deals with examples with multiple patterns using an evaluation procedure. This evaluation procedure calculates a score for each of the nodes in the design concept tree. However, the current evaluation method used for supporting drug design has been developed on a domain dependent basis. A more general method is needed to deal with examples with multiple patterns in order for the design concept learning system to support the applications where no design heuristics are available.

Isostere Replacement: In the domain of small molecule drug design, there are two tasks which can benefit from using inductive learning techniques: the first is to induce from a lead set of molecules a structural description which best describes the common features of all or most of the molecules; the second is to use this structural description, called pharmacophore, to guide the design of a new drug by replacing part of a molecule in the

lead set, using an isostere library as a case base. This thesis has focused only on the first task, i.e., the inductive learning of a pharmacophore description.

The directions in which these limitations can be overcome will be discussed in section 8.5.

8.3 Evaluation of Integration

Learning and designing are closely related activities throughout the design process. A computer-based design support system therefore needs to support these two activities in an integrated way. In this thesis, the following attempts have been made to unify the architecture and learning in an integrated computational environment:

- the development of a learning model for design;
- the development of a knowledge-based design support system architecture that can document design history for replay and explanation purposes;
- the development of a design concept learning system as an integral part of the architecture; and
- the utilisation of design heuristics as background knowledge during the learning process.

The learning model for design presented in chapter 4 is suitable for a class of design tasks for which the initial structure of the design problem is not well known. In this learning model, inductive inference is used to support the formulation of a design concept tree from raw data or past design examples. The integration of a design concept learning system with the architecture therefore jointly supports conceptual development stage of design problem solving.

From a learning point of view, inductive learning is more efficient if *background knowledge* can be utilised during the learning process. Considerable efforts have been made in this research to establish how design heuristics as background knowledge can be utilised in the learning process. The application of the design concept learning system in the domain of small molecule drug design suggested that design heuristics as background knowledge can be used in *data preparation, ranking examples, generalisation and evaluation*. In particular, design knowledge can be used to define the levels of concept at which a design concept tree is to be developed.

The design documentation system in the ATMB architecture documents the design results in terms of both product and process information. A recorded design history can

be replayed to restore an interrupted design session, or replayed to explain how a previous design solution was reached. A design history represents design product knowledge as well as the rationale behind that knowledge. It can therefore be used as a refined form of design knowledge that enables the design knowledge base of a design support system to grow.

However, currently, the design documentation system only records design events in terms of assumptions made by the user. It can be replayed to create the design session up to the point when it was recorded. The issue of transferring design history records into prototypes of product data model or design strategies has not been addressed in this thesis. In future, the issue of reorganising and generalising recorded design information needs to be addressed in order to use the captured design knowledge in a more effective and innovative way [Ball *et al* 1992 and Reich 1993].

8.4 Future Directions for the Architecture

As a general design support tool, the ATMB architecture needs to be improved in the following aspects.

Object-Oriented Modelling of Product Data Model: Design knowledge representation is an important issue in knowledge-based design system development. A design knowledge representation scheme should make it easy to construct task-dependent design models (or product data models), based on pre-defined task-independent models and domain knowledge, to provide knowledge structures for new design tasks.

The current architecture provides a combined rule-based and object-oriented knowledge representation scheme for the user to build design objects and design knowledge sources. However, it is necessary to have a product data model that can provide more specific data structures for reasoning with functional, structural, causal and geometric relationships of various design objects common to design applications.

For example, in the domain of mechanical engineering design, a library of functional components and their assemblies can be developed using this representation scheme that can be easily assembled to form new product data models [Wallace *et al* 1995]. The acquisition, representation and manipulation of functional components, parts, and their assemblies enable the development of a system capable of performing functional synthesis, embodiment generation and kinematic analysis within an integrated environment.

Supporting Concurrent Engineering Design: The integration of an ATMS and a blackboard control system in the ATMB architecture provides a unique facility for the

exploration and maintenance of multiple contexts of design. This facility can be utilised for supporting concurrent engineering design during which a team of designers can create different design contexts without affecting the others involved in the design process. However, the architecture presented in this thesis has not been fully developed to support co-operation and negotiation processes which are important in concurrent engineering design. Its potential for supporting concurrent engineering design has yet to be fully explored. In future research, it is necessary to have a computational model for concurrency management which involves the exploration of multiple contexts of design (representing alternative design solutions), negotiation (resolving conflicts and balancing trade-offs of design choices), and co-operation (supplying knowledge and information to one another).

Supporting concurrency management is an important research area for computer-based design techniques. For example, Bowen's GALILEO2 has a powerful constraint-based language and reasoning facilities with which different perspectives can be defined by different members of a design team for life-cycle engineering design tasks [Bowen *et al* 1992]. However, GALILEO2 does not support the generation and maintenance of multiple contexts of design, and the negotiation process starts as soon as someone makes a single change to the constraint network [Bowen 1991]. Medland proposed a truth maintenance approach for design for manufacturability [Medland 1995]. This approach divides a constraint network into sub-systems, each of which reflects a different part of a design. Any sub-system has a status *true* if the constraints associated with it are satisfied. Otherwise this status is *false*. These sub-systems can be explored *simultaneously*, during which process only those sub-systems whose status is *true* are put forward for further analysis at a higher level. If a sub-system returns *false*, then an automatic search program is invoked to search for a set of design variable values that could satisfy the constraints associated with the sub-system. However, this approach does not have a systematic method to support the exploration of alternative solutions or multiple viewpoints.

The design context management system developed in this thesis can be further developed as a concurrency management tool to support concurrent engineering design. The advantage of using the design context management system in concurrent engineering design is that different and even conflicting decisions from different types of users can coexist in the system. Negotiations can be delayed until such time as several feasible solutions have emerged and it is necessary to select the best solution. A way of utilising the design context management system developed in this thesis for concurrency management has been discussed in [Tang 1995d].

Providing Constraint-based Reasoning Facilities: The ATMB architecture developed in this thesis as a domain independent design support system tool needs some general design support facilities such as *reasoning about function, dealing with quantitative and*

qualitative constraints etc. A general design support system is identified in the learning model for design as illustrated in Figure 4.2. However, this component has yet to be fully implemented as a general purpose design support system.

One approach to improving the ATMB architecture further is to develop a constraint manager for *constraint specification, constraint simplification, and constraint propagation*. This constraint manager should allow a conceptual design solution structure consisting of design variables and constraints to be further explored when considering its physical and geometric attributes.

This constraint manager should move away from traditional geometric-based reasoning by manipulating a multi-perspective constraint network at symbolic level [Bundy *et al* 1981, Sussman *et al* 1980 and Bowen *et al* 1991 and 1992]. It should use symbolic algebraic manipulation methods to solve the equations in the constraint set, thus generating a *reduced set of constraints* in which the equalities, redundancies and inconsistencies are already removed. This reduced set of constraints can then be further explored by propagating any assumptions made by the designers. A constraint propagation engine, a constraint simplification engine and a symbolic equation solving engine within the constraint manager will provide a basis for solving the so-called *back-of-envelope* calculation common in engineering design [Johnson 1988].

8.5 Future Directions for the Learning System

The current implemented design concept learning system integrates a number of strategies, namely, agglomerative, divisive and heuristic-based strategies within the confines of an unsupervised learning approach. As a result, the design concept learning system can only perform a task in the conceptual design stage that can be formulated as a concept formation problem or a clustering problem.

Inductive learning techniques as well as other forms of learning, such as genetic algorithms, neural networks and explanation-based learning are being developed quickly [Reich 1993 and Carbonell 1990]. In order to utilise these techniques effectively in a knowledge-based design support system, a theoretical understanding of learning (not only inductive learning) and design is needed. To be a general learning support system within a knowledge-based design support system architecture, the current design concept learning system needs to be extended and tested in a wider range of design activities such as *design analysis, design synthesis, design evaluation, and design knowledge capturing*.

The design concept learning system developed in this thesis provides a good basis for further development of a system in the domain of small molecule drug design by enhancing the library of isosteres and by building more facilities for reasoning on structure-activity relationships (SARs). *Isostere replacement* and *structure-activity analysis* play equally important roles in the design of a new drug when a pharmacophore

description has been derived. The structure-activity analysis of a molecule, based on the information provided by the pharmacophore description, aims at identifying exactly what structures contribute to a particular feature [Marshall *et al* 1986, Hodgkin 1991 and Hunter 1993].

In the current learning approach to drug design, as discussed in chapter 5, a simple isostere library is used for isostere replacement. However, this library contains only a limited number of isosteres compiled from relevant literature. It has a memory structure, but this memory does not grow in any sense when a failure is encountered. That is to say, no learning support is given by the system during this stage of the drug design process. Consequently, the support that the current isostere library offers to the design of new drugs is limited [Smithers *et al* 1993a].

Within the context of Case-Based Reasoning (CBR), an important function of a CBR system is its capability of learning from failure by explaining why a past case is not appropriate for a new situation, using domain theory. But current CBR in design research is very much oriented towards the automatic search and retrieval of past design cases to suit new requirements [Maher *et al* 1994]. Therefore one future research problem for enhancing the current learning approach to drug design is how to use case-based reasoning to learn the structure-activity relationships after a pharmacophore description has been derived. Using a case-based learning approach, if an isostere cannot be found in the isostere library by exact matching, then the system can resort to similar cases and modify one of them to adapt to the new situation, using domain theory. It is also necessary to connect the current design concept learning system to a chemical database where information about molecules, their properties, isosteric information and other information about structure-activity relationships at component level, or fragment level etc., can be obtained.

The design concept learning system is currently being used as a sub-system within an Integrated Functional Modelling System that supports mechanical engineering design [Chakrabarti *et al* 1996 and Tang 1996].

8.6 Applicability of the Architecture in Other Domains

The ATMB architecture as a self-contained software system kernel can be used to support the development of knowledge-based design applications in other domains. Some of the above issues concerning future development of the ATMB architecture and the design concept learning system are currently being addressed by the author at the Engineering Design Centre in Cambridge University in the development of an Integrated Functional Modelling system in mechanical engineering design [Tang 1995a, 1995c and 1995d, Wallace *et al* 1995a and b]. This Integrated Functional Modelling system, as illustrated in

Figure 8.1 can be seen as an application and further extension of the architecture developed in this thesis in mechanical engineering design.

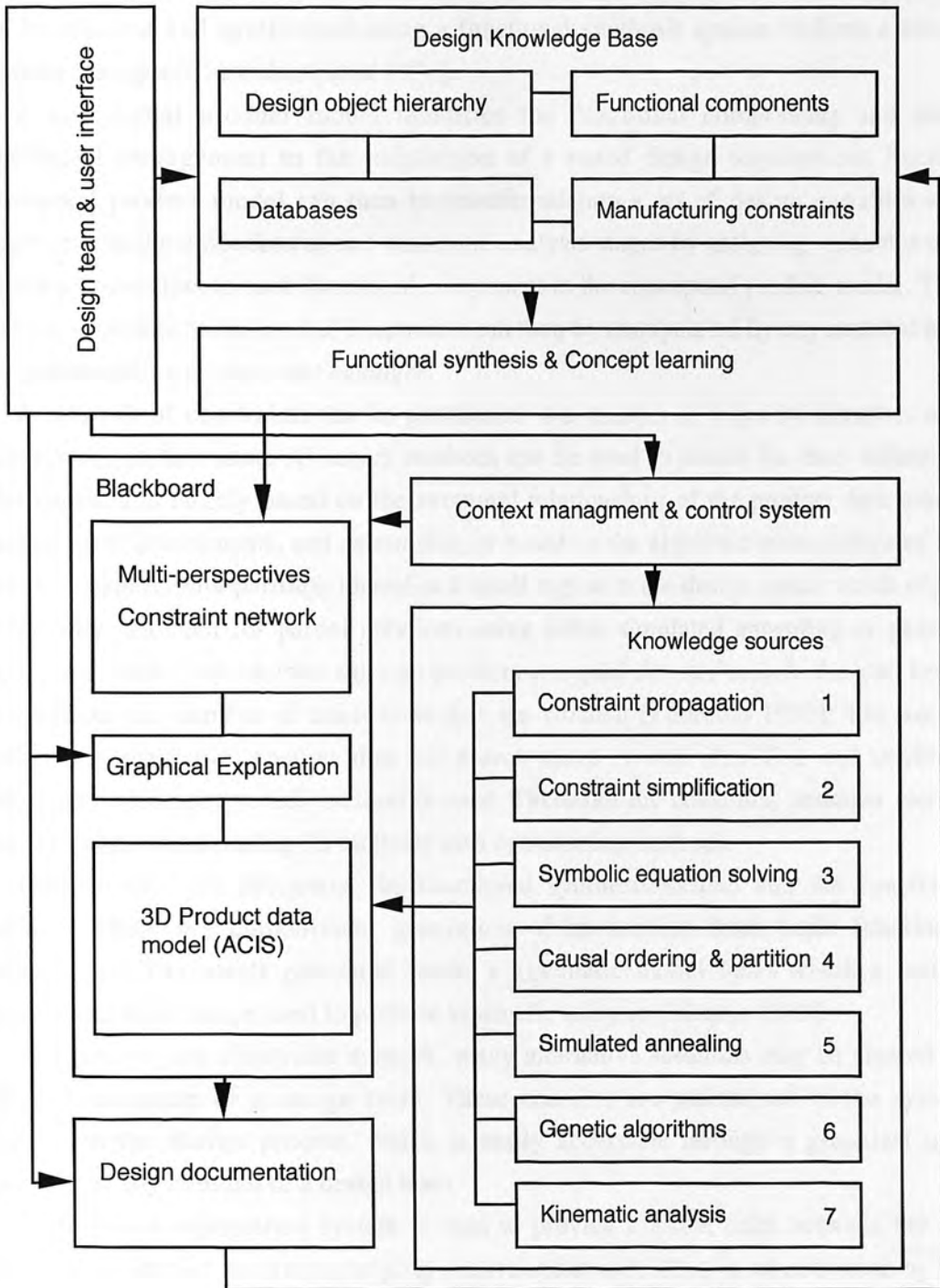


Figure 8.1: The Architecture of an Integrated Functional Modelling System

In this Integrated Functional Modelling system, the design knowledge base contains a library of *functional components*, a design object hierarchy representing the relationships

between these functional components, a set of manufacturing constraints on the assembly of these functional components, and a database containing detailed information about these functional components such as material, geometry etc. The functional components can be selected and synthesised using a functional synthesis system to form a set of solution concepts [Chakrabarti *et al* 1994].

A conceptual product model identifies the functional components and their topological arrangement to the satisfaction of a stated design requirement. Such a conceptual product model can then be transferred into a set of design variables and constraints in the embodiment and kinematic analysis stages by assigning variables and interface constraints to each functional component in the conceptual product model. This network of design variables and constraints can then be manipulated by any member of a design team using a *constraint manager*.

A network of constraints can be partitioned in a number of ways by members of a design team so that some AI search methods can be used to search for their solutions. This partition is largely based on the structural relationships of the product data model such as parts, components, and assemblies, or based on the algebraic relationships of the design variables. Any partition identifies a small region in the design space which might be usefully searched for partial solutions using either simulated annealing or genetic algorithms, both of which treat a design problem as a *goal-directed search*, the goal being to minimise the number of constraints that are violated [Thornton 1993]. The use of symbolic computation ensures that this search space is well identified and confined before any automatic search method is used. Therefore the constraint manager can be seen as a system integrating AI methods with optimisation methods.

Collectively, the designers, the functional synthesis system and the constraint manager allow the embodiment generation of an artefact from basic functional components. The result generated forms a kinematic model upon which a matrix reduction method can be used to perform kinematic analysis [Johnson 1993].

In exploring the constraint network, many alternative solutions may be created by different members of a design team. These contexts are maintained in the system throughout the design process, which is easily accessible through a graphical user interface by any member of a design team.

A graphical explanation system is used to provide a visual links between the 3D model of an artefact and its underlying constraint network. That is, what is done by the computer system at symbolic level is actually visualised in a 3D model. A design documentation system records the evolving product data model in terms of its specifications and the design history that has led to these specifications. The recorded design history can be reviewed and replayed by any member of a design team after the design session has concluded [Tang 1995a].

In order to make use of some of the software systems developed in this thesis, GoldWorks III¹ has been used to implement the Integrated Functional Modelling system. Some of the progress in the development of this system is reported in [Tang 1995a, 1995b, 1995c and Wallace *et al* 1995a and 1995b].

¹ GoldWorksIII is a updated version of GoldWorks II which has been used to develop the architecture and the design concept learning system in this thesis.

Chapter 9

Conclusions

In conclusion, the work recorded in this thesis addresses the issue of integrating inductive learning techniques into a knowledge-based design support system architecture. This work concerning knowledge-based design support and inductive learning was motivated by the need to develop an integrated knowledge-based design system using the best available AI techniques including inductive learning techniques to support the design tasks in the early stages of the design process when little is known about the product data model.

The ATMB architecture presented in this thesis is based on an integration of a blackboard control system and an assumption-based truth maintenance system. Apart from providing general support for the acquisition of design concept, design knowledge representation, intelligent control of design process and design documentation, this architecture provides a unique mechanism for the exploration and maintenance of multiple design contexts.

A design concept learning system has also been developed as an integral part of this knowledge-based design support system architecture. This design concept learning system utilises design heuristics as background knowledge in a unsupervised learning approach to support a class of design whose problem structures are initially not well known. It does so by developing a design concept tree that can be further analysed for design purposes.

Both the architecture and the design concept learning system have been implemented in a Lisp-based environment and successfully used in the domain of small-molecule drug design. The design concept learning system not only generates pharmacophore descriptions in an intuitive and efficient way, but also provides a design concept tree which allows multiple interpretation of the learning results.

A significant part of the research effort in this work was concerned with the hard task of becoming familiar with the domain and organising the knowledge in order to support the task of generating a pharmacophore description using computational techniques, including inductive learning techniques.

The limitations of the current architecture and the design concept learning system have been identified, which will benefit from a further investigation into the issues of modelling design context and user type for *concurrent engineering design*, developing a constraint manager to improve the ATMB architecture by combining AI and optimisation methods, and integrating the design concept learning system with other forms of machine learning techniques to support design knowledge acquisition and design concept learning.

Bibliography

[Andreasen 1991] Andreasen M., 1991, "Design Methodology", *Journal of Engineering Design*, Volume 2, Number 4, 1991.

[Anderson 1991] Anderson, J. R. and Matessa, M., 1991, "An Incremental Bayesian Algorithm for Categorisation", in *Concept Formation: Knowledge and Experience in Unsupervised Learning*, Fisher, D.(Eds.), Morgan Kaufmann.

[Archer 1970] Archer, L. B., 1970, "An Overview of the Structure of Design Process", in Moore, G. T.(Ed.), *Emerging Methods in Environmental Design and Planning*, MIT Press, Cambridge, Massachusetts, pp. 285-307.

[Arciszewski 1987] Arciszewski, T., Mustafa, M., and Ziarko, W., 1987, "A Methodology of Design Knowledge Acquisition for Use in Learning Expert Systems", *International Journal of Man-Machine Studies*, 27.

[Asimow 1962] Asimow, M., 1962, "Introduction to Design", New York: Prentice-Hall.

[Babin *et al* 1991] Babin, B. A. and Loganantharaj, R., 1991, "Designer's Workbench: a Tool to Assist in the Capture and Utilisation of Design Knowledge", First International Conference on Artificial Intelligence in Design, June, 1991, Edinburgh, Scotland.

[Balachandrian *et al* 1990] Balachandrian, M., and Gero, J. S., 1990, "A Knowledge-Based Approach to Mathematical Design Modelling and Optimisation", *Engineering optimisation*, 12,(12):99-115.

[Ball *et al* 1992] Ball, N. and Bauert, F., 1992, "The Integrated Design Framework: Supporting the Design Process Using a Blackboard System", International Conference on Artificial Intelligence in Design, 1992.

[Banares-Alcantara 1991] Banares-Alcantara, R., 1991, "Representing the Engineering Design Process: Two Hypotheses", First International Conference on Artificial Intelligence in Design, June, 1991, Edinburgh, Scotland.

[Bratko 1993] Bratko, I., 1993, "Machine Learning in Artificial Intelligence", Special Issue on Machine Learning and Design, *International Journal of Artificial Intelligence in Engineering*, Vol. 8, 1993.

[Bowen 1991] Bowen, J., 1991, "Supporting Cooperation between Multiple Perspectives in a Constraint-based Approach to Concurrent Engineering", *Journal of Design and Manufacturing* (1991) 1.

[Bowen *et al* 1992] Bowen, J., and Bahler, D., 1992, "Frames, quantification, perspectives, and negotiation in constraint networks for life-cycle engineering", *Artificial Intelligence in Engineering* 7 (1992).

[Bracewell *et al* 1993] Bracewell, R., Bradley, D., Chaplin, R., Landon, P., and Sharpe, J., 1993, "Schemebuilder: A Design Aid for the Conceptual Stages of Product Design", Proceedings of the 9th International Conference on Engineering Design, The Hague, 1993.

[Brown *et al* 1985] Brown, D. C., and Chandrasekaran, B., 1985, "Expert Systems for a Class of Mechanical Design Activity", in J. S. Jero (Ed.), *Knowledge Engineering in Computer-Aided Design*, North-Holland, Amsterdam.

[Brown *et al* 1989] Brown, D. C., and Chandrasekaran, B., 1989, "Design Problem

Solving: Knowledge Structures and Control Strategies", Pitman, London.

[Brown *et al* 1991] Brown, D. C. and Spilane, M. B., 1991, "An Experimental Evaluation of Some Design Knowledge Compilation Mechanisms", First International Conference on Artificial Intelligence in Design, June, 1991, Edinburgh, Scotland.

[Buck *et al* 1991] Buck, P., Clarke, B., Lloyd, G., Poulter, K., Smithers, T., Tang, M., Tomes, N., Floyd, C. and Hodgkin, E., 1991, "The Castlemaine Project: Development of an AI-based Design Support System", Artificial Intelligence in Design Conference, June 1991, Edinburgh, Scotland.

[Bundy *et al* 1981] Bundy, A., and Welham, B., 1981, "Using Meta-level Inference for Selective Application of Multiple Rewrite Rules in Algebraic Manipulation", Artificial Intelligence 16 (2), 1981.

[Burton *et al* 1990] Burton, A. M., Shadbolt, N. R., Rugg, G., Hedgecock, A. P., 1990, "The Efficacy of Knowledge Elicitation Techniques: a Comparison Across Domains and Levels of Expertise", *Knowledge Acquisition* 2.

[Carbonell 1990] Carbonell, J., 1990, "Introduction: Paradigms for Machine Learning", in *Machine Learning, Paradigms and Methods*, J. G. Carbonell (Ed.), The MIT Press.

[Carter *et al* 1991] Carter, I., and MacCallum, K., 1991, "A Software for Design Co-ordination", First International Conference on Artificial Intelligence in Design, June, 1991, Edinburgh, Scotland.

[Chandrasekaran 1986] Chandrasekaran, B., 1986, "Generic Tasks in Knowledge-Based Reasoning: High Level Building Blocks for Expert System Design", *IEEE Expert*, 1(3), 1986.

[Chakrabarti,*et al* 1994] Chakrabarti, A., and Bleigh, T., 1994, "A Two-step Approach to Conceptual Design of Mechanical Device", Third International Conference on Artificial Intelligence in Design.

[Chakrabarti,*et al* 1996] Chakrabarti, A., and Tang, M., 1996, "Generating Conceptual Solutions on FuncSION: the Evolution of a Functional Synthesiser", 4th International Conference on Artificial Intelligence in Design, Stanford, USA, June 1996.

[Christopher *et al* 1989] Reisberk, C., K. and Schank, R. C., 1989, "Inside Case-Based Reasoning", Lawrence Erlbaum Associates Inc.

[Clarke *et al* 1991] Clarke, J., and Randal, D., 1991, "An Intelligent Front-End for Computer-Aided Building Design", *Artificial Intelligence in Engineering*, 1991, Vol. 6, No. 1.

[Coyne *et al* 1990], Coyne, R. D., Rosenman, M. A., Radford, A. D., Balachandran, M., Gero, J. S., 1990, "Knowledge-Based Design Systems", Addison-Wesley Publishing Company, 1990.

[DeJong *et al* 1986] DeJong, G., and Mooney, R., 1986, "Explanation-Based Learning: An Alternative View" in *Machine Learning* 1:145-176, Kluwer Academic Publishers, Boston.

[de Kleer 1984a] de Kleer, J., 1984, "Choices without Backtracking", Proceedings of 1984 AAAI Conference, American Association of Artificial Intelligence, 1984.

[de Kleer *et al* 1984b] de Kleer, J. and Brown, J. S., 1984, "Qualitative Physics on Confluences", *Artificial Intelligence*, Vol. 24, 1984.

- [de Kleer 1986] de Kleer, J., "An Assumption-based TMS", *Artificial Intelligence*, Vol. 28, 1986.
- [de Kleer *et al* 1986] de Kleer, J. and Brown, J. S., 1986, "Theories of Causal Ordering" *Artificial Intelligence*, Vol. 29, 1986 (33-61).
- [Dietterich *et al* 1981] Dietterich, T. G. and Michalski, R. S., 1981, "Inductive Learning of Structural Description", *Artificial Intelligence*, (1981) 257-294.
- [Dixon 1986] Dixon, J. R., 1986, "Designing with Features: Creating and Using a Features Database for the Evaluation of Manufacturability of Castings", *ASME Computers in Engineering*, 1986, Vol. 1.
- [Doyle 1979] Doyle, J., 1979, "A Truth Maintenance System", *Artificial Intelligence*, 12(3):231-272, 1979.
- [Duffy *et al* 1993] Duffy, A. H. B. and Kerr, S. M., 1993, "Customised Perspectives of Past Designs from Automated Group Rationalisations", Special Issue on Machine Learning and Design, *International Journal of Artificial Intelligence in Engineering*, Vol. 8, 1993.
- [Everitt 1981] Everitt, B., "Cluster Analysis", London: Heinemann, 1981.
- [Feigenbaum 1961] Feigenbaum, E. A., 1961, "The Simulation of Verbal Learning Behaviour" in *Readings in machine learning*, Edited by Jude W. Shavlik and Thomas G. Dietterich, 1990, Morgan Kaufmann.
- [Feigenbaum *et al* 1984] Feigenbaum, E. A. and Simon, H. A., 1984, "EPAM-like Models of Recognition and Learning", *Cognitive Science*, 8, 305-336, 1984
- [Fielden 1963] Fielden, G. B. R., 1963, "The Fielden Report", *Engineering Design*, London: HMS.
- [Fisher *et al* 1985] Fisher, D. H. and Langley, P., 1985, "An Approach to Concept Clustering", *Proceedings of Ninth International Joint Conference on Artificial Intelligence* (pp. 691-697), Los Angeles, CA: Morgan Kaufmann, 1985.
- [Fisher 1987] Fisher, D. H., 1987, "Knowledge Acquisition via Incremental Conceptual Formation", in *Readings for Machine Learning*, Shavlik, J. (Eds.), Morgan Kaufmann.
- [Fisher *et al* 1991] Fisher, D. H. and Pazzani, M., 1991, "Computational Models of Concept Formation", in *Concept Formation: Knowledge and Experience in Unsupervised Learning*, Fisher, D.(Eds.), Morgan Kaufmann.
- [Floyd 1990] Floyd, C., 1990, "Concepts in Drug Design", The Castlemaine Project Report (Castlemaine/BBL/1.0), Department of Artificial Intelligence, University of Edinburgh, Scotland, 1990.
- [Forbus 1984] Forbus, K. D., 1984, "Qualitative Process Theory", MIT AI Laboratory, Boston MA, 12984.
- [Forsyth *et al* 1986] Forsyth, R. and Rada, R., 1986, "Machine Learning, Applications in Expert Systems and Information Retrieval", Ellis Howard Series in Artificial Intelligence.
- [Ganeshan *et al* 1991] Ganeshan, S., Finger, S., and Garrett, J., 1991, "Representing and Reasoning with Design Intent", *Proceedings of International Conference on Artificial Intelligence in Design*, June, 1991, Edinburgh, Scotland.

[Gennari *et al* 1989] Gennari, J. H. Langley, P. and Fisher, D., 1989, "Model of Incremental Concept Formation", in *Machine Learning, Paradigms and Methods*, J. G. Carbonell (Eds.), The MIT Press.

[Gero *et al* 1989] Gero, J. S. and Roseman, M. A. "A Conceptual Framework for Knowledge-Based Design Research at Sydney University's Design Computing Unit", in Gero, J. S. (Ed.), *Artificial Intelligence in Design*, Springer-Verlag, UK (1989).

[Gluck *et al* 1985] Gluck, M. A. and Corter, J. E., 1985, "Information, Uncertainty, and the Utility of Categories", Proceedings of the Seventh Annual Conference of the Cognitive Science Society (pp. 283-287), Irvine, CA: Lawrence Erlbaum.

[Gross *et al* 1988] Mark D. Gross, Stephen M. Ervin, James A. Anderson and Aaron Fleisher, "Constraints: Knowledge Representation in Design", *Design Studies*, Vol. 9 No. 3 July, 1988.

[Hammond 1989] Hammond, K. J., "CHEF", in *Inside Case-Based Reasoning*, Reisbeck, C.(Eds.), 1989, Lawrence Erlbaum Associates, Hillsdale, NJ.

[Hayes-Roth 1985] Hayes-Roth, B., 1985, "Blackboard Architecture for Control", *Journal of Artificial Intelligence*, 26:251-231, 1985.

[Hodgkin 1991] Hodgkin, E., 1991, "The Castlemaine Project, Development of an AI-Based Drug Design Support System", Workshop of Molecular Graphics Society, Essex, UK, 1991.

[Huang *et al* 1990] Huang, C. C., Klein, T. E., Morris, J. H., Ferrin, T. E., Langrige, R., 1992, "The Macromolecular Workbench and Its Application to the Study of Collagen", 24th Hawaii International Conference on System Science, 1990, Hawaii, USA.

[Hunt *et al* 1966] Hunt, E.B., Marin, J. and Stone, P.J., 1966, "Experiments in Induction", Academic Press, New York, 1966.

[Hunter 1993] Hunter, L., 1993, "Molecular Biology for Computer Scientists", 26th Hawaii International Conference on System Science, 1993, Hawaii, USA.

[Iwasaki *et al* 1986] Iwasaki, Y. and Simon, H., "Causality in Device Behaviour", *Artificial Intelligence*, Vol. 29, 1986 (3-32).

[Johnson 1988], A. L., Johnson, 1988, "Functional Modelling: A New Development in Computer-aided Design", Proceedings of IFIP WG5.3 workshop on Intelligent CAD, 1988, Cambridge, edited by H., Yoshikawa and T., Holden.

[Jones *et al* 1986] Jones, J., and Millington, M., "An Edinburgh Prolog Blackboard Shell", D.A.I. Research Paper No. 281, Edinburgh University, Department of Artificial Intelligence, 1986.

[Jones 1992] Jones, J. C., 1992, "Design Methods", Second Edition, Van Nostrand Reinhold, New York.

[Katai *et al* 1991] Katai, O., Kawakami, H., Sawarragi, T., and Iwai, S., 1991, "A Knowledge Acquisition System for Conceptual Design Based on Functional and Rational Explanation of Designed Objects", First International Conference on Artificial Intelligence in Design, June, 1991, Edinburgh, Scotland.

[Kim 1990] Kim, S. H., 1990, "Designing Intelligence, a Framework for Smart Systems" Oxford University Press, Oxford.

- [Kimura 1989] Kimura, F., 1989, "Architecture and Implementation", Summary of IFIP WG5.2 Workshop on Intelligent CAD, in "Intelligent CAD III" Yoshikawa et al (Eds.).
- [King *et al* 1993] King, R. D., Muggleton, S, Feng, C., Lewis, R. A., Sternberg, M. J. E., 1993, "Drug Design Using Inductive Logic Programming", 26th Hawaii International Conference on System Science, Hawaii, USA.
- [Koile *et al* 1991] Koile, K., Shapiro, R., 1991, "Building a Collaborative Drug Design System", 25th Hawaii International Conference on System Science, 1991, Hawaii, USA.
- [Kuipers 1986] Kuipers, B., 1986, "Qualitative Simulation", *Artificial Intelligence*, 29.
- [Laird *et al* 1987] Laird, J. E., Newell, A., and Rosenbloom, P. S., 1987, SOAR: An architecture for general intelligence, *Artificial Intelligence*, 33(1), 1987.
- [Laird *et al* 1987] Laird, J. E., and Newell, A. and Rosenbloom, P. S., 1987, "SOAR: A Architecture for General Intelligence", *Artificial Intelligence*, 33, 1987.
- [Latombe 1976] Latombe, J-C, "Artificial Intelligence in Computer Aided Design: The TROPIC System", AI Technical Note 125, Stanford Research Institute, Menlo Park, CA, 1976.
- [Lebowitz 1987] Lebowitz, M., 1987, "Experiments with Incremental Concept Formation: UNIMEM", *Machine Learning*, 2, 1987.
- [Logan *et al* 1991] Logan, B., Millington, K., and Smithers, T., 1991, "Be Economical with the Truth, Assumption-based Context Management in the Edinburgh Designer System", *Artificial Intelligence in Design Conference*, June 1991, Edinburgh, Scotland.
- [MacCallum 1990], MacCallum, K. J., 1990, "Does Intelligent CAD Exist?", *Artificial Intelligence in Engineering*, 5(2).
- [Maher 1988] Maher, K., L., 1988, "Engineering Design Synthesis: A Domain-Independent Representation", *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, 1(3):207-213.
- [Maher 1990] Maher, M. L., 1990, "Process Models for Design Synthesis", *AI Magazine* (winter, 1990) pp 49-58.
- [Maher *et al* 1994] Maher M. L., Balachandran, B., 1994, "Flexible Retrieval Strategies for Case-based Design", *Proceedings of the 3rd International Conference on Artificial Intelligence in Design*, June, 1994, Switzerland.
- [McLaughlin *et al* 1987] McLaughlin, S., and Gero, J. S., 1987, "Acquiring Expert Knowledge from Characterised Designs", *Artificial Intelligence for Engineering Design and Manufacturing*, 1(2):73-87.
- [Marcus *et al* 1988] Marcus, S., Stout, J., McDermott, J., 1988, "VT: An Expert Elevator Designer That Uses Knowledge-Based Backtracking", *AI Magazine*, 9(1):95-112.
- [Marshall *et al* 1986] Marshall, G., and Motoc, I., "Approaches to the Conformation of the Drug Bound to the Receptor", in *Molecular Graphics and Drug Design*, Burgen, A. *et al* (Eds.), Elsevier, Amsterdam, 1986.
- [Medland 1995] Medland, A.J., 1995, "Managing Design and Manufacturing Constraints in a Distributed Industrial Environment: the Creation of a Managed Environment for Engineering Design", *Lancaster International Workshop on Engineering Design*, Ambleside, UK, March, 1995.

- [Matchett 1968] Matchett, E., 1968, "Control of Thought in Creative Work" in *The Design Method* edited by S. Gregory, London: Butterworths.
- [Matwin *et al* 1991] Matwin, J. D. and Billman, D., "Representational Specificity and Concept Learning", in *Concept Formation: Knowledge and Experience in Unsupervised Learning*, Edited by Fisher, D. Pazzani, M. and Langley, P., Morgan Kaufmann, 1991.
- [Michalski 1977] Michalski, R.S., 1977, "Variable-Valued Logic and Its Application to Pattern Recognition and Machine Learning", in Rine, D. C.(Ed.), *Computer Science and Multiple-Valued Logic*, North-Holland, Amsterdam, 1977, 506-534.
- [Millington *et al* 1988] Millington, K., Robertson, A. and Smithers, T., 1988, "An Architecture for AI-Based Design: a Foundation Made Concrete", in H. Yoshikawa and T. Holden (Eds.), *Proceedings of the Second IFIP WG5.2 Workshop on Intelligent CAD*, 19-22 September 1988, Cambridge, UK.
- [Minton *et al* 1989] Minton, S., Carbonell, J. G., Knoblock, G. A., Kuokka, D. R., Etzioni, O. and Gil, Y., 1989, "Explanation-Based Learning: a Problem Solving Perspective", in *Machine Learning, Paradigms and Methods*, J. G. Carbonell (Eds.) The MIT Press.
- [Minton 1990] Minton, S., 1990, "Learning Search Control Knowledge, an Explanation-Based Approach", Kluwer Academic Publishers.
- [Mitchell 1977] Mitchell, T. M., "Version Spaces: a Candidate Elimination Approach to Rule Learning", *Proceedings of the 5th International Joint Conference on Artificial Intelligence (1977)* 305-310.
- [Mitchell 1982], Mitchell, T. M., 1982, "Generalisation as Search", *Artificial Intelligence* ", 18, 203-226.
- [Mitchell *et al* 1986] Mitchell, T. M., Keller, R. M. and Kedar-cabelli, S. T., 1986, "Explanation-Based Gneralisation: A Unifying View", *Machine Learning* 1:1, Kluwer Academic Publishers, Boston.
- [Mittal *et al* 1986] Mittal, S., Dym, C. L., and Morjaria, M., 1986, "PRIDE: An Expert System for the Desing of Paper Handling System", in C. L. Dym (ed.), *Applications of Knowledge-Based Systems to Engineering Analysis and Design*, America Society of Mechanical Engineers, New York, pp. 99-115.
- [Mortimer 1988] Mortimer, H., 1988, "The Logic of Induction", Ellis Howard Series in Artificial Intelligence.
- [Mostow 1989] Mostow, J., 1989, "Design by Derivational Analogy: Issues in the Automated Replay of Design Plans", *Machine Learning, Paradigms and Methods*, J. G. Carbonell (Eds.), The MIT Press.
- [Mostow *et al* 1987] Mostow, J. and Barley, M., "Automated Reuse of Design Plans" in W. E. Eder (Ed.), *Proceedings of 1987 International Conference on Engineering Design*, Boston, MA. 1987.
- [Muggleton 1990] Muggleton, S., "Inductive Acquisition of Expert Knowledge", Addison-Wesley, 1990.
- [Muggleton 1992] Muggleton, S., (ED.), "Inductive Logic Programming", Academic Press, London, 1992.
- [Nii 1986] Nii, H. P., 1986, "Blackboard Systems: Part 1", *AI Magazine* Vol 7 (1986) p

[Persidis *et al* 1989] Persidis, A., and Duffy, A. H. B., 1989, "Machine Learning in Engineering Design", in *Intelligent CAD III*, IFIP 5.2 WG Workshop on Intelligent CAD, Yoshikawa *et al* (Eds.), Osaka, Japan, 1989.

[Pugh 1989] Pugh, S., 1989, "Knowledge-based Systems in the Design Activity", in *Design Studies*, Vol.4, No. 10.

[Quinlan 1979] Quinlan, J. R., 1979, "Discovering Rules by Induction from Large Collections of Examples", *Introductory Readings in Expert Systems*, D. Michie (Ed.), Gordon and Breach, London, 1979.

[Quinlan 1986] Quinlan, J. R., 1986, "Learning from noisy data", *Machine Learning*, Volume 2. J. Carbonell and T. Mitchell (Eds.), Tioga, Palo Alto, USA, 1986.

[Quinlan 1988] Quinlan, J. R., 1988, "Induction, Knowledge and Expert Systems, Artificial Intelligence Developments and Applications", Jero, J. S., and Standon, R., (Eds.) North-Holland: Elsevier Science Publishers 1988.

[Reich *et al* 1991] Reich, Y., Coyne, R. D., Modi, A., Steier, D. and Subrahmanian, E., 1991, "Learning in Design: an EDRC(US) Perspective", First International Conference on Artificial Intelligence in Design, June, 1991, Edinburgh, Scotland.

[Reich 1993] Reich, Y., 1993, "The Development of Bridger: A Methodological Study of Research on the Use of Machine Learning in Design", Special Issue on Machine Learning and Design, *International Journal of Artificial Intelligence in Engineering*, Vol. 8, 1993.

[Reich *et al* 1993] Reich, Y., Konda, S. L., Levy, S. N., Monarch, I. A., and Subrahmanian, E., 1993, "New Roles for Machine Learning in Design", Special Issue on Machine Learning and Design, *International Journal of Artificial Intelligence in Engineering*, Vol. 8, 1993.

[Ross 1989] Ross, P., 1989, "A Simple ATMS", Department of Artificial Intelligence, University of Edinburgh, UK. June, 1989.

[Schank *et al* 1989] Schank, R. C. and Leake, D. B., 1989, "Creativity and Learning in a Case-Based Explainer", *Machine Learning, Paradigms and Methods*, J. G. Carbonell (Eds.) The MIT Press.

[Serrano *et al* 1994], Serrano, D. and Gossard, D, 1994, "Constraint Management in MCAE", Proceedings of International Conference on Artificial Intelligence in Design, 1994.

[Simon 1973] Simon H. A. , 1973, "The Structure of Ill-structured Problems", *Artificial Intelligence*, 4, 181-201.

[Smithers *et al* 1990a] Smithers, T. and Troxell, W., 1990, "Design is Intelligent Behaviour, but What is the Formalism", *AI EDAM*, 1990, 4(2), 89-98.

[Smithers *et al* 1990b] Smithers, T., Conkie, A., Doheny, J., Logan, B., Millington, K. and Tang, M., 1990, "Design as Intelligent Behaviour: An AI in Design Research Programme", *International Journal of Artificial Intelligence in Engineering*, 5.

[Smithers 1991a] Smithers, T. and Tang, M., 1991, "Towards AI-Based Design" International Conference for Young Computer Scientists, "Towards the Future", July 1991, Beijing, The People's Republic of China.

[Smithers *et al.* 1991b] Smithers, T., Tang, M. and Tomes, N., "The Maintenance of Design History in AI-Based Design", IEE Colloquium on Tools and Techniques for Maintaining Traceability During Design, 2nd December, 1991, IEE, London.

[Smithers *et al.* 1992] Smithers, T., Tang, M., Tomes, N., Buck, P., Clarke, B., Lloyd, G., Poulter, K., Floyd, C. and Hodgkin, E., 1992, "Development of a Knowledge-Based Design Support System", International Journal of Knowledge Based Systems, March, 1992.

[Smithers *et al.* 1993a] Smithers, T., Tang, M. and Tomes, N., 1993, "Approach in Intelligent Drug Design", 26th Hawaii International Conference on System Science, January, 1993, Hawaii, USA.

[Smithers *et al.* 1993b] Smithers, T., Tang, M., Ross, P. and Tomes, N., 1993, "An Incremental Learning Approach for Indirect Drug Design", International Journal of Artificial Intelligence in Engineering, Special Issue on Design and Machine Learning, Vol. 8, 1993.

[Sriram *et al.* 1989] Sriram, D., Stephanopoulos, G., Logcher, R., Gossard, D., Groleau, N., Serrano, D., and Navinchandra, D., 1989, "Knowledge-Based System Applications in Engineering Design: Research at MIT", AI Magazine, Fall 1989.

[Sriram *et al.* 1992] Sriram, D., Logcher, R., Groleau, N., and Cherneff, J., 1992, "DICE: An Object_oriented Programming Environment for Cooperative Engineering Design", in Artificial Intelligence in Engineering Design, Tong, C., (Ed.), Academic Press, 1992.

[Stepp *et al.* 1986] Stepp, R. E., Michalski, R. S., 1986, "Conceptual Clustering of Structured Objects: A Goal-Oriented Approach", *Artificial Intelligence*, 28, 43-69.

[Sussman *et al.* 1980] Sussman, G. J. and Steer L. G., 1980, "CONSTRAINTS - A Language for Expressing Almost-Hierarchical Descriptions", *Artificial Intelligence* 14 (1980).

[Tang *et al.* 1991a] Tang, M. and Smithers, T., 1991, "Towards Object-Oriented Simulation", IEE Colloquium on *Object-Oriented Simulation and Control*, March 1991, IEE, London.

[Tang *et al.* 1991b] Tang, M. and Smithers, T., 1991, "Object-Oriented Simulation of Behaviour in Design Support", 1991, Fourth UNB AI Symposium, September 19-21, 1991, Fredericton, N.B. Canada.

[Tang *et al.* 1992] Tang, M. and Smithers, T., 1992, "AI-Based Design and Simulation of Hydroelectric Power Systems", IFAC International Symposium on Power Plant and Power Systems, 10-12, March, 1992, Munich, Germany.

[Tang 1995a] Tang, M., 1995, "Development of an Integrated AI System for Conceptual Design", 1995 Lancaster International Workshop on Engineering Design: AI System Support for Conceptual Design, Ambleside, UK, March, 1995.

[Tang 1995b] Tang, M., 1995, "Development of Concept Learning System for Intelligent Design Support", Fifth Scandinavian Conference on Artificial Intelligence, Trondheim, Norway, June, 1995.

[Tang 1995c] Tang, M., 1995, "Exploring Design Solutions Using a Context Management System", IJCAI95 Workshop on Modelling Context in Knowledge Representation and Reasoning, August, 1995, Montreal, Canada.

[Tang 1995d] Tang, M., 1995, "An AI-Based Architecture for Concurrency Management

- in Engineering Design", ASME paper No. ASME-DE-96/1, presented at the ASME Design For Manufacturability Conference in Chicago on March 21, 1996.
- [Tang 1996] Tang, M., 1996, "An Integrated Application of Machine Learning Techniques in Intelligent Design Support", AID96 workshop on Machine Learning in Design, June 1996, Stanford, USA.
- [Tello 1989] Tello, E. R., 1989, "Object-Oriented Programming for Artificial Intelligence", Addison-Wesley, 1989.
- [Thoms 1990] Thoma, J., 1990, "Simulation by Bondgraphs: Introduction to a Graphical Method", Springer-Verlag, 1990.
- [Thornber 1979] Thornber, C. W., 1979, "Isosterism and Molecular Modification in Drug Design", Chem. Soc. Rev. 8, 563-580.
- [Thornton 1993] Thornton, A., 1993, "Constraint Specification and Satisfaction in Embodiment Design", PhD Thesis, University of Cambridge, Department of Engineering, 1993.
- [Wallace *et al* 1995a] Wallace K., Ball, N., and Tang, M., 1995, "AI in Engineering Design", Fourth Workshop on Research Directions for Artificial Intelligence in Design", University of Twente, Enschede, The Netherlands, January, 1995.
- [Wallace *et al* 1995b] Wallace K., Ball, N., and Tang, M., 1995, "Object-Oriented Technology and Engineering Design", Workshop on Object-Oriented Technology, Strathclyde, April, 1995.
- [Weininger 1989] Weininger, D., 1989, "Smiles. 3: Deepict. Graphical Depiction of Chemical Structures", *Journal of Chemical Information*, 1990.
- [Winsor et al 1994] Winsor, J., and MacCallum, K., 1994, "A Review of Functionality Modelling in Design", *The Knowledge Engineering Review*, Vol. 9:2, 1994.
- [Winston 1975] Winston, P.H., "Learning Structural Descriptions from Examples", in Winston, P.H. (Ed.), *The Psychology of Computer Vision*, McGraw-Hill, New York, 1975.
- [Yoshikawa *et al* 1989] Yoshikawa, M., Arbab, F., and Tomiyama, T., 1989, "Intelligent CAD III", IFIP WG5.2 Workshop on CAD, Osaka, Japan 1989.
- [Young *et al* 1991] Young, R. E., Greef, A. and Grady, P. O., 1991, "SPARK: An Artificial Intelligence Constraint Network System for Concurrent Engineering", First International Conference on Artificial Intelligence in Design, June, 1991, Edinburgh, Scotland.
- [Zhao *et al* 1988] Zhao, F and Maher, M L, 1988, "Using Analogical Reasoning to Design Buildings", *Engineering Computing*. Vol 4 (1988) pp 107-119.

Appendix A

Drug Design Concepts and Rules

1. Smiles String Representation of Molecular Structure

A simple 4-rule set suffices for the vast majority of organic compounds. This rule set uses only the symbols H, C, N, O, P, S, F, Cl, Br, I, (,), and digits. Examples of these rules are shown in Figure A1.

Rule 1: Atoms are represented by atomic symbols.

Rule 2: Double and triple bonds are represented by "=" and "#", respectively.

Rule 3: Branching is indicated by parentheses.

Rule 4: Ring closures are indicated by matching digits appended to symbols.

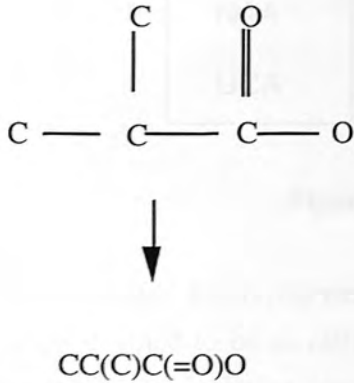
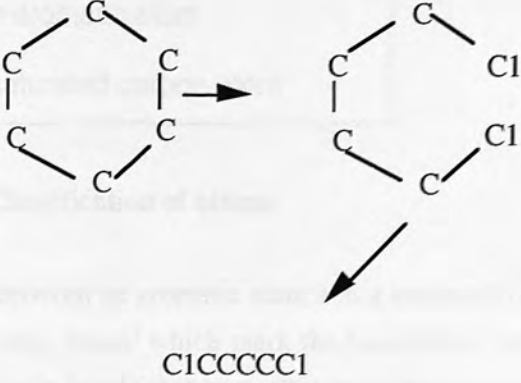
<i>Rule1 (atoms)</i>	<i>Rule2 (bonds)</i>
C ---> methane (CH ₄) N ---> ammonia (NH ₃) O ---> water (H ₂ O)	CC ---> ethane (CH ₃ .CH ₃ , single bond) C=C ---> ethylene (CH ₂ =CH ₂), double bond C#N ---> hydrogen cyanide (HCN) triple bond
<i>Rule 3 (branches)</i>	<i>Rule 4 (cyclic structures)</i>
 <p>CC(C)C(=O)O</p>	 <p>ClCCCCCCl</p>

Figure A1: Smiles String Rules

For example, in Figure A1, C ---> methane (CH₄) means C represents CH₄. In Rule 4 cyclic structures are represented by breaking one single bond in each ring. These bonds can be numbered in any order, designating ring opening (or ring closure) bonds by a digit immediately following the atomic symbol at each ring closure. This leaves a connected non-cyclical graph which is written as a non-cyclical structure using three other rules. Additionally, aromatic structures can be distinguished by writing the atoms in the atomic ring in lower case letters.

For example, benzoic acid can be represented as "c1ccccc1C(=O)O" where "c1ccccc1" is an aromatic ring. This sub-structure is recognisable using the smiles string rules and it can sometimes be used to search chemical databases.

2. Rules for Component Partition

There are rules for partitioning molecules into molecular components. These rules naturally represent the heuristic knowledge used by medicinal chemists when they analyse molecules. In order to partition a molecule, each atom in a molecule is assigned to a category such as *aromatic*, *heteroatom*, or *polar carbon atom*. The full atom classes are illustrated in Figure A2.

Class	Definition
HA	Hetero atom
PCA	Polar carbon atom
NPCA	Non polar carbon atom
AA	Aromatic atom
NAA	Non aromatic atom
UCA	Unsaturated carbon atom

Figure A2: Classification of atoms

Certain single bonds, for example, between an aromatic atom and a saturated carbon atom, are deemed to be so-called '*isolating bonds*' which mark the boundaries between two components. Isolating bonds are single bonds that separate two components. Four rules can be used to perform component partitioning. These rules, as shown in Figure A3, are universal and applicable to any organic structure.

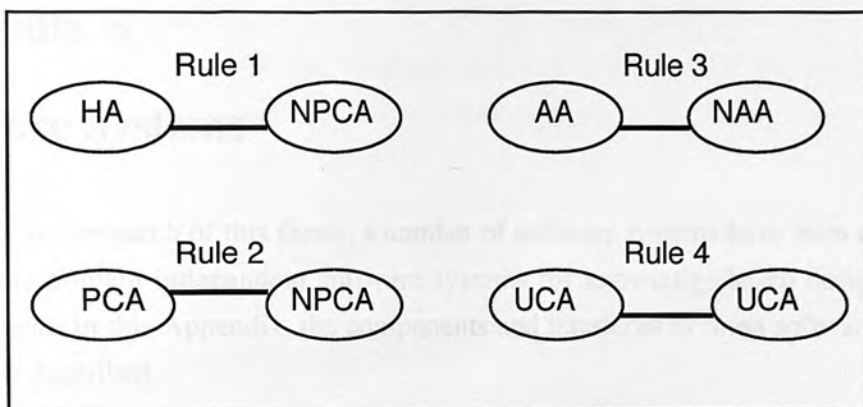


Figure A3: Component Partition Rules

3. Rules for Fragment Assembly

The following rules determine how to assemble molecular components into a molecular fragment:

1. Each aromatic seed component is a molecular fragment on its own.
2. Add all *components* to an *aromatic* seed which are (i) polar and (ii) bonded to the seed, to form an aromatic fragment.
3. Add all *components* to a *hydrophobic* seed which are (i) hydrophobic and (ii) bonded to the seed or connected to the seed via another hydrophobic component to form a hydrophobic fragment.
4. Same as rule 3, but also add no more than one 'proximal' hydrophobic component to the seed to form a hydrophobic fragment.
5. A hydrophobic seed component is a hydrophobic fragment if all bonded hydrophobic components are also aromatic.
6. Add all components to a polar seed component which are (i) polar and (ii) bonded to the seed to form a polar fragment.

Appendix B

Software Systems

As part of the research of this thesis, a number of software systems have been developed as reusable domain independent software systems for knowledge-based design system development. In this Appendix, the components and interfaces of these software systems are briefly described.

1. The Blackboard System

The following Lisp function, shown in Figure B1, illustrates how the blackboard control mechanism works.

```
(defun blackboard (&optional (knowledge-sources 'all))
  (bb-control knowledge-sources)
  (set-slot-value 'bb 'status 'terminated))

(defun bb-control (knowledge-sources)
  (set-slot-value 'bb 'status 'matching)
  (bb-append-agenda (bb-ks-invocation knowledge-sources))
  (set-slot-value 'bb 'status 'firing)
  (do ()
    ((null (slot-value 'bb 'agenda)))
    (when (bb-execute-ksar (bb-modify-agenda))
      (bb-control knowledge-sources))))
```

Figure B1: Top Blackboard Control Function

In this function, 'bb is an instance of blackboard (defined as an object class) and the expression (slot-value 'bb's 'agenda) means accessing instance 'bb's slot 'agenda. The blackboard control program accepts a list of *active design knowledge sources* as arguments. If no argument is supplied then it invokes all the design knowledge sources in the system.

The blackboard control cycle is performed using a recursive function called *bb-control*. The function *bb-ks-invocation* activates the active design knowledge sources and returns KSARs generated by any of them. These KSARs are subsequently managed by the function *bb-append-agenda*. The function *bb-modify-agenda* takes the first KSAR off the blackboard agenda. This KSAR is passed to the function *bb-execute-ksar* which executes it. Whenever a KSAR is executed the blackboard control circle is started again by recursively calling the function *bb-control*. This recursive function terminates when the blackboard agenda becomes empty.

The implemented blackboard control system software consists of a set of object definitions, an agenda control program, a knowledge source activation record management program, a user-directed control program and a graphical user interface. In order to test and evaluate the functionality of this blackboard system, a graphical demonstration system has also been built. This system demonstrates the blackboard control process and provides operations for testing.

The top level Lisp functions in the implemented blackboard control system are given in Table B1.

Table B1. Blackboard Control Functions

Function Name	Function
bb-init	<i>initialise the blackboard</i>
bb-make-ks	<i>instantiate knowledge sources</i>
bb-order-ks	<i>order knowledge sources based on their priorities</i>
bb-ks-invocation	<i>invoke rule-based knowledge sources</i>
bb-control	<i>the major blackboard control procedure</i>
bb-step-control	<i>blackboard control circle debugger</i>
bb-append-agenda	<i>the major agenda operation</i>
bb-claim-results	<i>claim results produced by knowledge sources</i>
bb-insert-ksars	<i>insert ksars into the blackboard agenda</i>
bb-generate-ksar	<i>generate a ksar</i>
bb-execute-ksar	<i>execute a ksar</i>
bb-save-ksar	<i>save a executed ksar</i>
bb-ksar-equal	<i>test whether two ksars are equal</i>

2. The ATMS Interface

Ross's implementation of an ATMS in C consists of 5 files: the major ATMS code, a hash table processing program, an initialisation program, a storage program for storing the justification network and an interface that provides 7 top level interface functions for linking the ATMS to an application.

To use Ross's C-Based ATMS in a Lisp environment³, it is necessary to create a so-called *Foreign Function Environment* (FFE). A Foreign Function Environment defines an interface between the native functions in Lisp and the *foreign functions* in C.

³ Allegro Common Lisp on the Macintosh, version 1.2.3

For example, the following Lisp code creates a Foreign Function Environment shown in Figure B2.

```
(ff-load '( "ff;atms.c.o"           ; atms core,
           "ff;hash.c.o"         ; atms hash table routines
           "ff;init.c.o"         ; atms initialise routine
           "ff;storage.c.o"      ; atms storage routine
           "ff;interface.c.o")    ; atms interface on C side
:ffenv-name 'macl-atms          ; it is called "macl-atms"
:libraries                      ; entries to Libraries
('("clib;StdCLib.o"            ; Standard C Library
  "clib;CInterface.o"         ; C interface Library
  "clib;CRuntime.o"           ; C Runtime Library
  "mpwlib;interface.o"))      ; MPW interface Library
```

Figure B2: Creating a foreign function environment

The function *ff-load* (**foreign function load**) creates a Foreign Function Environment for Ross' C-Based ATMS in the Macintosh Allegro Common Lisp environment on top of which the GoldWorks II software runs. In the above Lisp code *ff* is the logical host of the compiled C-Based ATMS source code. The Lisp equivalent of a C function can be defined within this Foreign Function Environment. For example, the following Lisp code shown Figure B3 links a C function called *c_new_node* with a Lisp function called *lisp-new-node* .

```
(defcfun (lisp-new-node "c_new_node")
  ((fixnum :long) (t :ptr)) :long)
```

Figure B3: Defining a foreign function

The function *defcfun* (**define a foreign c function**) defines the communication between *c_new_node* (a C function) and *lisp-new-node* (a Lisp function) and allows a *fixnum* (integer) to be passed as an argument between the two. The communication between Lisp and C within this Lisp environment is based on integer passing only. More complicated data structures can be represented as lists of lists of integers, or lists of lists of lists of integers etc.

A graphical interface has been built for integrating Ross's ATMS with Lisp-based applications. This interface can also be used to test the functionalities of ATMS and to explain how ATMS works. The software of this interface contains a program for loading, a program for creating the Foreign Function Environment, a set of lisp functions for interfacing the C-Based ATMS and a graphical user interface.

Table B2 lists all these interface functions which are the list versions of those provided in Ross's C-Based ATMS.

Table B2. Lisp version of Ross's ATMS functions

1. (**atms-new-node** *node-id*) *node-id = integer*
 => **t** if *node-id* is an unused integer
 => **nil** if *node-id* is not integer or is already used

2. (**atms-new-assumption** *assumption-id*) *assumption-id = integer*
 => **t** if *assumption-id* is an unused integer
 => **nil** if *assumption-id* is not an integer or is already used

3. (**atms-new-justification** *consequent-id antecedent-ids*)
 => **t** if the justification is valid
 => **nil** if the justification is invalid¹
consequent-id = integer antecedent-ids = list of integers

4. (**atms-see-envdata**)
 => a list of all environments including *nogood* environments in the system.

5. (**atms-see-node** *node-id*) *node-id = integer*
 => a list of lists representing all information about a node.
 => **nil** if the *node-id* is unknown

6. (**atms-get-envs** *node-ids*) *node-ids = a list of integers*
 => a list of lists of assumptions in which the given nodes collectively hold
 => **nil** if any of the *node-ids* is unknown

7. (**atms-get-context** *assumption-ids*) *assumption-ids = list of integers*
 => a symbol of either 'consistent or 'inconsistent and a list of node IDs which hold
 in this context
 => **nil** if any of the *assumption-ids* is unknown

3. The ATMB Architecture

The ATMB architecture software includes a set of ATMB kernel functions, a set of daemons attached to the blackboard agenda, a design documentation system, a foreign function interface, a graphical user interface, an ASCII parser and a design context management system. The ATMB kernel functions are listed in Table B3.

¹ A justification is regarded as invalid if any of the antecedent IDs are zero or equal to the consequent ID. The zero is reserved for representing inconsistency, ie, if a number of antecedents justify a zero consequent, then these antecedents are considered as inconsistent. In this ATMS, no node or assumption can justify itself.

Table B3. ATMB top level Lisp functions

Name	Function
atmb-setup atmb-build-ffenv	<i>setup atmb environment</i> <i>build up foreign function interface for C-based ATMS</i>
atmb-control atmb-get-node-id atmb-get-object-id atmb-get-node-label atmb-get-node-justif	<i>major atmb control circle</i> <i>get ATMS node ids from design objects</i> <i>get object ids from ATMS nodes</i> <i>access to the label of an ATMS node</i> <i>access to justifications of an ATMS node</i>
atmb-new-node atmb-new-assumption atmb-add-justification atmb-display-dkb atmb-modify-dkb atmb-reset-dkb bb-ksar-equal atmb-check-node atmb-check-ksar-format atmb-check-ksar-consistency atmb-check-environment atmb-remove-duplicate atmb-check-object-names	<i>create a new node</i> <i>create a new assumption</i> <i>create justification for a node</i> <i>display the ATMS database in domain terms</i> <i>modify the dynamic knowledge base</i> <i>reset the dynamic knowledge base</i> <i>test whether two KSARs are equal or not</i> <i>check an ATMS node</i> <i>check the format of a KSAR</i> <i>check the consistency of a KSAR</i> <i>check the consistency of an ATMS environment</i> <i>remove duplicated objects</i> <i>check the name of an object</i>
atmb-generate-document atmb-graphical-explain atmb-view-history atmb-repeat-history atmb-new-route atmb-new-branch atmb-switch-route atmb-switch-branch atmb-delete-route atmb-delete-branch atmb-reset	<i>generate text-based design document out of ATMB</i> <i>graphically explain ATMB data</i> <i>view design history record</i> <i>restore design using design history record</i> <i>create a new route of design</i> <i>create a new branch of design</i> <i>switch to a design route</i> <i>switch to a design branch</i> <i>delete a design route</i> <i>delete a design branch</i> <i>reset the atmb kernel</i>

4. The Design Concept Learning System

The design concept learning system contains the following components:

1. A set of object definitions for the design concept tree and concept node;
2. A program for the initialisation and the development of the design concept tree;and
3. A graphical user interface.

The graphical user interface is for the user to select data and the learning strategies. It also graphically displays the design concept tree as it is being built. This graphical user interface has a control panel and a design concept display window.

A graphical image is created to represent the design concept tree. This image has a *position-of-data* slot. When a design concept tree has been created by the design concept learning system, a function is invoked to automatically locate the positions in the graphical display window of various nodes in the design concept tree. This calculation takes into account the depth and the width of the design concept tree and the size of the graphical display window to decide the size and the location of each node automatically.

The design concept tree and the x-y positions of its nodes are stored in the *position-of-data* slot. A drawing function is associated with this slot. Whenever the design concept tree is changed, its image is re-drawn in the graphical display window. A particular node in the design concept tree can be selected when the user clicks on the graphical display window. This is done by comparing the x-y input of the mouse with the x-y positions of those nodes in the design concept tree. When a node is selected, the information associated with it is displayed in a pop-up window.

Table B4 lists the top level Lisp functions of the design concept learning system.

Table B4: Functions of the Design Concept Learning System

Function	Description
load-examples	load examples from file
Specify-control	Specifying the learning strategy
Specify-ranking-attr	Specify the attributes for ranking
Specify-weights	Specify weights for ranking
Specify-linkage	Specify distance measure
Calculate-distance	Calculate the distance between two examples
Create-node	Create a new node in the tree
Merge-nodes	Merge two nodes into one
Generalise-node	Generalise a node
Locate-node	Locate a node in the design concept tree
Calculate-position	Calculate the x-y positions of nodes in the tree
Draw-tree-image	Draw the image of the tree in the display window
Display-node	Display the contents of a node in the tree
Init-tree	Initialise the design concept tree
Build-tree	Build the design concept tree using load examples
Save-tree	Save the current tree into a file
Reset	Clear the system for restarting

Appendix C

The Development Tool, GoldWorks II

1. Introduction

GoldWorks II is a Lisp-based KBS development tool for desktop computers. GoldWorks II's development support facilities are based on different types of objects, namely frame, instance, assertion, relation, attempt, sponsor, agenda item, message passing handlers and daemon etc. with which reasoning and control programs can be developed.

GoldWorks II also includes a dynamic graphics toolkit, i.e., an object-oriented graphical system including a lattice of graphical objects and a set of message passing handlers which can be used together with the lattice; a graphics layout tool which serves essentially as an interface design tool; an interactive object inspector and an object browser; an ASCII parser for building interfaces to external databases.

The software systems developed in this thesis have been developed using GoldWorks II. The following is a description of the major functionalities of GoldWorks II.

2. Knowledge Representation

GoldWorks II's architecture is influenced by the ACTOR1 approach introduced by Carl Hewitt at MIT and is based on a system called MARS (Multiple Assertion Representation System). Within this architecture, the features of rule-based reasoning and object-oriented programming are well integrated and this typifies an important direction in AI programming language that brings together the power of rule-based system's heuristic deduction and the capabilities of object-oriented systems in dealing with complexity.

Programming in GoldWorks II is mainly concerned with creating and manipulating nine different objects, i.e., frames, instances, assertions, relations, rules, rule-sets, sponsors, attempts, and agenda items. Additional Lisp code can be easily attached to any of these objects in order to flexibly manipulate these objects.

2.1 Frames and Instances

GoldWorks II provides a frame-based knowledge representation scheme that includes mainly classes (frames), instances, and methods. It supports multiple-inheritance, i.e., a frame may inherit from more than one parent frame. It is a strongly classed system in which classes and instances are not combined, i.e., a frame provides a template for instances and an instance can only be created from one frame.

In AI applications, an instance may need to be created from more than one class (frame). This can be achieved in GoldWorks II through the use of multiple inheritance feature by firstly creating a multiply-inheriting frame and then creating instances of this frame. GoldWorks II allows objects themselves to be asserted as the values of slots of other instances, thus allowing it to handle composite objects. It also allows a wide range of slot properties to be defined as facets and these facets form a good basis for constraining and attaching small self-contained procedures to the slots of some frames.

Supporting expressive definition of facets in slots is important in knowledge representation and it adds a new dimension to building structured knowledge base. A facet to a slot is like a slot to a frame and it is a property of a slot that further defines it and adds functionality to it. Among all the facets of different type, the most important ones are constraints, default values, when-modified daemon and when-accessed daemon.

The constraints of an object are used to restrict the values or types of values that a slot can have. This prevents the system from being confused with types of data by checking the constraints whenever the value of a slot is asserted so that the new data generated is kept consistent with the intended data type. There are five types of constraint facets, i.e., one-of, range, lisp-type, instance-of, and child-frame-of.

The facets when-accessed and when-modified are important in object-oriented programming. They allow procedural knowledge to be attached to objects. This procedural attachment allows certain actions to be taken as function calls when a slot value is either accessed or modified. The functions defined in these two facets are usually called daemons. Daemons are particularly useful in building causal networks of the objects in a knowledge base to keep the information up to date.

2.2 Relations and Assertions

A GoldWorks II **relation** plays a similar role to predicates in logical programming. GoldWorks relations are used to group factual assertions and to state relationships between various slot values and instances in the knowledge base. Relations provide templates for assertions just as frames provide templates for instances. In GoldWorks II there are three different relations, assertion relations, functional assertion relations, and Lisp function relations.

The use of relations and assertions extends the power of a representation scheme beyond that of a usual rule-based system. There are two types of assertions, unstructured assertions (an unstructured assertion is not linked to any other objects and its pattern can have any number of elements) and structured assertions (those assertions created from slots of instances or other relations). Assertions are especially useful in representing large numbers of factual data. A collection of all unstructured assertions represents something like a Prolog database within which pattern matching can be performed. A structured

assertion has a pattern which can be used in pattern matching or direct manipulated by other at Lisp functions.

GoldWorks II relations and assertions together provide a basis for logical programming and object-oriented programming within a Lisp environment. Inference engines can be written entirely using assertions and relations. In particular, they can also be used in the antecedent part of a rule to test whether the rule is ready to fire. If a relation is defined as a Lisp function relation, then a number of Lisp functions can be called before a rule is fired. This provides a flexibility for those programmers who may prefer Lisp level programming.

In the Multiple Assertion Representation System approach, instance variables and slots are implemented as instance assertions. The inherited objects are built up within the system by creating instance assertions in which instance variable and slot name symbols are identifiable to the slot accessor functions and pattern matching procedures.

3. Rule-based and Object-Oriented Programming

GoldWorks II supports different style of reasoning and provides a wide range of options for developing inference engines. It is possible to develop pure rule-based systems, object-oriented systems, or a combination of both in GoldWorks II.

3.1 Embedded Rules

For rule-based reasoning an expressive rule syntax and three inference engines (forward chaining, backward chaining, and goal-directed forward chaining) are provided. For object-oriented reasoning, handlers and daemons can be defined for object-oriented message passing. The object-oriented message passing scheme is well integrated into the GoldWorks II rule syntax, i.e., messages can be sent to various objects from within rules. GoldWorks II also provides a complete set of built-in slot-accessing methods that allow slot values to be accessed from within rules or hand-written LISP code. This means that GoldWorks II offers two things which are important to developing AI-based systems, i.e., a rule-syntax that can refer to any member of a class with a desired level of generality and with the ability to use variables as slot values in the rules.

3.2 Rule Syntax

The GoldWorks II rule syntax allows rules to refer to instances and slot values using variable names. The bindings established during pattern matching can hold throughout a rule or a set of rules. By using the Lisp function relations and message passing handlers, or by programming directly at the Lisp level from within either 'IF' or 'THEN' part of a

rule, those variable names can be bound to anything in the system. In GoldWorks II rule-based deduction, object-oriented programming, and Lisp programming are integrated. Rules in GoldWorks II also take a number of different options such as dependency, direction, priority, and certainty, each of which has an important role in rule-based reasoning.

3.3 Rule-based Inference

There are three different types of rules: forward, backward and bi-directional rules. Three inference engines are provided for reasoning on these three different types of rules.

Backward chaining is supported in GoldWorks II by using backward rules and attempts. This is a mechanism that centralises and controls the information needed to satisfy a goal-directed search. The idea of using attempts is similar to that of using a Structured Query Language (SQL) in database systems. An attempt has a goal pattern which can be used to match the factual data in the dynamic knowledge base. The backward inference is triggered if this goal matches the consequent part of a backward rule and it is continued by using the antecedent part of the triggered rule as a sub-goal. GoldWorks II allows attempts to be queried in different ways so that a goal can be satisfied either exhaustively or selectively. During backward chaining, new sub-attempts are generated and queried and all the backtracking points are recorded.

A goal-directed search for information in a large knowledge base can also be speeded up significantly when forward chaining is initialised to create assertions that allow further backward chaining to be avoided. Goal-directed forward chaining can be initialised when the pattern of an attempt matches the pattern of a rule-set containing a set of forward chaining rules.

3.4 Object-Oriented Message Passing

GoldWorks II supports object-oriented programming. A number of functions are supplied in the developer's interface in GoldWorks II for object-oriented programming. These functions include *frame-instances*, *frame-parents*, *frame-children*, *all-instances*, *all-assertions*, *slot-value*, *set-slot-value*, and *assertion-matches* etc.

Procedural attachment or method attachment is one of the important features of object-oriented programming. It provides a way for coping with complexity and integrating conventional applications. Methods which are attached to an object can be applied to all the instances of an object and are inherited by those objects which are defined at the lower level of the object hierarchy.

In GoldWorks II, objects are defined as frames. Daemons or handlers are designed as methods to perform the operations on the objects. Handlers are named procedures which

are attached to objects and are invoked via message passing. Daemons have the same functionalities but they are usually automatically invoked whenever the active value of an object is either accessed or modified.

4. Rule-based Control Facilities

Control problems are central to knowledge-based systems, especially for rule-based systems where rules are used as the major way of representing heuristic knowledge. In GoldWorks II, while frames, instances, assertions, and relations provide a good knowledge representation scheme and rules and attempts serve as the basis for writing inference engines, there are three more objects designed for controlling the inferences and system resources. They are rule-sets, sponsors and agenda items.

4.1 Agenda Items

An agenda item is generated whenever a rule successfully matches its antecedent part against the data in the dynamic knowledge base, and is put into an agenda in a certain order according to the priority values of the rules. The execution of an agenda item may result in new data being asserted into the dynamic knowledge base, which in turn, may trigger more rules to generate new agenda items. In forward chaining, a match-fire circle continues to work until such time as the agenda becomes empty. This approach is common to many production systems. What differs with GoldWorks II is that agenda items themselves are implemented as objects and they can be accessed and manipulated by the programmers. This offers some advanced features for controlling the behaviour of the inference. Usually a pending agenda item is taken off the agenda only when its antecedents no longer hold in the dynamic knowledge base as a result of executing some other agenda items. In GoldWorks II, agenda items can also be forced out of an agenda or moved to another agenda by manipulating agenda items and the agenda's sponsors.

4.2 Rule Set

For real applications, matching and firing a large number of rules would consume a large amount of memory and a lot of time. The efficiency of the reasoning can be improved by partitioning rules into rule sets to reduce the overhead for rules that are not applicable to the current problem. In a single-agenda system, there is only one agenda and the only way to control rules is to assign different priority values to different rules so that the agenda items produced can be put into the agenda in an ordered way. With rule-sets, all rules are firstly partitioned into different sets and the activation of these rule-sets is at a higher level than priority values. The rule-set mechanism offers four features:

1. Representing model-based heuristic knowledge: The rule-set mechanism allows heuristic knowledge to be organised into modules, i.e., into sets of independent rules, each of which may represent a different sub-task. Only one set of rules is activated at a time so that rules representing different sub-tasks do not produce conflicting agenda items.
2. Ease of control: With rule-sets, the control of rules can be switched to a higher level user-oriented control of rule-sets. A rule-set can activate or deactivate any rule-sets including itself. This means that meta rules (or control rules) can be put into each rule-set to perform a control task. For example, a lowest priority rule in rule-set X can deactivate itself and then activate the rule-set Y by testing whether all the tasks involving rule-set X have already been completed and whether the conditions for activating rule-set Y have already been met. This can be done explicitly by the user or the system.
3. Initialising forward chaining during backward chaining: A rule-set can have a rule-set pattern that can be queried during backward chaining so that forward chaining rules can be fired to speed up the search for the goal. An important feature of this is that the bindings established in this process hold throughout the rule-sets. This means that rules are not only small chunks of IF-THEN clauses, they can also be effectively organised into modules with which more complicated reasoning can be developed.
4. Allocating system's resources: As an attempt to reduce the number of rules used in the reasoning process, GoldWorks II only considers rules in the activated rule-sets at one time. If a rule-set is deactivated, the rules in this rule-set are considered by the system as if they have not been loaded at all. In this way, the system's resource is effectively allocated to those rules which are relevant to the problem being solved. If a problem is too complicated to solve as a whole, then it can be divided into a chain of sub-problems, which may be more easily approached within the limited system resources. This scheme is similar to the fragmentation of the code in conventional structured programming such as Pascal and FORTRAN and the swapping functions in Common Lisp.

4.3 Sponsors

GoldWorks II also provides an object called sponsor for more sophisticated control and resource allocation by building a sponsor hierarchy.

A sponsor is an object, which has a status to indicate whether it is enabled or disabled, and an agenda in which the agenda items produced by the sponsored rules are

kept. When rules are defined they can be assigned to different sponsors. If a sponsor is enabled, its agenda items will be fired. If there is more than one sponsors, i.e., there is more than one agenda, then they can be structured into a hierarchy. The focus on a particular agenda can then be controlled by enabling and disabling the sponsors.

The enabling and disabling of sponsors can be controlled explicitly by the user or implicitly by sponsor rules themselves. A sponsor has two essential roles:

1. to control when a group of rules assigned to it may or may not fire, and
2. to allocate the system's resources to a particular set of rules.

When an application is running, the system matches all rules that are loaded in, i.e., the system matches rules associated with both enabled and disabled sponsors, but only executes agenda items on enabled sponsors. If all sponsors are enabled, then the execution of the agenda items will travel through the sponsor hierarchy in a depth-first, width-first or an user-defined manner. The system's resources are equally distributed from the top level to the lowest level of the sponsor hierarchy. This mechanism brings inference control and system resource control together as a whole.

However, in many cases it is unnecessary to fire all the rules in the rule base at one time. Again, by disabling all the sponsors but the one which is relevant to the problem, all the system resource will be collected and allocated to this sponsor and this would speed up the execution of the enabled agenda items. Moreover, GoldWorks II does not support the maintenance of multiple contexts.

The control over sponsors and rule sets is essentially the same. A sponsor can be enabled or disabled explicitly by the user or by the system automatically. This is achieved by defining a control rule with a low priority value in each sponsor whose task is to enable another sponsor and then disable itself.

A sponsor hierarchy is suitable for explicit control of a system in which the tasks are well organised into a hierarchy and system needs more one agenda in order to cope with the complicated task. Rule-sets are suitable for controlling a chain events and for speeding up the reasoning process when different rules are involved in a system and share the same objects. It is also flexible in controlling different types of rules (forward, backward, and bi-directional).

In GoldWorks II, the generation of an agenda item is actually triggered by the assertion of some of the facts which match the antecedent of a rule (not by an explicit call to the forward-chain function). If the facts already exist in the system, then the agenda item is generated when a rule is defined or loaded. Therefore, a rule can only be re-fired if the assertions that satisfy the rule's conditions are retracted and re-asserted or the rule itself is deleted and reloaded (or redefined).

5. Dynamic Graphical Toolkit

Most knowledge-based system development tools include some sort of dynamic graphics toolkit for supporting knowledge-based applications. GoldWorks II is equipped with a Dynamic Graphics Toolkit which includes a lattice of graphical objects, a set of message passing handler and a graphic layout tool for designing graphical user interfaces. GoldWorks II's dynamic graphics toolkit has the following features:

Object-oriented Implementation: GoldWorks II's dynamic graphic toolkit provides a lattice of graphical objects (frames) for the users to create graphical instances. Each graphical object in the lattice has a set of methods implemented as message passing handlers. For example, a screen-layout object has four message passing handlers named open, close, delete, and scroll, which can be used to manipulate instances of a screen-layout object. Graphical objects can be consistently used together with other objects in the system and they can be reasoned about in exactly the same way as other objects are reasoned about.

Supporting Active Images: Active images and dynamic graphics are fully supported by the dynamic graphics toolkit. Images can be connected with instances of objects and their changes are co-ordinated, i.e., the change in instance slot values will result in its graphic image being changed as well and a click on an image may trigger a series of user actions on the instances connected with the image. There is a range of facilities for building active images. These include various mouse actions and drawing functions for customising user-defined graphical objects.

Graphics Development Support: Apart from the system defined lattice of graphical object hierarchy, GoldWorks II's dynamic graphics tool kit provides an easy way of extending this lattice by customising user's own graphical objects. For developing new images in applications, user defined graphical objects which inherit the features of the existing graphical objects can be designed. Only two message passing handlers need to be re-defined for each new graphical object, a draw handler and a draw-value handler. Usually the draw handler is for drawing the whole image and the draw-value handler is for drawing only the active part of the image which has connections to some instance slots in the system. The proper use of both, together with some mouse action functions, can achieve a high degree of activeness and sophistication of dynamic graphics. The definition of these new handlers is backed up by a window system called Gold Hill Window (GHW) system and the underlying Macintosh Allegro Common Lisp's Object System. A number of high-level drawing functions can be used to define graphical image handlers or directly draw images on windows.

Most of GoldWorks II facilities have been utilised in the development of the Castlemaine drug design support system in an environment which included Macintosh

Allegro Common Lisp, Macintosh MPW compiler, Macintosh MPW C and GoldWorks

II.

Published Papers

AN INTEGRATED AI SYSTEM FOR CONCEPTUAL DESIGN SUPPORT

Ming Xi Tang

Engineering Design Centre, Department of Engineering
University of Cambridge
Trumpington Street, Cambridge CB2 1PZ
Telephone: 0223 332826, Email: mx2@eng.cam.ac.uk

ABSTRACT

The problem of supporting conceptual design lies in the difficulty of understanding the design process as an intelligent behaviour and modelling this intelligent behaviour in a knowledge-based design support system. The study of a design model which models the nature of design, the development of an intelligent design support system architecture and which this design model can be realised in an integrated computational environment, the identification and integration of AI-based reasoning techniques that can support design problem solving, and the effective implementation of such an architecture in design applications, are some of the key issues in developing a knowledge-based design support system.

This paper describes a knowledge-based functional modelling system that is being developed at the Engineering Design Centre in Cambridge University. This system is an embodiment of an AI-based design support system architecture, a functional synthesis engine and an embodiment design system. It is intended to support functional synthesis, conceptual generation and kinematic analysis in an integrated knowledge-based environment. Four design knowledge systems are developed in this integrated system: a model reference engine; a rule-based functional synthesis engine; a constraint propagation engine for embodiment generation; a simulated annealing engine and a genetic algorithm engine for constraint based optimisation.

Keywords: functional synthesis, embodiment design, constraint management, genetic algorithm, simulated annealing

1 INTRODUCTION

Design problems are typically ill-structured: they start with a design requirement which is often vague, or imply all the criteria by which acceptable solutions can be completely defined. Such a design requirement may be incomplete, inconsistent, impossible or ambiguous. An important characteristic of design therefore involves developing a

Proceedings of 1995 Lancaster International Workshop on AI Systems Support for Engineers, Design Research Centre, Lancaster University, Lancaster, UK, 1-3 March 1995, Ambleside, UK.

DEVELOPMENT OF AN INTEGRATED AI SYSTEM FOR CONCEPTUAL DESIGN SUPPORT*

Ming Xi Tang

Engineering Design Centre, Department of Engineering
University of Cambridge
Trumpington Street, Cambridge CB2 1PZ
Telephone: 0223 332826, Email:mxt@eng.cam.ac.uk

ABSTRACT

The problem of supporting conceptual design lies in the difficulty of understanding the design process as an intelligent behaviour and modelling this intelligent behaviour in a computer-based design support system. The study of a design model which models the process of design, the development of an intelligent design support system architecture upon which this design model can be realised in an integrated computational environment, the identification and integration of AI-based reasoning techniques that can support design problem solving, and the effective implementation of such an architecture in design applications, are some of the key issues in developing knowledge-based design support systems.

This paper describes a knowledge-based functional modelling system that is being developed at the Engineering Design Centre in Cambridge University. This system is an integration of an AI-based design support system architecture, a functional synthesis system and an embodiment design system. It is intended to support functional synthesis, embodiment generation and kinematic analysis in an integrated knowledge-based environment. Four design knowledge sources are developed in this integrated system as main inference engines: a rule-based functional synthesis engine, a constraint management engine for embodiment generation, a simulated annealing engine and a genetic algorithm engine for constraint-based optimisation.

Key words: functional synthesis, embodiment design, constraint management, genetic algorithms, simulated annealing.

1 INTRODUCTION

Design problems are typically ill-structured: they start with a design requirement which does not contain, or imply all the criteria by which acceptable solution can be completely identified. Such a design requirement may be incomplete, inconsistent, imprecise or ambiguous. An important characteristic of design therefore involves developing a

* In Proceedings of 1995 Lancaster International Workshop on AI System Support for Conceptual Design, (Eds.) John Sharpe, March 1995, Ambleside, UK.

complete and consistent design requirement at the same time as developing a solution that satisfies it. This requires an integrated design environment in which domain concepts, design objects, design heuristics and design materials, are classified and structured so that their functional, structural and causal relationships can be identified and effectively reasoned about, allowing high level design tasks, such as design synthesis, design analysis, design evaluation, and design modification to be carried out whenever the design requirements are specified or changed.

Supporting design tasks at the early stage of the design process is difficult because at this stage little is known about the inconsistency of the design requirement and the structure of the design problem. The concept of the design is not formed until a designer has identified a basic knowledge structure in which information about structural, functional and causal relations of design objects becomes available and organised. The central role of an AI system supporting conceptual design is to provide solutions, through proper representations and reasoning mechanisms, for a class of design problems where the design requirement is incomplete and inconsistent, and the design problem is largely under-constrained.

Although it is possible to develop tools that automatically perform design tasks that require intensive numerical calculation, it is more useful to build design support systems that combine human and machine intelligence. Automatic design tools can only support some well defined and isolated design tasks which rely on procedural knowledge and fixed design strategies. These tools contribute little to decision making in the early stage of the design process which requires intensive use of both heuristic and procedural knowledge.

This paper presents an integrated AI system for conceptual design support. In this system, functional reasoning methods are used to transform an initial design requirement, using knowledge transformation rules, to an initial design solution concept structure consisting of a set of connected design elements which satisfy the initial design requirement. This knowledge transformation process is supported by a representation and reasoning scheme developed by Chakrabarti [2]. The results of functional synthesis provide a basis for embodiment design during which the physical components realising the proposed initial design solution concept structure are identified and their geometric relationships established. In this stage the values of all design variables are specified and explored using a constraint management system utilising genetic algorithms and simulated annealing techniques [4, 7]. The results of the embodiment design and kinematic analysis provide the information for evaluating the conceptual design results. An integrated application of functional reasoning, constraint management and kinematic analysis techniques is one of the current research themes at the EDC in Cambridge University. The aim is to develop an AI-based design support system capable of deriving conceptual design solutions from functional design requirements [1]. In this paper, some

of the existing systems used in this integration are reviewed. Based on this review the blackboard architecture of an integrated functional modelling system is presented. The design knowledge sources in this system are then described in the context of this blackboard architecture. Finally, some implementation issues of using a knowledge based system development tool are discussed.

2 A KNOWLEDGE-BASED APPROACH TO CONCEPTUAL DESIGN

Knowledge-based design techniques provide rich knowledge representation methods that suitably represent and manipulate different types of design objects, design heuristic knowledge, and design constraints; they provide sufficient inferencing support and control mechanisms for designers to perform design tasks, including design synthesis, evaluation and optimisation; and provide mechanisms for detecting and resolving conflicts among various design requirements and design constraints.

The following requirements are identified in developing the integrated functional modelling system:

- It should help designers to construct and extend the design knowledge base within which domain concepts, design objects, dependency information of design objects are well structured and consistently maintained.
- It should help designers to derive solutions quickly from initial, not necessarily complete and consistent, design requirements. In other words it should provide an efficient mechanism to transform an initial design requirement description to a final design specification.
- It should provide explicit explanations and justifications for any chosen aspects of the current status of a design, not only in terms of how something has been derived, but also to explain why something is not happening as expected. Locating areas of difficulty and suggesting strategies for solutions contribute to good decision making in design.
- It should allow designers to vary data, design requirements, problem solving strategies, or evaluation criteria, to obtain alternative designs. Simply speaking, a knowledge-based design support system must support multiple-context problem solving so that a design problem can be explored from many different directions.
- It should provide mechanisms for capturing and refining design knowledge so that the design knowledge base can be incrementally enlarged and enhanced. In other

words, a knowledge-based design support system must have some facilities for learning.

AI methods are used in the development of the integrated functional modelling system to deal with the problems of: how to break down a design task and represent the hierarchy of components or sub-systems in a formal and natural way; how to effectively manipulate a potentially large set of design variables and constraints; how to maintain the knowledge generated during design and to detect inconsistency; and how to create and maintain multiple design solutions.

2.1 ATMB, a knowledge-based architecture for design support

Assumption-based Truth Maintained Blackboard (ATMB) is a knowledge-based design support system kernel developed in Edinburgh [5, 6]. The ATMB kernel was built on the percept that design problems at the early stage of design process are ill-structured. It supports the representation and exploration, using a number of AI-based computational techniques, of domain and design knowledge necessary for driving design solutions from an initial, not necessarily complete design requirement description.

In a knowledge-based design support system, alternative design solutions need to be explored and retained in order to compare them in contexts where design criteria, design requirements and design methods differ. In mathematical terms, features of design can be described by a set of design variables and a set of dependent design parameters. The values of design variables and dependent design parameters are determined by a set of constraints in which these variables and parameters are logically or causally related to each other. Design variables and dependent design parameters are used when it is necessary to distinguish the part of a design problem which is flexible to change (described by design variables) from other parts of the design problem (described by dependent design parameters) which are relatively dependent on design variables. It is assumed that the values of dependent design parameters can be determined by the values of the design variables through constraint propagation.

Based on this formulation, the structure of a design problem can be determined by the identification of the relationship between design variables and dependent design parameters. The constraints, in most cases, are different forms of mathematical equations, rules, or reasoning modules. These constraints typically interact in complex manners.

During design exploration, many plausible selections arise in the presence of under-constrained design variables, giving rise in turn to many plausible values of dependent design parameters. Furthermore the way in which design variables and dependent design parameters are constrained may depend on what design strategy (or method) will be employed. In other words, using a different design problem solving strategy may result

in the same set of design variables and dependent design parameters being differently constrained. A design solution is a complete set of values for all the design variables and dependent parameters which jointly describe the features of the design problem and satisfy the constraints and the design requirement.

The ATMB architecture uses a frame-based representation scheme for building design knowledge bases. A design knowledge base contains design object class definitions which partially defines the spaces of possible designs for small and independent design problems. The multiple relationships and dependencies among design concepts and design objects enable them to be used together to jointly define the structure of new design problems. Such a design knowledge base provides design knowledge in a behavioural, functional and structural vocabulary. A structural relationship between design objects determines how objects are physically connected and how, for example, characteristics of a system object can be derived from its constituent component objects; a functional relationship between design objects relates objects in terms of their performances or behaviour, and their relevance to a particular design requirement and design task; a causal relationship between two design objects decides how one object might depend on another and the consequences of any change in either object.

The ATMB architecture uses a combination of computational techniques to provide general support to knowledge-based design applications. It consists of a blackboard control system, an assumption-based truth maintenance system, a design documentation system, and a graphical explanation system [6, 8]. In this architecture an assumption-based truth maintenance system (ATMS) is integrated with a blackboard control system to maintain the consistency of the design knowledge generated, and to manage the exploration of design contexts when *design requirements*, *initial design data* or *design methods* are changed. The ATMS-based design context management system has three primary purposes for design support: to maintain consistency in the design knowledge base; to maintain multiple design contexts; and to be used as a search control mechanism to reduce the number of combinations. It is particularly suitable for design exploration because of its incremental updating of the dependency networks and its ability to maintain a multi-contextual environment to allow different design solutions to be explored simultaneously [5].

The ATMB architecture is self-contained with a design documentation system which maintains design history records; and a graphical explanation system which provides context-dependent explanation of newly derived design knowledge or any design difficulties. The ATMB mechanism as a basic architecture for knowledge-based design applications has been used the Edinburgh Designer System (EDS) and the Castlemaine Design System [5, 6].

2.2 FUNCISION, a functional synthesis system

FUNCSION is a system developed by Chabrabarti at the EDC in Cambridge University that performs *instantaneous synthesis* of design concepts. FUNCSION was intended to support two design tasks in the early stage of the design process:

1. instantaneous synthesis of design solution concepts based on functional requirements;
2. temporal reasoning of behaviour of a design solution concept [2].

FuncSION as a functional synthesis system is capable of deriving an initial design solution concept structure based on a design requirement description and a goal function. When the design requirement function and the goal function can be defined in advance as input and output of a transformation system, the system's roles are: to search for valid elements whose input functions match the requirement functions; and then to apply predefined transformation rules to these elements until a structure that satisfies the goal function can be derived. FUNCSION is basically a qualitative reasoning system and it has been implemented using Harlequin's KnowledgeWorks.

The important concepts in FUNCSION are *element* (predefined building blocks for design synthesis) which defines the type and constraints of a physical component, *transformation rules* which transform one function to another, *input vector* (functional requirement) and *output vector* (goal function). Both requirement function (input) and goal function (output) are represented as vectors which have *kind*, *orientation*, *sense*, *position* and *magnitude* as attributes. A solution concept is defined as an abstract description of a system of identifiable individual elements which can satisfy the given functional requirements [2].

Functional synthesis of design solution concept is performed first by a kind-synthesiser (reasoning on the kind attribute of input/output vectors only). This kind-synthesis process searches for a set of elements which satisfy the kind attributes of the input/output vectors. A result of this kind-synthesis is a causal network of elements in which transformation rules apply. The termination condition of the kind-synthesiser is specified by the user as the maximum number of transformations to be allowed.

The kind-synthesiser generates a list of candidates, each of which specifies a causal relation between the elements concerned. These candidates are then further specialised by checking their orientation and sense configurations using the orientation and sense transformation rules. The results of this are a number of initial design solution concept structures, each of which satisfies the kind, orientation and sense requirements.

2.3 CADET, a design embodiment tool

Computer Aided Design Embodiment Tool (CADET) is a system developed by Thornton at the EDC in Cambridge University for supporting embodiment design using systematic constraint satisfaction techniques. CADET provides facilities for: (1) constraint specification by building a product model using a library of generic functional components; and (2) constraint satisfaction by solving the design problem represented by the product model using either simulated annealing techniques or genetic algorithms [7].

A product model in CADET consists of a set of generic components connected through the relationships of either two-components interface or three-components interface. Each component has variables and pre-defined initial constraints. The initial constraints associated with any component are reasonably well understood and domain independent. A product model, once built by the user through an interactive user interface, will have a set of design variables which are constrained by those pre-defined initial constraints and any new constraints added in by the user while building up the product model. In particular, each component has a 3D geometry information associated with it. This process is referred to as constraint specification [7].

Much of the CADET's facilities deal with the problem of constraint specification during which a designer builds up a product model, using a library of components. Once a product model is built, the CADET system generates a text-based file containing design variables and constraints. This text-based file can then be used in the constraint satisfaction stage when two programs are invoked to search for a set of values of design variables that do not violate any design constraints.

During the process of constraint satisfaction, the constraint set presented by the product model is first simplified by a constraint evaluation program in order to exclude any constants and equalities, resulting in a reduced set of constraints. This reduced set of constraints is then regarded as an optimisation problem with the goal of the optimisation being defined by a constraint violation function that gives an account of how many constraints are violated. This optimisation problem is subsequently solved using either a Genetic Algorithms program (GA) or a Simulated Annealing program (SA). Both GA and SA use the same goal function, i.e., the constraint violation function in the search for a consistent set design variable values without any constraint violation. The GA program uses a two-points masked crossover strategy in its reproduction process while the SA uses a fixed strategy to change its search direction (this fixed strategy sets the values of some of coefficients used in the simulated annealing program). Both GA and SA terminate when either the goal function is satisfied to an extent, or a local minimum has been reached. This indicates that there is no guarantee that an optimised solution will be found using either GA or SA methods.

3 AN INTEGRATED ARCHITECTURE FOR FUNCTIONAL MODELLING

The ATMB architecture was used in the DtoP Project and the Castlemaine system [6]. Both FUNCISION and CADET have demonstrated the capability of providing knowledge representation and reasoning support to early stage design tasks when little is known about the basic structure of the design problem. These systems, however, do not operate in an integrated and consistent way. The need for integrating these systems arises from the fact that, with a unified knowledge representation scheme and an intelligent control system, they can form a more powerful tool for the tasks of functional synthesis of design requirement, embodiment design of product structure and kinematic analysis of product behaviour. In order to build such a system, it is necessary to abstract useful features of ATMB, FUNCISION and CADET and then to redevelop them in the form of an intelligent design support system architecture and domain independent design knowledge sources (or inference engines). The integrated functional modelling system is intended to be used by designers as a conceptual design tool in the domain of mechanical engineering.

CADET's approach to design constraint satisfaction is characterised by two features, i.e., its inclusion of 3D geometry information in its generic component definitions, and its coupling of constraint satisfaction and optimisation methods (GA and SA methods) in the embodiment design process. While CADET has been developed as a self-contained design embodiment tool and has demonstrated the sufficient capability of solving a number of design problems, several problems need to be addressed in order to develop it further as a domain independent design knowledge source.

CADET successfully used genetic algorithms and simulated annealing in constraint satisfaction in a number of design examples. However, further effort needs to be made in order to establish how these methods can be used as domain independent design knowledge sources. Two important research problems need be further addressed when transferring the facilities in CADET into a general design knowledge source:

1. how to expand its product model by incorporating more complicated knowledge structures above the component level,
2. how to validate its optimisation-based (use of GA and SA in combination with constraint evaluation) approach to design constraint satisfaction.

The product model definitions in CADET need to be extended as the current ones cannot be used for functional synthesis, design embodiment and kinematic analysis. The lack of definition and classification of constraints in CADET makes it difficult to know what kind of constraints can be solved. Little heuristic design knowledge is utilised in the

process of constraint satisfaction and optimisation (Both GA and SA use fixed coefficients in their algorithms). No explanation is provided when the system gets into difficulty.

One approach to enhancing CADET is to combine heuristic-based constraint management with search-based optimisation methods. This approach is more interactive than the current CADET system and therefore it is possible to utilise more design heuristic knowledge in the design embodiment stage. The automatic search methods such as simulated annealing and genetic algorithm are only used when the network of constraints has been explored and simplified by the user.

This requires the development a more powerful constraint management system capable of performing algebraic manipulation of a potentially large set of complex constraints in order to provide a good basis for using GA and SA methods in the design optimisation process. It is also important to incorporate a rule-based reasoning system into CADET so that it can reason on design heuristics or use assumptions. This rule-based system which utilises design heuristics can simplify the constraint set before the GA and SA methods are invoked.

In the integrated functional modelling system, FuncSION system is also enhanced by enlarging its database of standard elements and transformation rules, and by integrating this database with the system's knowledge base where elements can be extended to have the links to a component knowledge base. A new interactive user interface is developed for FuncSION that allows the functionality of the system to be fully utilised and explored by a designer. The original inference mechanism in FuncSION is embodied with its control program. This is extended and redeveloped using a rule-based approach for representing the transformation rules.

1 The Blackboard architecture

The architecture of an integrated AI system identifies the necessary AI components for intelligent design support. An integrated application of FuncSION, CADET, a kinematic analysis package [4] and AI-based methods forms an important part of a research theme at the EDC towards the development of an Integrated Design Framework (IDF) [2]. In this architecture, the ATMB kernel controls FuncSION, CADET and a kinematic analysis package as independent design knowledge sources.

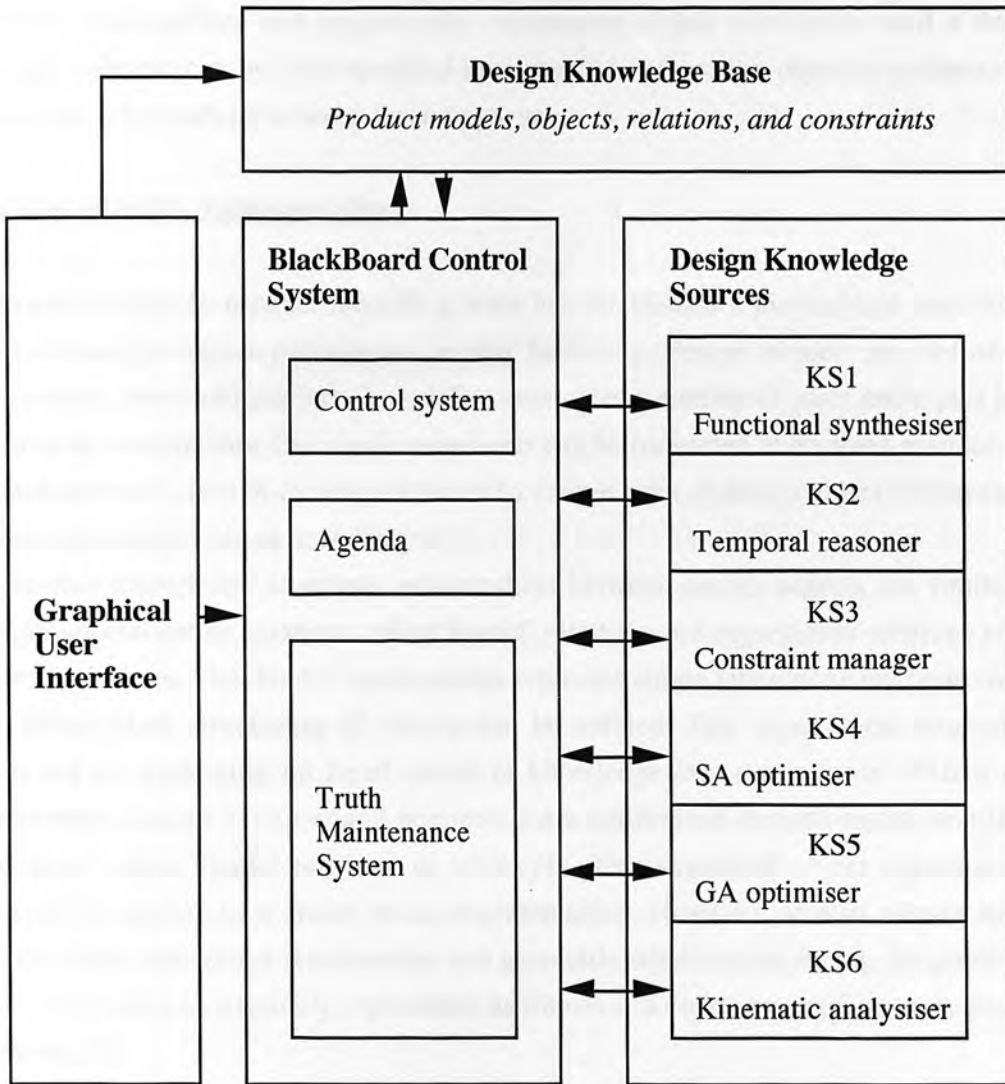


Fig 1. An Integrated AI Architecture for Functional Modelling

The architecture of the integrated functional modelling system, as illustrated in Figure 1, consists of a design knowledge base containing product models which can be shared by FUNCTION, CADET and a kinematic analysis program; a blackboard upon which design knowledge sources are controlled on an opportunistic basis [3]; a number of design knowledge sources performing functional, design embodiment and kinematic analysis; and a graphical user interface capable of explaining design results, design history and producing design documents.

In this architecture, the system's design knowledge sources (KS), controlled and maintained by the ATMB kernel, interact with a designer to perform design tasks incrementally. The designer's actions such as initiating design data, specifying design constraints, and choosing design problem solving methods are transformed into the ATMB kernel as design assumptions. The system maintains the knowledge derived based

on these assumptions and supports the exploration of this knowledge until a design concept solution can be fully specified in terms of functionality, physical attributes and geometric relationships between its components.

3.2 Knowledge representation

Research in EDC's product modelling team has established a hierarchical structure of generic design object definitions. In this hierarchy, design objects are defined as component, parts and artefact. An artefact assembles a number of parts and a part has a number of components. Parts and components can be connected in complex manner. The instantiation of a part of this object hierarchy creates a set of design object instances that can be explored to derive new designs [1].

In this knowledge structure, relationships between design objects are limited to simple specialisation relations called 'kindof' relations and aggregation relations called 'partof' relations. The 'kindof' relationships represent single inheritance relations so that the hierarchical structuring of objects can be defined. This hierarchical structure is achieved by packaging up small pieces of knowledge into components. Within each component, design variables and parameters are constrained through linear, non-linear equations, rules, spatial relations or tables etc. This structured object representation scheme is similar to a frame-based representation. However, it also allows spatial relationships, functional relationships and geometric relationships among design objects and variables to be explicitly represented as frames and object-oriented message passing methods [5].

The design knowledge base contains design object classes and the reasoning methods associated with each object class. For example, a component has the attributes of input/output function, type of transformation for functional synthesis purpose. The type of transformation determines how a component transfers an input to output. A component also has attributes to describe its physical and geometric features for the purpose of embodiment design and kinematic analysis.

Three kinds of reasoning modules are attached to object classes. These are: demons, constraint handlers, and forward chaining rules. Demons are functions attached to object slots. Whenever a change is made to an object instance, the demons attached to the object class are invoked to carry out inference to determine the consequence of this change within the object instance. Constraint handlers are used to propagate changes from one object instance to the others which are physically or causally related to the object instance being changes. Rules are used to perform special purposed reasoning such as orientation synthesis and constraint management.

3.3 Blackboard controlled constraint management

Constraint management is the central problem solving method in the integrated functional modelling system. It decides how the values of design variables and dependent design parameters change when assumptions are made by the designers whilst exploring an initial design solution concept structure.

In the integrated functional modelling system, embodiment design is carried out in an incremental way. The components in the design knowledge base are first selected by a designer to build up an instance of a product model. Alternatively such an instance of a product model can be derived by the functional synthesis knowledge source based on the initial user requirements. An instantiated product model only constrains the design solution space in terms of the components used, their physical connections and orientations. This needs to be further explored in the embodiment design process to determine the values of all the variables without violating any constraints. Internally, an instantiated product model is represented as a network of variables and constraints.

While genetic algorithms and simulated annealing methods can be used to carry out a systematic search once a design problem has been transferred into a network of design variables linked by constraints of different forms, it is more desirable in the integrated functional modelling system to combine systematic search methods with heuristic reasoning methods. In such a system the designer's input is monitored by the control system and the system infers as much as possible whenever the designer makes any changes to the constraint network. This process is supported using a blackboard control system.

A blackboard model represents a highly structured control scheme, and a special case of opportunistic problem solving. It has a simple architecture consisting of a number of knowledge sources, a blackboard data structure and a control system which controls knowledge sources on an opportunistic basis. The working memory of the system is viewed as a blackboard where the communication between different independent knowledge sources takes place. The blackboard provides globally available data structures for all the knowledge sources and controls them in a systematic way. Opportunistic control strategy of the blackboard system is probably most appropriate for a design support system [5, 6]. Such a control strategy contributes to design support tasks in the following ways:

- it allows a design support system to work as a self-organising system whose problem solving knowledge sources can respond dynamically and differently to design situations;

- it encourages the development of independent and self-contained design knowledge sources, thus making it easy to integrate design knowledge of different forms with the other parts of a design support system; and
- it has a control mechanism that is suitable for different knowledge representation schemes, ie, any thing that can be treated as a black box such as a rule set, an internal message passing handler, or an external software package, as long as it is self-contained.

The blackboard control system consists of an agenda and a number of knowledge sources. The agenda contains all the design variables to be determined. Whenever a component is included in the instantiated product model, the variables in that component are put by the system onto the blackboard agenda. A variable remains on the blackboard agenda until its value has been decided. Whenever the system receives an input from the user, the system verifies this input and then take the necessary actions. In order for the system to do so, user actions in this stage are classified into:

1. add a new design variable,
2. add a new constraint concerning several variables,
3. specify the value of a variable,
4. delete a variable,
5. delete a constraint.

The first action only adds a new agenda item onto the blackboard agenda and triggers no system's action. The second action triggers the system to check whether any values can be derived for the variables concerned in the newly added constraint. In general adding a constraint contributes to tightening the design solution space and reducing the number of variables to be solved. The third action also triggers the system to carry out constraint propagation during which the newly added value of a variable is propagated throughout the constraint network. The deletion of a variable or a constraint also results in serious work of the system in retreating the results that have been derived based on the deleted variable or constraints. These five user actions and the relevant blackboard inferences reflect an interactive process in which the system actively supports a designer in trying to solve a constraint-based design problem.

The system also explains, at any point of the design process, how variables and constraints are related, what are the unsolved variables and remaining constraints, and what are the likely consequences if the value of one of the remaining variable is specified or changed. In this way, the system helps designers to know what has been derived, what are the remaining problems, and what to do next. Throughout this process the designers still play a decisive role in exploring the design concept. The blackboard control system actively supports this decision making by regularly checking the blackboard agenda and then taking the necessary actions.

If this process does not result in all the variable being solved, then the remaining constraint network can be transferred into a search problem. Simulated annealing methods or genetic algorithm can then be used to search the solution space by defining a utility function based on the number of constraints that are still being violated [7]. With the support of the blackboard controlled constraint management system, the search space will be more confined than it is originally defined by the instantiated product model.

3.4 Design knowledge sources

Design knowledge sources in the integrated functional modelling system are self-contained programs controlled by the blackboard control system. These design knowledge sources perform inferences and return results which are subsequently managed by the blackboard control system in a design document. An important feature of them is that they share a common data structure on the blackboard and they act opportunistically and independently. Each design knowledge source consists of: a preconditions part which decides when it should get invoked; and an action part that actually performs a specified task.

In the first version of the integrated functional modelling system, the following design knowledge sources are included.

1. a rule-based knowledge source performing functional synthesis based on Charkarabarti's method [2]. This knowledge source defines generic element of conceptual design in terms of name, input, output, and type of transformation. The type of transformation of an element determines how an element transfers an orientation or sense input to output. In this design knowledge source, knowledge about these transformations are represented as 84 production rules.
2. a constraint manager performing constraint-based reasoning for embodiment design purposes. This design knowledge sources creates a constraint network from an instantiated product model, and then simplifies and evaluates it. This design knowledge source also propagates any changes made by the user throughout the

constraint network. In the process of embodiment design it acts as a main inference engine.

3. a simulated annealing program performing constraint-based search. This design knowledge source is being developed by modifying the program used in CADET so that: the control factors of the annealing algorithm can be adjusted by the user if necessary; and more complicated constraints can be dealt with.
4. a genetic algorithm program performing automatic search for feasible solutions in a potentially huge space of possible designs. This is used as an alternative to knowledge source 3 and it is a Lisp version of the genetic algorithms used in CADET.
5. a kinematic analysis program performing kinematic analysis of any design proposals returned by any of the above design knowledge sources. This design knowledge source is based on a matrix reduction method developed by Johnson in Cambridge University [4].

These design knowledge sources are integrated with the blackboard control system and a graphical user interface to complete a design session when an initial design requirement description in terms of input/output function is given by a designer.

4 IMPLEMENTATION

One of the major concerns of the implementation is to combine rule-based and object-oriented representations to represent and reason about product models of different complexity. The use of an AI-based tool ensures that quick prototyping can be achieved to provide feedback to the architecture and design knowledge source design. An initial prototype of the integrated functional modelling system has been implemented using a knowledge-based tool called GoldWorks III on a SUN SPARC station 5 running Solaris. GoldWorks III is a Lisp-based tool for developing knowledge-based applications. It supports both rule-based and object-oriented programming. It supports knowledge representation via eight different objects, i.e., *frame*, *instance*, *assertion*, *rule* (forward and backward chaining), *attempt* (backward chaining), *agenda item*, *relation*, and *sponsor*. It has an object-oriented graphic system for developing graphical user interfaces. GoldWorks III Sun version runs on top of Lucid Common Lisp which has interface to C or C++ applications.

Work is currently being focused on the consolidation of the above discussed design knowledge sources to achieve a high degree generality in mechanical engineering design.

This initial prototype is to be further developed in order it to be connected with the knowledge base developed at the EDC's product modelling group. A knowledge base of mechanical components and reasoning methods have been developed by this group in C++. In order to combine the integrated functional modelling system and the knowledge base of the components. An interface between C++ and Lisp is being implemented.

5 CONCLUSIONS

Future design applications are likely to be supported by sophisticated design support system tools that employ the best available AI techniques. The integration of a blackboard inferencing control system, an ATMS-based design context management system, a design documentation system and a graphical explanation system forms the core of an intelligent design support system. The development of four domain independent inference engines enables it to be used as a tool to support constraint-based design applications.

Supporting the tasks in the early stage of the design process is always difficult and challenging. The integrated functional modelling system presented in this paper integrates a number of AI-based systems that supports knowledge representation and intelligent control, functional synthesis of design requirements, embodiment design as constraint satisfaction, and optimisation using simulated annealing or genetic algorithms. In the integrated functional modelling system, a more systematic and knowledge-based approach has been adopted in order to fully demonstrate the potentials of the current systems and to develop them further.

6 ACKNOWLEDGEMENTS

The work presented in this paper is currently being funded by the EPSRC and the author wishes to thank all EDC members, especially Dr. Nigel Ball, Dr. Amaresh Chakrabarti, and Dr. Tim Murdoch at the EDC for their support in the development of the presented system.

REFERENCES

- [1] Ball, N. et al, 1992, "The Integrated Design Framework: Supporting the Design Process Using a Blackboard System", *AI in Design*, 92.
- [2] Chakrabarti, A et al, 1994, "A Two-step Approach to Conceptual Design of Mechanical Device", *AI in Design*, 94.
- [3] Hayes-Roth, B., 1985, "Blackboard Architecture for Control", *Journal of Artificial Intelligence*, 26:251-231, 1985.

- [4] Johnson, A. L et al, 1993, "Modelling functionality in CAD: implications for product representation", in Proceedings of the 9th International Conference on Engineering Design, 1993.
- [5] Smithers, T., Conkie, A., Doheny, J., Logan, B., Millington, K. and Tang, M., 1990, "Design as Intelligent Behaviour: An AI in Design Research Programme", *International Journal of Artificial Intelligence in Engineering*, 5.
- [6] Smithers, T., Tang, M., Ross, P. and Tomes, N., 1993, "An Incremental Learning Approach for Indirect Drug Design", *International Journal of Artificial Intelligence in Engineering*, Special Issue on Design and Machine Learning, Vol. 8, 1993.
- [7] Thornton, A., 1993, "Constraint Specification and Satisfaction in Embodiment Design", PhD Thesis, University of Cambridge, Department of Engineering, 1993.
- [8] [de Kleer 1986] de Kleer, J., "An Assumption-based TMS", *Artificial Intelligence*, Vol. 28. 1986.

An Inductive Learning System for Intelligent Design Support*

Ming Xi Tang

Engineering Design Centre, Department of Engineering

University of Cambridge

Trumpington Street, Cambridge CB2 1PZ, UK

Abstract. This paper presents an approach to intelligent design support that incorporates inductive learning methods into a knowledge-based design model and an integrated knowledge-based design support system architecture. This approach explores the role of learning in exploratory design and integrates design heuristics with inductive learning methods in a knowledge-based design support system. A class of design tasks with initially poorly known structure is well supported. This approach is demonstrated through the development and implementation of a knowledge-based design support system kernel and the its application to small-molecule drug design: a realistic design problem where identification of a pharmacophore (initial structure of a design problem) is induced from existing example molecules.

1. Introduction

Learning and designing are closely related activities: finding a new design solution involves the use of knowledge abstracted from previous designs; searching for an alternative design strategy can be guided by the experience gained from previous design failures; evaluating a design solution relies largely on the knowledge generalised from the features or simulated behaviour of the design.

In the early stage of the design process there is a great deal of uncertainty about the design problem, constraints; and solutions. Faced with the these uncertainties, designers need to discover or learn, through the study of the existing design examples and new design requirements in order to:

- determine what it is possible (defining design goals);
- describe and constrain the design problem (defining the design space); and
- evaluate the design (defining evaluation criteria).

The potential of inductive learning techniques such as *concept formation* for intelligent design support is widely recognised [2, 3, 5, 9]. In this paper an inductive learning approach to intelligent design support is presented. In this approach a class of design

* In Proceedings of 1995 Scandinavian Conference on Artificial Intelligence, June, 1995, Trondheim, Norway.

tasks is formalised as learning the concept structure of a design problem from design examples. An incremental inductive learning system is developed and implemented to build a *design concept tree*. Any concept node in the induced design concept tree can be explored further to obtain a new design. The system has been implemented using a Lisp-based tool and integrated with an AI-based design support system kernel. In this paper this incremental inductive learning system and its integration with an AI-based design support system are described through an application in the domain of small molecule drug design. Before describing the learning system and its implementation, the concepts and the design problem in the domain of small molecule drug design are introduced in section 2.

2. Drug Design

Rational drug discovery is based upon the precept that the pharmacological activity of a *drug molecule* is a direct consequence of the drug molecule's binding to the target *receptor molecule*. That is, when a drug molecule binds to a receptor molecule, some biological change takes place.

2.1. The Lock-and-key Problem

Drugs that 'fit' better to a receptor will bind more strongly with improved activity. A *lock-and-key* metaphor can be used here: the drug '*key*' is specifically designed to fit and to operate the receptor '*lock*'. In drug design, the structure of the receptor and its binding site can be entirely unknown or perhaps only partially postulated in terms of its important features.

Designing a drug molecule (key) without the explicit knowledge its receptor (lock) is termed *indirect drug design* which is the majority of drug design practice. In indirect drug design, the structure of the design problem is a description of required receptor binding features called the *pharmacophore*. This pharmacophore is a hypothetical model of a receptor in terms of its arrangement of chemical properties. It is characterised by the example molecules (keys) which are thought to be necessary for binding specifically to the receptor (lock).

There are two important tasks in indirect drug design:

1. Inducing common features of example molecules which are known to bind to a particular receptor; and
2. Designing a chemical structure which is to be synthesised based on the pharmacophore obtained.

2.2. Drug Design Concepts

This section describes and explains the concepts in indirect drug design that are relevant to the inductive learning system presented later.

2.2.1. Molecules

A molecule is represented by its structure and some other major attributes such as its *activity*, *assay type*, etc. A molecule is traditionally represented as a two-dimensional structure (see Figure 1). A computational representation called a *Smiles String* has become a major way of representing molecular structures in a computer system. A smiles string denotes the structure of a molecular structure using a string which enables it to be easily manipulated by a computer program [10].

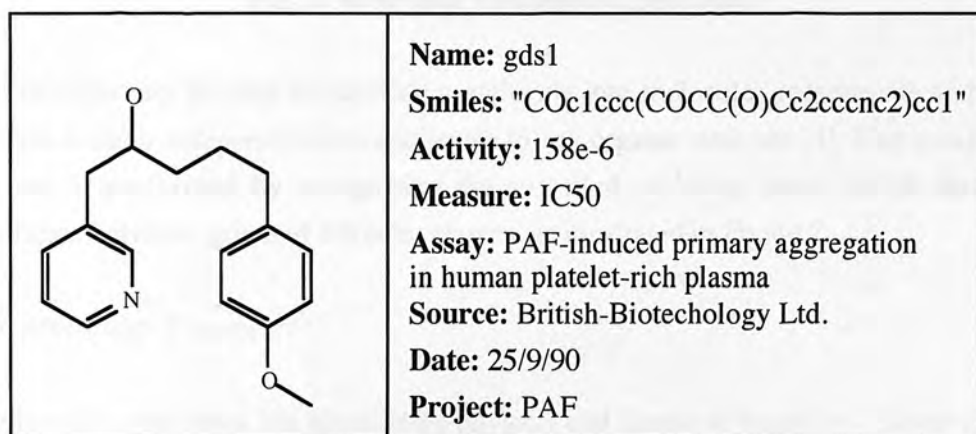


Fig. 1. Structure and attributes of a molecule

2.2.2. Molecular Components

A molecular component is defined as the smallest unit containing recognisable chemical functions, ie, they are self-contained units with a physical and chemical identity and properties such as *numbers-of-atoms*, *planner*, *polar* etc. Partitioning molecules into constituent components to derive physical and chemical properties is essentially how medicinal chemists deal with chemical structures. The properties of a molecule are derivable from the properties of the molecular components joined together to form the molecule.

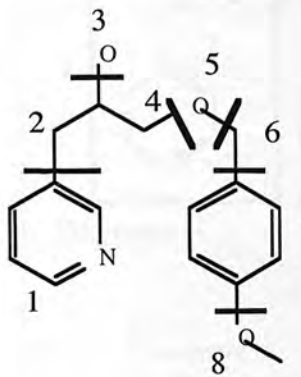
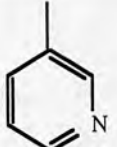
<p>— Isolating bond</p> 		<p>Component number: 1 Connected_to: 2 Smiles: "c1ccncl" Property-model: pyridine</p>																
<p>Pyridine</p> <table border="0"> <tr><td>ring</td><td>yes</td></tr> <tr><td>planar</td><td>yes</td></tr> <tr><td>flexible</td><td>no</td></tr> <tr><td>hydrophobic</td><td>yes</td></tr> <tr><td>polar</td><td>no</td></tr> <tr><td>no-of-h-acceptors</td><td>1</td></tr> <tr><td>no-of-h-donors</td><td>0</td></tr> <tr><td>no-of-atoms</td><td>6</td></tr> <tr><td>aromatic</td><td>yes</td></tr> </table>	ring	yes	planar	yes	flexible	no	hydrophobic	yes	polar	no	no-of-h-acceptors	1	no-of-h-donors	0	no-of-atoms	6	aromatic	yes
ring	yes																	
planar	yes																	
flexible	no																	
hydrophobic	yes																	
polar	no																	
no-of-h-acceptors	1																	
no-of-h-donors	0																	
no-of-atoms	6																	
aromatic	yes																	

Fig. 2. Molecular component (pyridine)

A set of rules can be used to partition a molecule into molecular components and these rules are *domain independent* and applicable to any organic structure [4]. This component partition is performed by recognising the so-called *isolating bonds* which mark the boundaries between atoms of different classes, as illustrated in Figure 2.

2.2.3. Molecular Fragments

A molecular component has identifiable physical and chemical properties. However, it is not necessarily the correct unit for describing a molecule's binding to a receptor. An additional object is needed to represent the part of a molecule that describes the molecule's binding features. This object is called a *molecular fragment* which is a small collection of molecular components recognisable by a receptor in a co-operative manner.

For example, two components (comp4 and comp6), as shown in Figure 3, are combined to form a fragment (F9). This fragment will principally demonstrate a hydrophobic feature when binding to the target receptor because both comp4 and comp6 are hydrophobic. A fragment has a *primary property* as the major receptor binding feature, and a number of *secondary properties* describing fragment's remaining chemical properties. Again, rules exist for assembling molecular components into fragments [4]. It can be seen from Figure 4 that some of the fragments are overlapping. While there are 8 components in the given molecule, 12 fragments are derived.

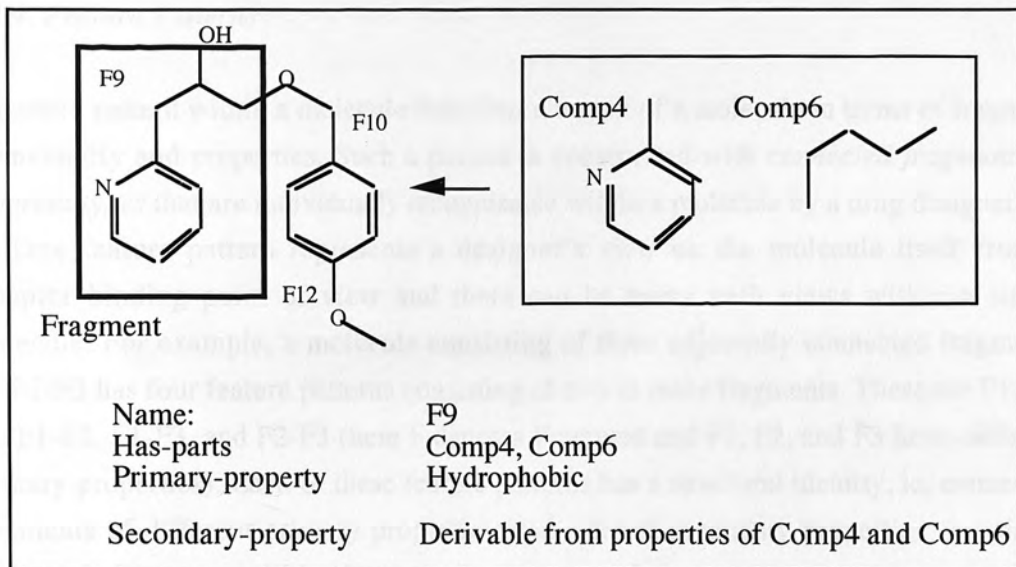


Fig. 3. A molecular fragment

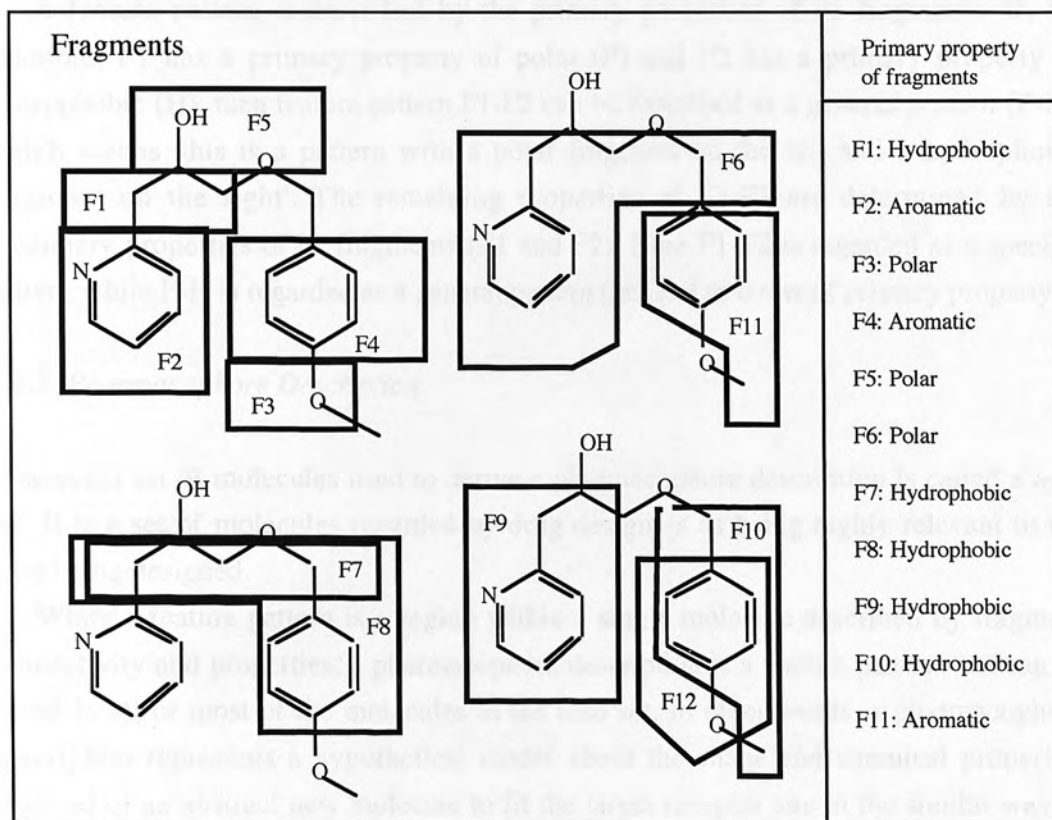


Fig. 4. Multiple fragments

2.2.4. Feature Patterns

A feature pattern within a molecule describes a region of a molecule in terms of fragment connectivity and properties. Such a pattern is constructed with *connected fragments of different types* that are individually recognisable within a molecule by a drug designer.

One feature pattern represents a designer's view on the molecule itself from a receptor binding point of view and there can be many such views within a single molecule. For example, a molecule consisting of three adjacently connected fragments F1-F2-F3 has four feature patterns consisting of two or more fragments. These are F1-F2-F3, F1-F2, F1-F3, and F2-F3 (here F denotes Fragment and F1, F2, and F3 have different primary properties). Each of these feature patterns has a structural identity, ie, connected fragments of different primary properties, and a set of secondary properties associated with each fragment within the pattern. Any one of these patterns might be used to describe the receptor binding feature of the original molecule. Therefore every feature pattern must be considered as a candidate for generating a pharmacophore description.

A feature pattern is described by the primary properties of its fragments. If, for example, F1 has a primary property of polar (P) and F2 has a primary property of hydrophobic (H), then feature pattern F1-F2 can be described as a *general pattern* (P-H), which means 'this is a pattern with a polar fragment on the left and a hydrophobic fragment on the right'. The remaining properties of F1-F2 are determined by the secondary properties of its fragments (F1 and F2). Here F1-F2 is regarded as a specific pattern while P-H is regarded as a general pattern (general in terms of primary property).

2.2.5. Pharmacophore Description

A selected set of molecules used to derive a pharmacophore description is called a *lead set*. It is a set of molecules regarded by drug designers as being highly relevant to the drug being designed.

While a feature pattern is a region within a single molecule described by fragment connectivity and properties, a pharmacophore description is a feature pattern that can be found in all or most of the molecules in the lead set. In other words, a pharmacophore description represents a hypothetical model about the shape and chemical properties required of an abstract new molecule to fit the target receptor site in the similar way as those molecules in the lead set. A pharmacophore description is abstracted from the common features of the molecules in the lead set and used as a requirement description for the new molecule being designed. As illustrated in Figure 5, a pharmacophore description contains the following information:

- the feature pattern of the pharmacophore;
- the number of features it has;

- the primary properties of those features;
- the fragments which compose the pharmacophore description; and
- the constitution of the fragments in the feature pattern.

A good pharmacophore description should be extracted from all or most of the molecules in the lead set, which are known to bind well to a receptor. That is, it should highlight the common features by ways of measuring similarities between the examples molecules. One molecule can bind to a receptor in different ways, ie, a molecule will have multiple feature patterns as a result of fragments overlapping (see section 2.2.3. for the definition of molecular fragment). Therefore many candidate pharmacophore descriptions can be derived from one lead set, but only the one abstracted from all or most of the molecules is regarded as being useful. The basic idea of pharmacophore generation in indirect drug design is therefore to find out the similarities among all or most of the molecules in the lead set, and to describe them in terms of their receptor binding features. This is where inductive inference can be used.

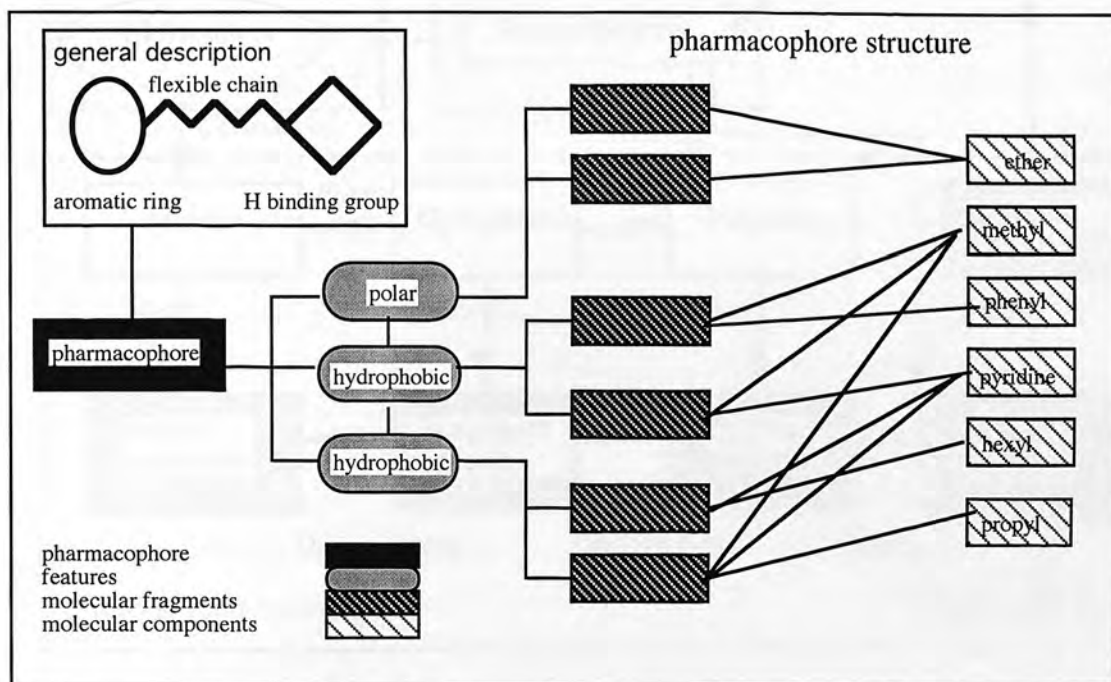


Fig. 5. Representation of a pharmacophore

3. An Inductive Learning System for Drug Design

Deriving a pharmacophore description from a lead set of molecules is the main task in the indirect drug design process that can benefit from inductive learning techniques. An inductive learning system has been implemented by the author based on a knowledge-based drug design system developed in the Castlemaine project [6, 7, 8].

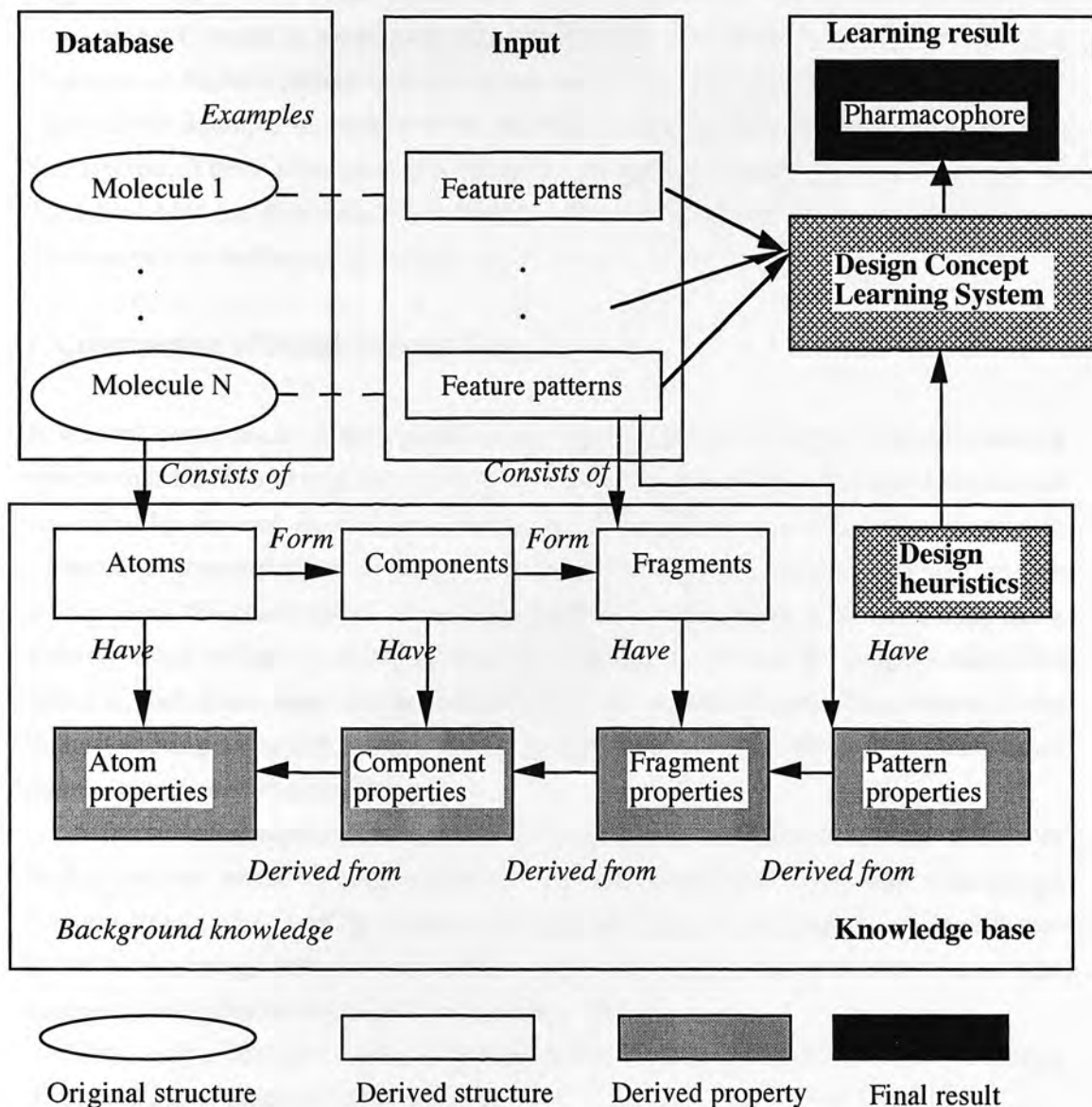


Fig. 6. An inductive learning scheme for drug design

In this inductive learning system the design of a new drug is modelled as a process involving two closely related activities: the inductive learning of a design problem structure (pharmacophore description); and the exploration and modification of this

design problem structure to obtain new solutions (specification of novel molecules through isostere replacement).

Figure 6 highlights the learning process and the relationships between the original example molecules and the pharmacophore description to be learned. An important feature of this approach is its combination of inductive inference and design heuristic knowledge. In order to achieve this, intermediate representations, such as components, fragments and feature patterns, are used for each molecule. For example, a molecule consisting of atoms is partitioned into components. One or more components form a fragment. A feature pattern is an arrangement of fragments. Feature patterns identified from all the example molecules in the lead set are used to describe the pharmacophore description. These structural relationships are used as *background knowledge* to determine how the properties of the objects are derived and how the common features of the example molecules are to be induced.

4. Construction of Design Concept Tree

A central component of the inductive learning scheme is a design concept learning system that builds a design concept tree incrementally until all the selected examples are classified by the tree. Each node in the design concept tree represents a class of feature patterns abstracted from a subset of example molecules, whilst the whole tree incorporates the whole set of example molecules in an organised way. Each node has a scoring value which is calculated from the numbers of molecule examples classified under it, and some other design criteria which are described later. The node with the highest scoring value can be regarded as the best pharmacophore description while other nodes are regarded as alternatives.

A frame-based representation is used to construct the design concept tree. All nodes in this tree are linked through a *parent-node* and *child-node* relationship. The design concept tree is initialised by creating an empty top node. In the process of incremental inductive learning, instances of concept nodes are created as new feature patterns of example molecules are supplied to the learning system.

The design concept learning system performs one of the following operations whenever a new example molecule is supplied:

- creating a child-node if an instance is to be further classified;
- creating a new branch if a new feature pattern is encountered;
- generalising a node if a new instance can be classified under it; and
- specialising a node if an instance under it is retreated.

The induced design concept tree represents a hierarchy of concepts of chemical objects, with the higher nodes in the tree representing more general concepts.

If a training example has multiple interpretations (ie, if it contains instances of more than one class), then it will be classified under more than one node. To evaluate the learning results, a concept scoring method is devised to give a numerical account of how training examples are distributed and how the concept associated with a node is being measured by some pre-defined evaluation criteria. Providing different weighting factors for the calculation of the scores may result in the concept tree being interpreted differently.

5. Learning Strategies

Whenever a new feature pattern is classified under a node, the concept description associated with that node must be further generalised. Two specific feature patterns with different topological arrangements are structurally different and cannot be directly compared with each other. Two structurally similar patterns can however be compared with each other in terms of the closeness of their secondary properties.

A nearest neighbour strategy is used here to devise a *knowledge source* that calculates a numerical value to indicate the similarity of secondary properties of two specific feature patterns. For example, suppose a general feature pattern G1 is derived from molecule M1 and is associated with a concept node N1 in the initial concept tree. G1 has a number of specific feature patterns (S11, S12, S13, S14,, S1m), each of which consists of different fragments with different secondary properties. Suppose G2 is a general feature pattern that is derived from a newly added molecule M2 with specific feature patterns (S21, S22, S23, S24,, S2n). If G2 can be classified under node N1, this means that G1 and G2 have the same number of fragments and the same general feature patterns. However, they may have different specific feature patterns with different secondary properties. Then concept node N1 must be generalised to include two similar specific patterns, one from each molecule, in order to describe the features that M1 and M2 have in common.

To generate a concept description that includes both G1 and G2, a nearest neighbour strategy is used to find a pair of specific patterns, one from G1 and one from G2, that give the shortest *Euclidean distance*. Generalisation is done by using transformation rules that inductively combine two features from two examples into one. After the generalisation, the concept associated with node N1 may be stated as something like "*a general feature pattern G1=G2 can be found in molecule M1 and M2 with the constituent specific patterns S11 and S21 (suppose S11 and S21 give the shortest Euclidean distance), and the secondary property ranges of the concept are determined by the fragments in S11 and S21*".

The major transformation rule used to modify the secondary properties of a generalised pharmacophore description is the so called *closing interval rule* [2]. This rule states that the expression $[a = a_x, a_1 < a_x < a_n]$ is less general than the expression $[a = a_1$

,....., a_n], where a is an attribute of an example. The Euclidean distance generalisation process discussed above is actually trying to find the tightest possible range of a_1 to a_n for each of the secondary properties.

A second rule deals with attributes that have *yes/no* values. This rule states that expression [$a = \text{yes}$] is more general than expression [$a = \text{yes}$ or $a = \text{no}$]. This rule only applies to the domain of drug design where *yes/no* values can be logically OR-combined. This means that if any of the attributes is *yes*, then the result of generalisation is *yes*. For example, this rule typically answers the question '*do any of the fragments have this property at all ?*'.

As a result of fragment overlapping (see section 2.2.3. for the definition of a molecular fragment), one molecule may have many feature patterns. Therefore one molecule may be classified under more than one node in the design concept tree. This indicates that there will be *multiple clusters* of pharmacophore descriptions. It is necessary to evaluate them in order to select the best description for all or most of the example molecules in the lead set.

Any induced node in the design concept tree can be evaluated using a scoring method. This method counts how many examples are classified under a given node. In addition, it takes into account of the following considerations:

- the most active molecules in the lead set;
- the number of different fragments in the feature pattern;
- features with tightly-constrained secondary property ranges;
- the number of tightly-constrained properties; and
- covering most example molecules in the lead set.

These are desirable features of a pharmacophore description from a design point of view. Based on these features a scoring method is devised which gives each node in the design concept tree a numerical account of its *goodness and fit* in the context of the above evaluation criteria. The node in the concept tree that scores the highest value can then be chosen as the final pharmacophore description for the design of a new drug.

6. Conclusions

The task of inducing a pharmacophore description from a set of example molecules is by nature an inductive learning task. The inductive learning system presented in this paper addressed the issue of using inductive learning techniques in intelligent design support and demonstrated the use of inductive inference in a realistic design application. In this approach design activities are supported by:

1. using background knowledge to pre-process the original examples to ensure that the learning system only derives feature patterns useful for design purposes.

2. providing a way of linking the induction of pharmacophore descriptions with the design considerations by giving weighting factors to the learning algorithm. The nodes in the design concept tree have different scoring values when different weights are given. The attempt to link a learning scheme with a particular design intention is useful when designers want to look at different aspects of the original examples.
4. allowing the designers to see how the design concept tree is affected by the newly introduced molecules, or to see how the deletion of a molecule affects the structure of the concept tree. The learning process is incremental and each node in the concept tree can be explained after each molecule is introduced.

The developed design concept learning program has been integrated with a knowledge-based design support system architecture called Assumption-based Truth Maintained Blackboard (ATMB). The ATMB has been intended to form a basic kernel for knowledge-based design applications [1, 6, 8, 11, 12].

The inductive learning system presented in this paper is domain specific. To be used as a general machine learning tool in a knowledge-based design support system, some more work is required. The current indirect approach to drug design and the learning scheme developed for pharmacophore identification can be enhanced by integrating it with a proper chemical database where enough information about molecules, their properties, isosteric information and other information about structure-activity relationships at component level, or fragment level etc, can be obtained.

Acknowledgements

My work in the Castlemaine project was supported by Dr. Tim Smithers and Dr. Peter Ross in the Department of Artificial Intelligence, University of Edinburgh. I would like to acknowledge Logica Cambridge Ltd, British Bio-technology Ltd. and CamAxys Ltd. for their involvement and financial commitment in the Castlemaine project during which a knowledge-based drug design support system was built. My current work at the Engineering Design Centre in Cambridge University is funded by EPSRC. Finally I would like to thank Mark Nowack for his proof reading of this manuscript.

References

- [1] de Kleer, J., "An Assumption-based TMS", *Artificial Intelligence*, Vol. 28. 1986.
- [2] Dietterich, T. G. and Michalski, R. S., 1981, "Inductive Learning of Structural Description", *Artificial Intelligence*, (1981) 257-294.
- [3] Fisher, D. H. and Pazzani, M., 1991, "Computational Models of Concept Formation", in *Concept Formation: Knowledge and Experience in Unsupervised Learning*, Fisher, D.(Eds.), Morgan Kaufmann.

- [4] Hodgkin, E., 1991, "The Castlemaine Project, Development of an AI-Based Drug Design Support System", Workshop of Molecular Graphics Society, Essex, UK, 1991.
- [5] Reich, Y., Konda, S. L., Levy, S. N., Monarch, I. A., and Subrahmanian, E., 1993, "New Roles for Machine Learning in Design", Special Issue on Machine Learning and Design, International Journal of Artificial Intelligence in Engineering, Vol. 8, 1993.
- [6] Smithers, T., Conkie, A., Doheny, J., Logan, B., Millington, K. and Tang, M., 1990, "Design as Intelligent Behaviour: An AI in Design Research Programme", *International Journal of Artificial Intelligence in Engineering*, 5.
- [7] Smithers, T., Tang, M., Ross, P. and Tomes, N., 1993, "An Incremental Learning Approach for Indirect Drug Design", International Journal of Artificial Intelligence in Engineering, Special Issue on Design and Machine Learning, Vol. 8, 1993.
- [8] Smithers, T., Tang, M., Tomes, N., Buck, P., Clarke, B., Lloyd, G., Poulter, K., Floyd, C. and Hodgkin, E., 1992, "Development of a Knowledge-Based Design Support System", International Journal of Knowledge Based Systems, March, 1992.
- [9] Stepp, R. E., Michalski, R. S., 1986, "Conceptual Clustering of Structured Objects: A Goal-Oriented Approach", *Artificial Intelligence*, 28, 43-69.
- [10] Weininger, D., 1989, "Smiles. 3: Deepict. Graphical Depiction of Chemical Structures", *Journal of Chemical Information*, 1990.
- [11] Tang, M. X., "Development of an integrated AI system for conceptual design support", 1995 Lancaster International Workshop on Engineering Design: AI System Support for Conceptual Design, 27-29, March, 1995, Ambleside, UK.
- [12] Tang, M. X., "Exploring Design Solutions using a Context Management System", IJCAI'95 Workshop on Representing and Reasoning about Context, August, 1995, Montreal, Canada.

EXPLORING DESIGN SOLUTIONS USING A CONTEXT MANAGEMENT SYSTEM*

Ming Xi Tang

Engineering Design Centre
Department of Engineering, University of Cambridge
Trumpington Street, Cambridge CB2 1PZ, UK
Telephone: 0223 332826, Fax: 0223 332662, Email: mxt@eng.cam.ac.uk

ABSTRACT

Design problems are ill-structured and design decision process is incremental and nonmonotonic. In design applications, multiple design solutions or alternative design solutions arise in the presence of under-constrained design variables and parameters. The exploration of these multiple design solutions are context dependent, ie., design solutions are derived from a space in which *design requirements*, *design methods* and *design criteria* are subject to frequent change. Exploring and maintaining multiple design solutions is therefore an essential task for a computer-based design support system.

ATMB is a lisp-based software architecture which identifies the necessary components (or sub-systems) of a computer system for intelligent design support, and creates a computational environment within which these components are integrated to provide general design support functions. These functions include the management of a design knowledge base, control of design knowledge sources, the creation and maintenance of multiple design contexts, and graphical explanation of design results. A central component in the ATMB is a design context management system based on an integration of a blackboard control system and an Assumption-based Truth Maintenance System (ATMS). In this paper, the issue of modelling context in intelligent design support is discussed first. An important sub-system of the ATMB, ie., its context management system is then described.

1. INTRODUCTION

One of the most frequently carried out tasks in design is to make plausible modifications to part of an initial design solution to observe the repercussions in order to find multiple design solutions. Design decision making process is *nonmonotonic*, *constructive* and *incremental* because designers often change strategies when information about the consequence of earlier decisions becomes available and the design problem space becomes more and more constrained. In a knowledge-based design system, alternative design solutions need to be explored and retained in order to compare them in contexts where *design criteria*, *design requirements* and *design methods* differ.

Traditionally, multiple design solutions are explored by human designers using an approach known as the Analysis of Interconnected Decision Area (AIDA) [Jones 1992]. This approach employs a generate-and-test strategy to identify alternative design solutions. It works in five steps:

1. Identify the decision areas in a design problem structure by focusing on some interesting design variables and dependent design parameters.
2. Identify several feasible options in each decision area within its isolated context.
3. Indicate which options are incompatible with others.
4. List that can be combined together to form candidate compatible designs.
5. Evaluate the candidate designs using some quantitative measures in order to choose the acceptable ones.

Two problems arise when using this approach in a computer-based design system: first a design problem is not easily separated and the decision areas are not clear to designers until

*Presented at the IJCAI (International Joint Conference on Artificial Intelligence) Workshop on Modelling and Representing Context in Knowledge-based Systems, August, 1995, Montreal, Canada.

considerable effort has been made to explore them; second some design problems are too complicated to afford a generate-and-test strategy, ie, the derivation of dependent design parameters from design variables in a complex system is too expensive to search blindly for their value space. This view is supported by noting that in a knowledge-based design support system:

- design needs to be incrementally conducted;
- designers need the freedom to manoeuvre around a huge design space, but cannot afford the expensive generate-and-test strategy;
- it is necessary for incompatibility in design requirement, constraints and evaluation criteria to be discovered early during the design process;
- designers need to know the immediate consequence of any change either in input data, design method or design evaluation criteria; and
- designers need explicit explanations of the design results derived from any design decisions, or the reasons for any difficulties in reaching a satisfactory design solution.

These issues can be dealt with in a knowledge-based design support system using a consistency maintenance and context management system, the central role of which is to maintain the consistency of the knowledge generated during design and to support the exploration of this knowledge when design context changes. In a knowledge-based design support system, a design context can be defined based on the following three different sources of knowledge:

- knowledge loaded from a design knowledge base in the form of instantiated design objects and constraints;
- knowledge provided by the designer in the form of design decisions (value assignments or design method selections etc.); and
- knowledge inferred by the system as a result of propagating the values of design parameters or variables throughout the whole set of data and constraints already held in the system.

When any of these knowledge sources is changed or modified, the design context is changed and an alternative design solution may be derived based on the new context. The role of a design context management system is: to retain as much of the knowledge generated during the design process as possible; to provide easy access to, and a good explanation of this knowledge; to make the best use of the knowledge already held in the system to enable it to generate new knowledge without performing redundant inferencing; and to help the designer compare different, sometimes conflicting design solutions.

2. MODELLING CONTEXT IN DESIGN SUPPORT

2.1 Design as constraint satisfaction

Edinburgh Designer System (EDS) is an AI-based design support system developed in the Alvey large demonstrator project 'Design to Product' [Smithers et al 1990]. In EDS, design variables and constraints are structured into module class definitions. A module class definition declaratively represents a generic component in the domain of mechanical engineering design. Module class definitions can be instantiated and assembled to form new designs. Four inference engines are used to support the design process during which a designer makes assumptions on the values of the design variables embodied in the instantiated module class definitions. These inference engines are an *algebraic manipulation engine* (AME) solving algebraic expressions; a *spatial relationship engine* (SRE) reasoning on spatial relationships between geometrically related design objects; a *geometric modelling engine* (GME) performing geometric shape and space occupancy reasoning; and a *relational manipulation engine* (RME) inferring upon tabular data.

EDS supports multiple-context problem solving by using an Assumption-based Truth Maintenance System (ATMS) [de Kleer, 1984; Smithers et al 1993; Logan et al 1992]. The ATMS mechanism is used in the EDS to maintain all the derived information based on different and even conflicting assumptions made by a designer. Any derived value of a design variable is justified by an assumption or a number of assumptions. All the assumptions and their derivatives are recorded in a Design Description Document (DDD) from which design solutions can be sorted out based on their ATMS justifications.

However, as design variables in EDS are globally visible to all the inference engines, maintaining the DDD with a large number of design variables can become computationally expensive and inefficient.

Galelio2 [Bowen et al 1992] is a constraint programming language that supports multiple context problem solving in life-cycle engineering using the concept of '*multiple perspectives*'. In Galelio2, design variables and constraints can be viewed from different perspectives by the people involved in design, manufacturing and maintenance etc. A perspective is a small set of design variables and constraints. The variables within each perspective can be declared as being *local* or *overwritable*. The local variables within a perspective can only be modified by the person who created the perspective whilst overwritable variables can be changed from within any other perspectives. Any change to an overwritable variable in any perspective triggers a so-called negotiation process during which people who created the affected perspectives are asked to resolve the potential conflicts.

Nonmonotonic reasoning is supported in Galelio2 by the use of overwritable variables in multiple perspectives and by the reasoning support to the process of negotiation. However, the original context is deleted once an overwritable variable is overwritten and the new value is propagated throughout the constraint network. In other words, the information associated with an earlier context is not maintained, even though it could still have potential value as an alternative design solution.

2.2 Formulating context in design support

In a knowledge-based design system, features of design can be described by a set of *design variables* and a set of *dependent design parameters*. The values of design variables and dependent design parameters are determined by a set of constraints in which these variables and parameters are logically or causally related to each other. Design variables and dependent design parameters are used when it is necessary to distinguish the part of a design problem which is flexible to change (described by the design variables) from other parts of the design problem (described by the dependent design parameters) which are relatively dependent on design variables. A design solution is a complete set of values for all the design variables and dependent design parameters which jointly describe the features of the design problem and satisfy the constraints.

Based on this formulation, the space of design solutions can be determined by the identification of the relationships between design variables and dependent design parameters. The constraints, in most cases, are various forms of mathematical equations, rules, or reasoning modules. In exploring the design space described by such a network of constraints and variables, many plausible selections arise in the presence of under-constrained design variables, giving rise in turn to many plausible values of dependent design parameters. Furthermore the way in which design variables and dependent design parameters are constrained may depend on what design strategy (or method) will be employed. In other words, using a different design problem solving strategy may result in the same set of design variables and dependent design parameters being differently constrained.

Design exploration involves exploring the convergence of all the design variables. During the exploration of alternative design solutions, any value assignment to any of the design variables in the constraint set is treated as an assumption (or a design decision). A design solution is derived as a result of the system's inferencing from designer's assumptions. It is based on the designer's choice of initial data, design method or design procedure and design evaluation criteria. These choices form the basis for defining design context for which computational support can be provided.

Inconsistency arises in a constraint-based design system when conflicting assumptions are being made. An inconsistency may occur in any one of the following design situations:

- when two different values are assigned to the same design variable; and
- when a particular value assignment to a design variable results in the constraint set being violated;

For example, if there is a constraint set $\{a + b = 2\}$ where a and b are two design variables, then assumption $[b = 1]$ and $[b = 2]$ are considered as inconsistent because nothing can

be derived from $b = 1$ and $b = 2$ jointly. However, results may still be usefully derived from $[b = 1]$ or $[b = 2]$ separately, giving two possible solutions: $[a = 1]$ or $[a = 0]$. Here solution $[a = 1]$ is justified by assumption $[b = 1]$ and $[a = 0]$ by $[b = 2]$.

2.3 Managing design contexts using an ATMS

A number of truth maintenance or reasoning maintenance techniques have been developed in AI to deal with the problem of inconsistency and context management [de Kleer 1984], [Doyle 1979]. An assumption-based truth maintenance system (ATMS), based upon de Kleer's work, offers sufficient facilities for working with inconsistent information, and for multiple context problem solving. ATMS is particularly suitable for design exploration because of its incremental updating of a dependency network and its ability to maintain a multi-contextual environment to allow different design solutions to be explored simultaneously [Smithers *et al* 1990; Smithers *et al* 1993; Banares-Alcantara 1991].

In a design situation, a design context means something like "what circumstance is this design solution derived from?" The role of an ATMS in context management may be demonstrated through the following simplified example. Design diameter (d) and the design speed (n) of a motor are supposed to be constrained by the following two separate constraints, each of which may be derived when employing a different design method:

$$\text{Constraint1: } d = 9.81 * \text{sqrt}(n)/2 \quad (1)$$

$$\text{Constraint2: } d = 0.3 * \text{sqrt}(n)/2 \quad (2)$$

When designing a motor a designer might wish to explore the possible values of both d and n . It is supposed that the speed can be regarded as a design variable and the diameter as a dependent design parameter. If two assumptions on the values of the speed n are made by the designer then there can be four basic assumptions in the ATMS database:

$$\{a_1\} \quad d = 9.81 * \text{sqrt}(n)/2 \quad (\text{original constraint 1})$$

$$\{a_2\} \quad d = 0.3 * \text{sqrt}(n)/2 \quad (\text{original constraint 2})$$

$$\{a_3\} \quad n = 125 \quad (\text{design decision 1})$$

$$\{a_4\} \quad n = 200 \quad (\text{design decision 2})$$

The following nodes can be derived from these assumptions by the design support system's knowledge sources:

$$n_5: \quad d = 54.83 \quad \text{justification of } n_5 = \{\{a_1, a_3\}\}$$

$$n_6: \quad d = 1.67 \quad \text{justification of } n_6 = \{\{a_2, a_3\}\}$$

$$n_7: \quad d = 69.36 \quad \text{justification of } n_7 = \{\{a_1, a_4\}\}$$

$$n_8: \quad d = 2.12 \quad \text{justification of } n_8 = \{\{a_2, a_4\}\}$$

This gives the designer some contexts to consider. So far there are six contexts, two of which are inconsistent $\{\{a_1, a_2\}\}$ and $\{\{a_3, a_4\}\}$ because two values of the same design variables, or two constraints alone cannot derive anything meaningful to the design problem. Nodes 5, 6, 7, and 8 are all held in consistent contexts. The value of $d = 54.83$ (node 5) is justified by, and is the consequence of the existence of a_1 and a_3 . In other words, the antecedents of node 5 is assumptions a_1 and a_3 . If later the designer finds out that a_3 is incompatible with other knowledge in the system, then node 5 will no longer be justified.

ATMS has been adopted in a number of knowledge-based design support systems for consistency maintenance and context management [Smithers *et al* 1990], [Logan *et al*

1992], and [Banares-Alcantara 1991]. However, no attempt has been made so far in developing a generic architecture that utilises the ATMS for knowledge-based design applications.

3. AN ATMB KERNEL FOR CONTEXT EXPLORATION

ATMS as a general belief revision system has two primary functionalities: to maintain consistency of a knowledge base; and to support multiple context problem solving. In a knowledge-based design system, four problems need to be addressed when employing the ATMS mechanism for consistency and context maintenance:

- determining the granularity of the ATMS network;
- defining the meaning of context;
- developing inference mechanisms for exploring contexts; and
- providing explanation of contexts.

Assumption-based Truth Maintained Blackboard (ATMB) is a knowledge-based design support system kernel developed by the author to address the above problems. The ATMB has been developed using a Lisp-based expert system tool called GoldWorks II based on the Castlemaine Design System¹.

3.1 The architecture

The ATMB provides a frame-based knowledge representation scheme for building design knowledge bases and domain specific design knowledge sources. A blackboard control system [Hayes-Roth 1985] is used in the ATMB to control the design process in an incremental and opportunistic way. This blackboard control system is integrated with an Assumption-based Truth Maintenance System (ATMS) for maintaining the consistency of design knowledge and for managing design contexts when multiple design solutions are explored. A graphical explanation system is used to explain the design results maintained in the ATMS database.

A *design knowledge source* in the ATMB is an independent and self-contained program performing a design task. It is opportunistic and acts when its preconditions are matched by the data already on the blackboard. Design knowledge sources use a combination of algorithmic and heuristic techniques to perform inferences. Their work is controlled by a blackboard control system in a systematic way. A single production rule, a set of such rules, a procedure, or even a complicated domain specific software package can each be regarded as a design knowledge source.

A *user assumption* in the ATMB is a design decision made by a designer when exploring a design problem using the ATMB kernel. Such a decision might be to assume the value of a design variable, create a set of design object instances, or choose a particular design method etc. In the ATMB kernel, the blackboard control system acts on user assumptions. A user assumption represents a designer's intention to explore the design space. User assumptions are distinguished from other user commands, such as asking the system to display or explain its current status, because they do not necessarily trigger the system's inference engines.

A *design context* in the ATMB is a set of values for some design variables and parameters, which specify a design solution or a partial design solution. A design context involves, and is justified by, *user assumptions*, *design object instances* and *design knowledge sources* used to derive the design results. Such a design context is defined in the ATMS database as a set of nodes which are held in a given environment (a set of assumptions). If an environment contains any false node, the context (and that environment) is marked as inconsistent.

¹ The Castlemaine system was developed in Edinburgh in collaboration with Logica Cambridge Ltd., British Bio-technology Ltd. and CamAxys Ltd. for supporting small molecule drug design.

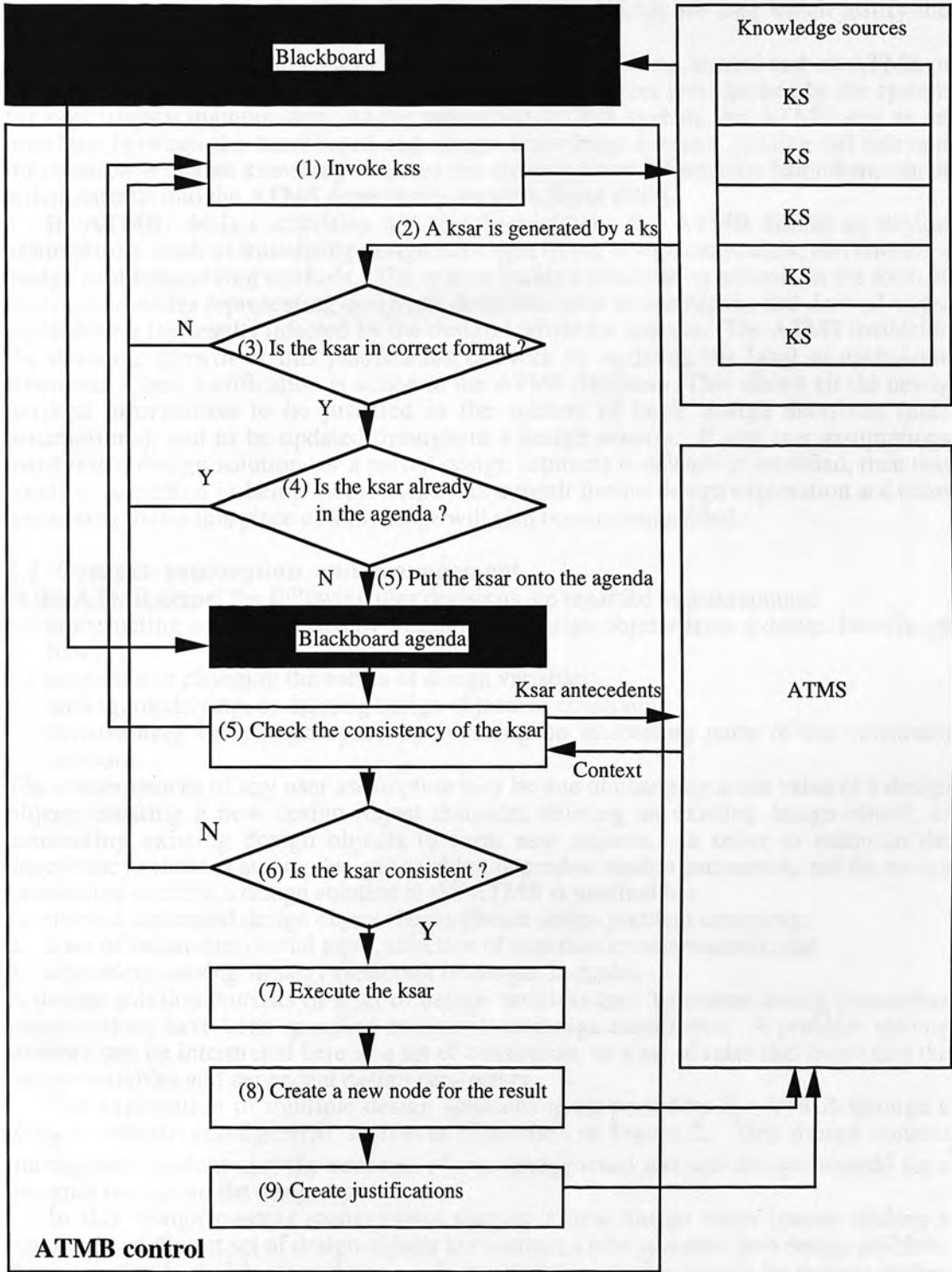


Figure 1. Control process in ATMB

A *knowledge source activation record* (KSAR) in the ATMB is an object representing a bid, i.e., a design knowledge source's intention to perform a design task. A KSAR contains explicit preconditions and the consequences of a proposed action. It carries the necessary bindings established through pattern matching and intended action function names. These bindings are used as antecedents to justify the results generated by the the

proposed action functions. Here the antecedents of a KSAR are data which justify the proposed actions of the KSAR.

Figure 1 illustrates the control process of an integrated blackboard and an ATMS in which KSARs are proposed by the design knowledge sources and checked by the system for consistency maintenance. In the integrated ATMB system, the ATMS acts as an interface between the blackboard and design knowledge sources, passing out relevant information to design knowledge sources and receiving new information from them which it then installs into the ATMS dependency network [Ross 1989].

In ATMB, design activities are transformed into the ATMB kernel as design assumptions, such as initialising design data, specifying design constraints, and choosing design problem solving methods. The system builds a justification network in the form of *assumption nodes* representing designer's decisions (user assumptions), and *derived nodes* representing the results inferred by the design knowledge sources. The ATMS maintains the dynamic growth of this justification network by updating the label of each node whenever a new justification is added to the ATMS database. This allows all the newly derived information to be justified in the context of basic design decisions (user assumptions), and to be updated throughout a design session. If any user assumptions justifying a design solution (or a partial design solution) is deleted or modified, then that solution is marked as being inconsistent. As a result further design exploration activities attempting to use this piece of knowledge will also become unjustified.

3.2 Context exploration and management

In the ATMB kernel the following user decisions are regarded as assumptions:

- constructing a design space by instantiating design objects from a design knowledge base;
- assuming or changing the values of design variables;
- adding, modifying, or deleting design objects or constraints;
- constraining the design space by focusing on interesting parts of the constraint network.

The consequences of any user assumption may be one of changing a slot value of a design object; creating a new design object instance; deleting an existing design object; or connecting existing design objects to form new objects. In order to maintain the dependency relations among design variables, dependent design parameters, and the design knowledge sources, a design solution in the ATMB is justified by:

1. a set of structured design object classes (initial design problem structure);
2. a set of initial data (initial input, selection of materials or components); and
3. a problem solving strategy (selection of design methods).

A design solution consists of a set of design variables and dependent design parameters whose values have been specified as a result of design exploration. A problem solving strategy can be interpreted here as a set of constraints, or a set of rules that constrains the design variables and dependent design parameters.

The exploration of multiple design solutions is supported by the ATMB through a *design context management system*, as illustrated in Figure 2. This design context management system uses the concepts of *new design route* and *new design branch*² for a designer to explore the design space.

In this design context management system, a new *design route* means loading a completely different set of design objects to construct a new structure for a design problem. A new *design branch* is created under a design route or a design branch for making further choices on either design variable values or design methods. Once the structure of a design problem is initialised by loading a number of existing design object classes and instances, this structure can be further explored in terms of object set, method set, constraint set, and evaluation criteria, by creating different and new design branches. When a new design branch is created under a design route or a design branch, it inherits some of the design decisions from its parent design routes or branches. Further exploration of the design

² The concepts of design route and design branch were exercised in the Castlemaine project for supporting drug design.

problem is then performed by making new design decisions. For example, in Figure 2 design branch 2 inherits the method set, constraint set and evaluation criteria of design route 1, but changes object set for further exploration. At the end of a design session, there will be more than one design route or branch, each of which represents a different design context justified by a set of user assumptions.

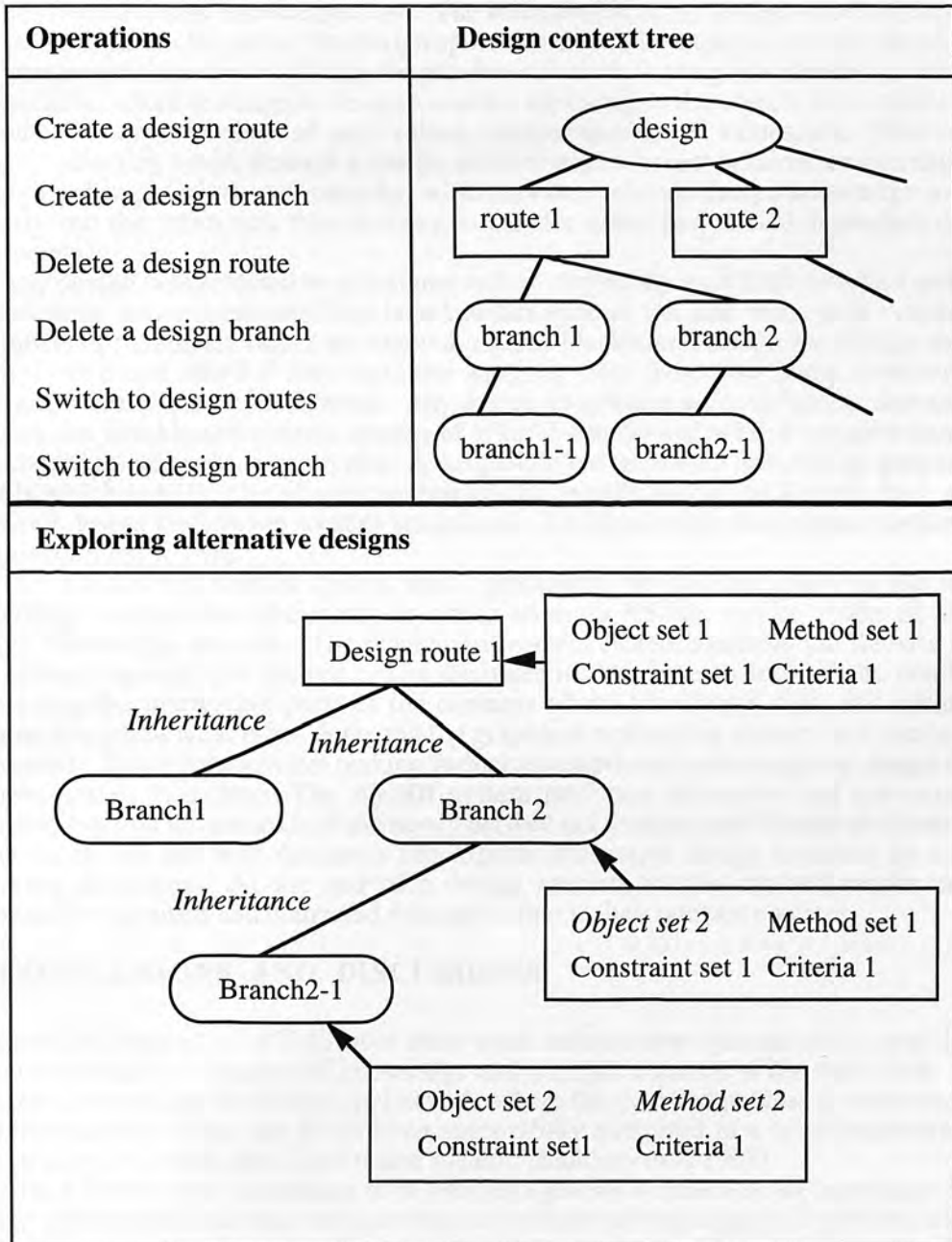


Figure 2. Exploring design solutions

This design context management system provides six domain independent design exploration operations. These operations include creating or deleting a new design route, creating or deleting a new design branch, and switching design contexts by moving to design routes or design branches for design exploration purposes. The design context management system is further supported by a graphical explanation system which displays derived design results and their contexts in both domain terms (ie, how design objects are

related to each other), or in ATMS terms (ie, how a derived node is justified by assumption nodes).

3.3 How is it intended to work ?

The ATMB as a design support tool works in the following way: the ATMB system is initialised by a designer by loading a number of design objects and design knowledge sources from a design knowledge base. The instantiation of the design objects effectively represents a particular part of the design space selected to be explored in more detail. The designer might then start exploring the possible solutions to this particular design problem by choosing values to assign to design variables appearing in the objects and constraints to see what the consequences of such values, or combinations of values, are. This can be done by selecting a task through a design requirement statement window, or alternatively, through a design task menu hierarchy, which invokes relevant design knowledge sources to carry out the inference, thus deriving values for other unspecified dependent design parameters.

Any design task selected by a designer is first verified by an ATMB interface and then passed to the truth maintained blackboard control system. As new items such as instances of objects or parameter values are entered into the blackboard, design knowledge sources are informed and asked if they can infer anything from these new items, together with anything already on the blackboard. Any design knowledge source which is able to infer notifies the blackboard control system of what it can do and what it requires from the blackboard to carry out its inference. A design knowledge source does this by proposing a KSAR which is to be placed onto the blackboard agenda, where the KSARs from all the involved design knowledge sources are queued. Justification for the proposed actions are contained in the KSAR.

The blackboard control system starts processing the KSARs when all the design knowledge sources become silent, ie, when no more KSARs can be produced by any design knowledge sources. The blackboard control system executes the KSARs in the blackboard agenda one by one. The designer is then able to look at the results by displaying the interesting parts of the contents of the blackboard data, and asking the system to explain what it has done, via the graphical explanation system, and decide what to do next. This might involve making further assumptions by creating new design routes or new design branches. The ATMB system performs inferences and maintains the justifications and the contexts of the newly derived information until further decisions need to be made. In this way designers can explore alternative design solutions by making different decisions. At the end of a design session, all the derived results can be graphically explained and compared with each other in their relevant contexts.

4. CONCLUSIONS AND DISCUSSIONS

The main advantage of ATMS over other truth maintenance systems is its capability of maintaining the consistency of knowledge and multiple contexts at the same time. Such systems are suitable for design applications where the decision process is nonmonotonic and incremental. This has so far been successfully exercised in a small-molecule drug design support system (the Castlemaine system) [Smithers et al 1993].

The ATMB kernel is intended to be used as a general architecture for knowledge-based design applications that need multiple context problem solving support. It performs the task of design context management by supporting the exploration of alternative designs based on different design routes or design branches. When applying the ATMB system to a new domain, a major problem is how to transfer domain dependent user actions into the internal, domain independent ATMB operations in order to get the maximal support from the system. In the Castlemaine drug design system an ATMB interface provides some support for doing this and has demonstrated to a sufficient extent how this might be achieved in the domain of small molecule drug design.

The ATMB kernel discussed in this paper defines design context based on the initial data used, design problem selected and design control or evaluation criteria. However, the issue of defining context and connecting the ATMS with a problem solver is domain

dependent. A suitable granularity needs to be decided, ie., (what should be regarded as ATMS nodes and assumptions) in order for the whole system to be computational efficient.

Beyond the domain of engineering design, ATMS-based context management systems can be used in decision making systems or conceptual knowledge acquisition systems where multiple solutions need to be explored simultaneously. In order to do so, a general strategy is needed for the user to define contexts. In such systems, the decision areas are to be formulated as a set of decision variables and constraints. A context can be created by the user by relating small sets of decision variables to different views of decision areas [Logan *et al* 1992]. In this way a large set of decision variables and constraints can be divided and managed by an ATMS-based context management system.

Work is being carried out at the Engineering Design Centre in Cambridge University to develop an integrated functional modelling system for mechanical engineering design. In this system the ATMB is to be used as the basic architecture for create simplified views of a potentially large set of design variables and constraints so that various functional reasoning methods and constraint-based optimisation methods such as simulated annealing and genetic algorithms can be more efficiently used in user-defined design issues or contexts.

ACKNOWLEDGEMENTS

An earlier version of the ATMB architecture was implemented as part of the Castlemaine system which involves Logica Cambridge Ltd, British Bio-technology Ltd. and CamAxys Ltd. The current implementation of the ATMB uses a C-based ATMS implemented by Peter Ross of Edinburgh University's Artificial Intelligence Department. My current research on AI-based design at the Engineering Design Centre in Cambridge University is being funded by EPSRC.

REFERENCES

- [Banares-Alcantara 1991] Banares-Alcantara, R., 1991, "Representing the Engineering Design Process: Two Hypotheses", First International Conference on Artificial Intelligence in Design, June, 1991, Edinburgh, Scotland.
- [Bowen et al 1992] J. Bowen, D. Bahler, 1992, "Supporting multiple perspectives: a constraint-based approach to concurrent engineering", *AI in Design' 92*.
- [de Kleer 1986] de Kleer, J., "An Assumption-based TMS", *Artificial Intelligence*, Vol. 28. 1986.
- [Doyle 1979] Doyle, J., 1979, "A Truth Maintenance System", *Artificial Intelligence*, 12(3):231-272, 1979.
- [Hayes-Roth 1985] Hayes-Roth, B., 1985, "Blackboard Architecture for Control", *Journal of Artificial Intelligence*, 26:251-231, 1985.
- [Jones 1992] Jones, J. C., 1992, "Design Methods", Second Edition, Van Nostrand Reinhold, New York.
- [Logan *et al* 1991] Logan, B., Millington, K., and Smithers, T., 1991, "Be Economical with the Truth, Assumption-based Context Management in the Edinburgh Designer System", *Artificial Intelligence in Design Conference*, June 1991, Edinburgh, Scotland.
- [Ross 1989] Ross, P., 1989, "A Simple ATMS", Department of Artificial Intelligence, University of Edinburgh, UK. June, 1989.
- [Smithers *et al* 1990] Smithers, T., Conkie, A., Doheny, J., Logan, B., Millington, K. and Tang, M., 1990, "Design as Intelligent Behaviour: An AI in Design Research Programme", *International Journal of Artificial Intelligence in Engineering*, 5.
- [Smithers *et al* 1993] Smithers, T., Tang, M. and Tomes, N., 1993, "Approach in Intelligent Drug Design", 26th Hawaii International Conference on System Science, January, 1993, Hawaii, USA.
- [Tangg 1995] Tang, M.X., 1995 "Development of an integrated AI system for conceptual design", 1995 Lancaster International Workshop on Engineering Design, Lake District, UK, March, 1995.