SRC TR 88-87

MARYLAND

TECHNICAL
RESEARCH
REPORT

Knowledge Representation For Expert Systems In Chemical Process Control Design

by

Gregory J. Birky

SYSTEMS RESEARCH CENTER

UNIVERSITY OF MARYLAND

COLLEGE PARK, MARYLAND 20742

ABSTRACT

Title of Dissertation :

KNOWLEDGE REPRESENTATION FOR EXPERT SYSTEMS IN CHEMICAL PROCESS CONTROL DESIGN

Gregory J. Birky, Doctor of Philosophy, 1988
Dissertation directed by : T.J. McAvoy
Professor
Chemical & Nuclear Engineering Dept.

The subject of this dissertation is knowledge representation for expert systems applied to chemical process control. For the purposes of the dissertation, control system design is defined as establishment of the necessary single input - single output controllers for regulatory and constrained control. One of the most difficult tasks in creating an expert system to solve any problem is organization of the knowledge such that creation, modification, and extension of the expert system can progress smoothly.

An introduction to expert systems and the way knowledge is represented within them is given. Included in the introduction is a description of the requirements of an expert system shell to be used to solve the control design problem. Idiomatic control and the goal tree success tree model for knowledge are presented as organizational tools to represent the problem solution prior to expert system construction.

A methodology for constructing the knowledge base of an expert system for control system design is proposed. That method is used to create the expert system DICODE (DIstillation COntrol Design Expert) using a commercially available expert system shell. The expert system DICODE is presented in detail, and generalizations are drawn concerning use of the method for creating expert systems for control system design. Customization of the shell in the form of a partial knowledge base specific to control system design but not specific to distillation is discussed.

Most of the control design problem solution is cast into an organized structure which facilitates creation of the expert system. The only exception to this is the specific knowledge used to determine variable pairings for regulatory control. The knowledge associated with this variable pairing is expert dependent, and therefore generalizations concerning structure are difficult to make.

A simulated neural net is applied to the regulatory control variable pairing problem in the hopes of providing a "learn by example" solution. A neural net has the ability to learn an input/output pattern mapping through repeated presentation of a learning set. The net used for regulatory control design is taught 100 pairs of input/output data vectors and tested on data not in the learning set. The net is found to adequately provide a regulatory control design for input data not in the learning set.

KNOWLEDGE REPRESENTATION FOR EXPERT SYSTEMS IN CHEMICAL PROCESS CONTROL DESIGN

by

Gregory J. Birky

Dissertation Submitted to the Faculty of the Graduate School
of The University of Maryland in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
1988

Advisory Committee :
Professor T. J. Mc Avoy
Associate Professor M. Modarres
Professor O. A. Asbjornsen
Assistant Professor S. Azarm
Dr. B. D. Tyreus

For Sevan,

whose love gives my life meaning.

ACKNOWLEDGMENTS

I am grateful to my advisor, Dr. Thomas J. McAvoy for his encouragment, support and guidance during the progress of my graduate research at the University of Maryland. I would also like to acknowledge Page S. Buckley formerly of E.I. du Pont de Nemours & Co. for his contribution to the cooperative project which constitutes much of the substance of my Ph.D. research. I also thank Bjorn D. Tyreus for his constant suggestions and hours of invaluable conversation. I would like to acknowledge the contribution of Dr. Mohammad Modarres for his contribution of the formulation of the goal tree success tree model as applied to the research.

I could not have completed my degree without the constant love, support, and encouragement of my wife Sevan. So often, she put up with the lunatic ravings of a discouraged graduate student. Her love and caring made my life possible during the bad times, and always gave it meaning. My parents deserve only the highest praise for their ever present support as well as for instilling within me the desire to achieve the highest of goals.

I must acknowledge the Systems Research Center, the Graduate School of the University of Maryland, and E.I. du Pont de Nemours & Co. for the financial support which made my Ph.D. studies possible. I thank the faculty and staff of the University of Maryland, and particularly the Department of Chemical and Nuclear Engineering.

I also thank my office mates for frequent discussions which always helped to keep my work in perspective.

TABLE OF CONTENTS

Page

LIST OF TABLES ix

LIST OF FIGURES x

CHAPTER 1: Introduction 1
1.1 Expert Systems for Chemical Process Control Design 1
1.1.1 Contributions of This Work 4
1.1.2 Requirements for Expert System Solution 6
1.1.3 Expert Systems Versus Algorithmic Programming 8
1.2 Expert System Shells and Knowledge Representation 11
1.2.1 Knowledge Representation 12
1.2.1.1 Production Rules 12
1.2.1.2 Semantinc Nets 14
1.2.1.3 Frames 15
1.2.1.4 Object Oriented Programming 17
1.2.2 Inferencing Strategy 18
1.2.2.1 Backward Chaining 18
1.2.2.2 Search Strategy 20
1.2.2.3 Forward Chaining 21
1.2.3 User Interface 22
1.3 Expert System Shell Requirements for Design 23
1.3.1 Chaining 24
1.3.2 Controlling Inferencing 25
1.3.3 Explanation Facilities 26
1.3.4 Access Oriented Programming 26
1.3.5 External Program Use 27

CHAPTER 2: Idiomatic Control 29
2.1 Idiomatic Contol System Analysis 30

2.1.1 Idiomatic Control System Decomposition 31
2.1.2 Idiom Definition 35
2.1.3 Example of Control System Analysis 36
2.1.3.1 Control System Decomposition 37
2.1.3.2 Idiom Definition 37
2.2 Idiomatic Control System Synthesis 40
2.2.1 Control Objective Definition 41
2.2.2 Translation of Objectives to Idioms 44
2.2.3 Example of Idiomatic Control System Synthesis 45
2.2.3.1 Control Objective Definition 45
2.2.3.2 Control Objective Translation 51
2.2.4 Regulatory Control Variable Pairing 52

CHAPTER 3: Goal Tree – Success Tree 58
3.1 Goal Tree Structure 59
3.2 Building a Goal Tree 60
3.3 Idiomatic Control Synthesis as a Goal Tree 62
3.3.1 The Root Goal 63
3.3.2 Subgoal 1: Determine the Best Control System 63
3.3.3 Subgoal 2: Constraint Operation 66
3.3.4 Subgoal 3: Supportive Control System 69
3.3.5 Subgoal 4: Control System Implementation 71

CHAPTER 4: DICODE: Distillation Control System Design 76
4.1 Project Definition 76
4.2 Goal Tree – Success Tree for DICODE 77
4.3 Choosing an Expert System Shell 83
4.3.1 Expert System Function 84
4.3.2 Frame Representation for Knowledge 84
4.3.3 External Call Capability 85
4.3.4 Flexibility 86
4.3.5 Cost 87
4.4 KES™ PS 87
4.4.1 Classes 87
4.4.2 Attributes 89
4.4.3 External Program Use 90
4.4.4 Rules 91

4.4.5 Demons 93
4.4.6 Actions 94
4.5 DICODE Implementation 95
4.5.1 Role of the GTST 95
4.5.2 Classes in DICODE 98
4.5.3 Global Attributes 98
4.5.4 Rules 99
4.5.5 DICODE Demons 104
4.5.6 DICODE Actions 108
4.5.7 DICODE Execution 110
4.6 Insights Gained Through DICODE 116

CHAPTER 5: Knowledge Base Structuring 119
5.1 Knowledge Organization in a GTST 120
5.2 Process Specific Knowledge Representation 122
5.2.1 Frames 123
5.2.2 Attributes 126
5.2.3 Rules 127
5.3 Knowledge Base Building Building Methodology 128
5.3.1 Building the GTST 128
5.3.2 Defining Goals and Success Paths as Frames 129
5.3.3 Rules 131
5.3.3.1 Type 1 Rules 131
5.3.3.2 Type 2 Rules 133
5.3.3.3 Type 3 Rules 134
5.3.3.4 Type 2a Rules 136
5.3.4 Demons 137
5.3.4.1 Type 1 Demons 138
5.3.4.2 Type 2 Demons 140
5.3.4.3 Type 3 Demons 141
5.3.4.4 Type 4 Demons 142
5.4 Shell Requirements for GTST Use 143

CHAPTER 6: Customized Shell for Control Design 146
6.1 Shell Requirements 148
6.1.1 Integration of Heuristics and Procedures 148
6.1.2 Frame Structure 149

6.1.3 Chaining 150
6.1.4 Access Oriented Programming 151
6.1.5 Justification of Results 152
6.2 Shell Extensions 153
6.2.1 Goals and Success Paths 154
6.2.1.1 Goals as Frames 154
6.2.1.2 Success Paths as Frames 155
6.2.2 Rules for GTST 156
6.2.3 Facets or Demons for GTST 160
6.2.4 External Program Use 163
6.3 Summary 169

CHAPTER 7: Neural Net for Control Design 172
7.1 Neural Networks 173
7.1.1 Learning in Neural Nets 178
7.1.2 Classic Problems 179
7.2 The Regulatory Control Design Neural Net 182
7.2.1 The Problem to Be Solved 182
7.2.2 The Network Used 184
7.2.3 Results 185
7.3 Summary of Results 195

CHAPTER 8: Conclusions 198
8.1 Distillation Control Design with an Expert System 199
8.2 Application of Neural Nets to Control Design 201
8.3 The Combinatorial Nature of Design 202
8.4 Recommendations for Future Work 203

APPENDIX A: Graphics, Objects, and Idioms 205
A.1 Graphics 205
A.2 Objects 208
A.3 Idioms 209

APPENDIX B: DICODE: A History 211
B.1 Preliminary Work 212
B.2 Industrial Applications 213
B.3 Project Definition 213

B.4 Interviews With the Expert 214
B.5 Extension of the Prototype 218
B.5.1 Controls for Entire Column 218
B.5.2 Other Measures 221
B.5.3 Differences in Philosophy 223
B.6 Present State of DICODE 226

APPENDIX C: Using DICODE 227
C.1 Overall Operation 227
C.2 Getting Started 230
C.3 Input Attributes and Their Use 231
C.4 The Menu System 232
C.5 Getting Results 233
C.6 External Program Use 235
C.7 DICODE in the Computer 238
C.8 Tuning DICODE 240

APPENDIX D: Customized Shell 249

BIBLIOGRAPHY 262

LIST OF TABLES

Number		Page
2.1	General Process Idioms.	41
2.2	Unit Process Idioms (Distillation).	42
2.3	Tabulation of Design Issues and Solutions: Primary Issues.	43
2.4	Tabulation of Design Issues and Solutions: Secondary Issues.	44
2.5	Material balance control variable pairings.	48
2.6	Primary Control Objectives.	49
2.7	Secondary control objectives.	51
2.8	Variables for idiomatic control.	52
2.9	Objective–Idiom Pairing.	53
7.1	Neural Net Input Interpretation.	186
7.2	Neural Net Output Interpretation.	187
7.3	Neural Net Topologies Used.	190
7.4	Material Balance Control Variable Pairings.	194

LIST OF FIGURES

Number		Page
1.1	Industrial control system design.	2
1.2	Example of a semantic net.	14
1.3	Example of a frame representation for the concept of car.	16
1.4	Backward chaining inferencing strategy.	19
1.5	Depth first search strategy.	20
1.6	Breadth first search strategy.	21
1.7	Forward chaining inference strategy.	22
2.1	Industrial control system for column 1.	32
2.2	Control system with secondary idioms removed.	33
2.3	Control system with secondary and constraint idioms removed.	34
2.4	Level 1 control flow diagram for reflux valve.	35
2.5	Level 0 control flow diagram for reflux valve.	36
2.6	Column 1 with no controls.	38
2.7	Level 1 control flow diagram for hot oil valve.	39
2.8	Level 0 control flow diagram for hot oil valve.	40
2.9	Column 2 with no controls.	46
2.10	Column 2 with primary regulatory controls.	54
2.11	Column 2 with primary regulatory and constraint controls.	55
2.12	Column 2 with primary and secondary controls.	56
3.1	Root goal and level 1 subgoals.	64
3.2	Goal tree for subgoal 1.	67
3.3	Goal tree for subgoal 2.	69
3.4	Goal tree for subgoal 3.	72
3.5	Goal tree for subgoal 4.	74
4.1	Goal tree for DICODE.	79
4.2	Success paths for pressure control goal.	80
4.3	Success paths for reflux drum level control.	80
4.4	Success paths for column base level control.	80
4.5	Success paths for distillate composition control.	81
4.6	Success paths for bottom composition control.	81
4.7	Success paths for condensate temperature control.	81
5.1	Example goal tree.	131
5.2	A lowest level subgoal and its success paths.	134
6.1	Functional representation of customization to an expert	

	system shell.	147
7.1	One processing node with weighted inputs and one output.	174
7.2	Output of processing node as a sigmoidal function activation.	175
7.3	Part of a three layer neural net.	177
7.4	Net to solve XOR with two hidden layer nodes.	180
7.5	Net to solve XOR with one hidden layer node.	181
7.6	Neural net results for 4 hidden nodes.	191
7.7	Neural net results for 5 hidden nodes.	192
7.8	Neural net results for 10 hidden nodes.	193

CHAPTER 1

INTRODUCTION

1.1. Expert Systems for Chemical Process Control Design

The design of control systems for new and existing chemical plants is an important problem today. As operating and capital costs rise, there is more incentive to design better control systems for chemical plants. Significant savings have been documented for many chemical processes with new, better control system designs[U.S. D.O.E., 1980]. The best control systems are designed by experts with extensive experience in process control. These few experts have a large body of knowledge which they use to design control systems, but which is not readily available to others.

A typical example of an industrial control system design is shown in Figure 1.1. Most engineers would look at this P/I diagram and wonder how the designer ever produced such a control system. Since many control systems are

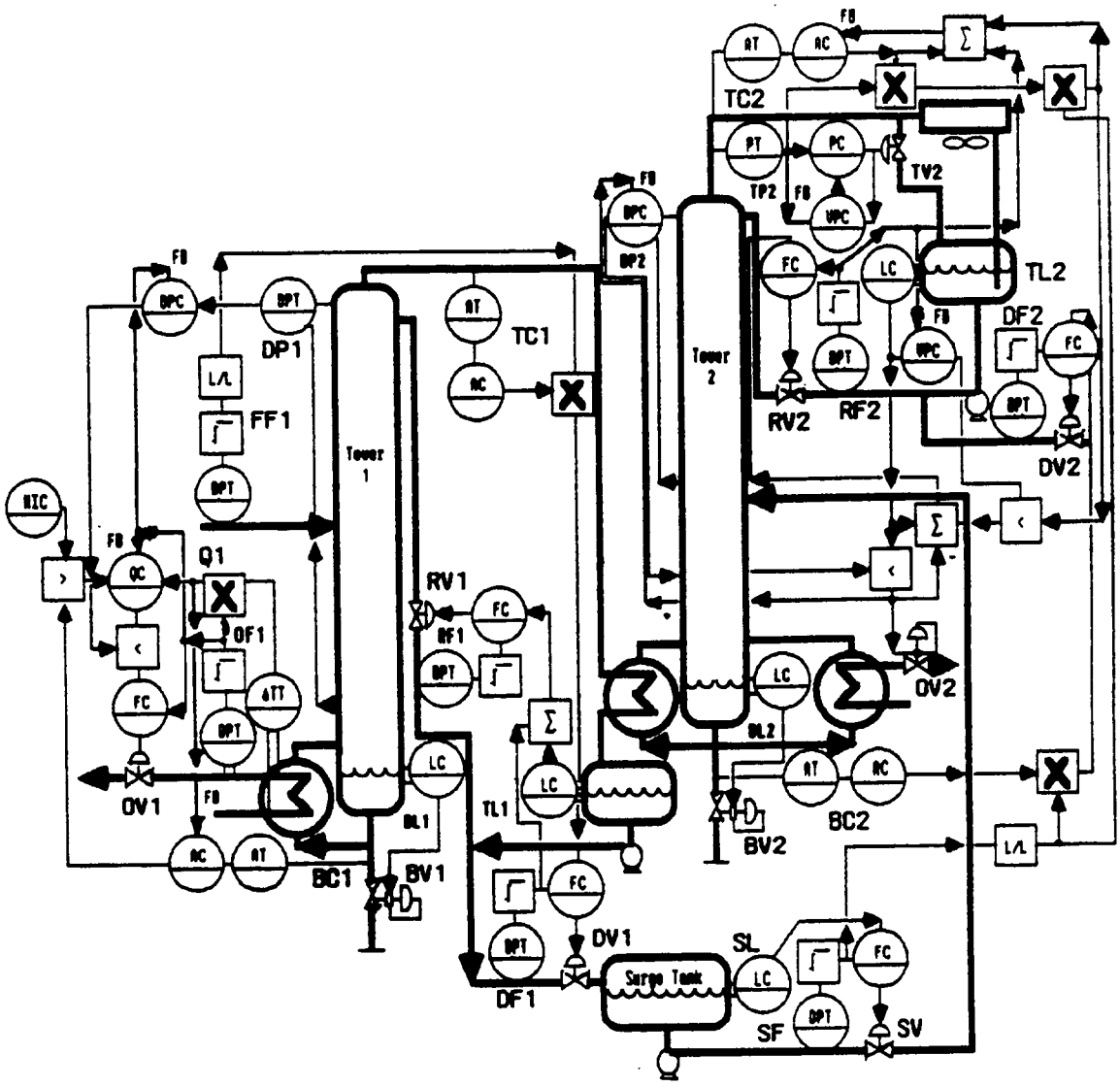


Figure 1.1. Industrial control system design.

designed by a control expert whose knowledge of chemical process control is very extensive, how can others gain insight to the control system design methodology? Furthermore, what happens to the knowledge contained in the expert's mind when he retires? How can the expert's knowledge be captured, and made readily available to others in the control design field?

An expert system containing the knowledge in process control design could conceivably be the answer to these questions. However, not all problems can be solved by an expert system. Also, how does one go about building such an expert system? The questions of expert system solution to the control design problem will be addressed in the body of this dissertation with particular emphasis on the structured nature of the knowledge. Since the use of an expert system shell for development of the expert system is desirable due to the amount of time which can be saved, a methodology has been developed which organizes the knowledge and leads directly to implementation of the knowledge in an expert system shell. Representation of the knowledge in an intermediate form is used as a part of the methodology for translating knowledge into the knowledge base. The methodology is independent of the shell used for expert system development, but does lead to evaluation of the demands on the shell.

The body of this dissertation will focus first on expert systems for problem solution and the various forms of knowledge representation available in expert system shells. Next, idiomatic control as developed by E. Bristol in the early 1980's is introduced in order to understand and synthesize control systems in

a structured way. The goal tree success tree as used in the nuclear power industry for fault diagnosis is introduced next as a model for deep knowledge. Both idiomatic control and the goal tree success tree are established formalisms but have never been combined. Section 3 of Chapter 3 describes the use of the goal tree to model idiomatic control synthesis for the purposes of expert system application to the control design problem. The use of the goal tree to model the design problem solution is new, and some slight differences exist between the formal goal tree success tree model and its use as described here.

1.1.1. Contributions of This Work

The contribution of the research described in this dissertation is to combine idiomatic control synthesis and the goal tree success tree model into a formal methodology for programming expert systems in the control synthesis domain. The methodology is validated by creation of the expert system DICODE for distillation control design. Structure has been given to the knowledge necessary to implement expert systems for control design, except for a small portion of expert dependent knowledge concerning regulatory variable pairing. The methodology, specific principles, and guidelines for programming expert systems in the control design domain lead to demands on the expert system shell. These requirements are discussed in detail as as to how they should be used to implement expert systems for control system synthesis.

Generalization of the principles used to create DICODE leads to customization of an expert system shell. The customization is in the form of a partial knowledge base which provides the facilities needed to create an expert system for control system synthesis easily and quickly. The customization is described, and the partial knowledge base is developed using the shell in which DICODE was created.

A continuing difficulty in creating expert systems is obtaining the knowledge from an expert. In the control synthesis problem, the inability to give a general structure to this expert specific knowledge as mentioned above, results in part from the difficulty in obtaining the knowledge. An alternative to acquiring the knowledge for regulatory variable pairing from the expert is proposed in this thesis. A neural net whose purpose is to avoid obtaining the knowledge from the expert by learning existing designs is investigated. The net is found to have the ability to learn many examples, and performs adequately on examples which are not part of the learning set.

It should be noted that combining existing tools to solve the control design problem is a systems approach. Furthermore, the hierarchical breakdown described by the goal tree success tree model is an organization or structuring of the problem similar to top down problem solving, also an integral part of systems engineering [Eisner, 1987, and Asbjornsen, 1987]. Therefore, the overall approach to the problem solution can be described as an example of systems engineering.

1.1.2. Requirements for Expert System Solution

To address the question of feasibility of an expert system for design of control systems, some background information is useful. In order for a problem to be solvable by an expert system, it must satisfy some requirements. The development of an expert system should be considered only if development is possible, justified, and appropriate[Waterman, 1986].

For a problem to be solved by an expert system,

1. the problem must not require common sense,
2. the problem must require only cognitive skills,
3. experts in the domain must be able to articulate their solution methods,
4. genuine experts must exist,
5. experts must agree on solutions,
6. the problem must not be too difficult, and
7. the problem must not be poorly understood.

Note that all of these conditions must be satisfied in order for an expert system solution to be possible. In the area of control system design, the problem solutions do not require common sense, but only cognitive skills. This can be illustrated by the fact that control system design can be taught to individuals, whereas common sense cannot. Although some parts of the design procedure may seem to come from common sense, there is a specific reason for each decision and each step of the design. A large portion of the design may be the result of

experience, but this too can be taught as any skill can be taught. Experts surely exist and are able to articulate their solution methods as demonstrated by the publication of several excellent reference books. The problem is well understood by the experts and not too difficult for them to solve. The only requirement which may not be fulfilled is that experts agree on the solution. Although not all experts would design the same control system for a given process, one expert would generally agree that the solution given by another expert is valid.

The justifications for expert system solution to a problem are;

1. the problem solution has a high payoff,
2. human expertise is being lost,
3. human expertise is scarce,
4. expertise is needed in many locations, or
5. expertise is needed in a hostile environment.

Note that only one of these five justifications needs to be fulfilled. In the control system design field, the solution to the problem definitely has a high payoff. As mentioned earlier, significant savings can result from improved control. Human expertise is scarce, and is being lost as well. The experts with the most experience retire at some point, and their knowledge is no longer available. Many of the experts act as consultants, which makes their experience available in many locations, but at high cost if the expert must travel to the site of the equipment. Therefore, the first four of the five justifications are met in the control system design area.

The problem characteristics that make the use of expert systems appropriate are;

1. the problem requires symbol manipulation,
2. the problem requires heuristic solutions,
3. the problem is not too easy,
4. the problem solution has practical value, and
5. the problem solution is of manageable size.

Note that all of these characteristics must be true of the problem for an expert system solution to be appropriate. For the design of control systems, symbol manipulation is necessary. The design procedure includes many aspects which are not calculational in nature, but which require the use of concepts and relationships between objects. These aspects are a form of symbol manipulation. The design of a control system is also heuristic in nature. There are many decisions which must be made based on experience or on "rules of thumb" which the designer uses. These decisions are heuristic in nature rather than algorithmic. The problem must not be too easy; in general, if the problem takes the expert four to eight hours to solve, then the complexity is appropriate for solution by an expert system[Waterman, 1986].

1.1.3. Expert Systems Versus Algorithmic Programming

Although the problem of control system design satisfies the requirements

for an expert system solution, the question remains as to whether it is more appropriate to solve the problem using an algorithmic program. In the past, most design packages have been algorithmic in nature. Many have been written in FORTRAN and therefore it is clear that the problem can be solved with an algorithmic program. There are however, some advantages to using an expert system for design as for any application. The first is the difference in program construction and execution. One of the salient features of an expert system is the separation of program or executable portion from the knowledge. This separation allows the author of the expert system to concentrate on the knowledge to be captured without having to worry about the details of execution. However, some effort must be spent to give priority to knowledge which can cause conflicts in the knowledge base. Another advantage is that the knowledge is more easily understood when it is in the form of an expert system. The rules are to some extent self documenting. This does not mean that documentation within the knowledge base is unnecessary, however the amount of documentation needed is decreased if the author makes an effort to provide descriptive attribute (and object) names.

A clear advantage of using an algorithmic solution to the design problem is that most engineers have experience in algorithmic programming languages. The time which the author must spend learning the syntax of an expert system shell can be spent on the actual solution to the problem. Another advantage of using an algorithmic language for the solution is the increased execution

speed and availability of the hardware and software. FORTRAN is the most common programming language in the engineering field and is readily available for almost any machine. Expert system shells are not as readily available and sometimes require AI computers to run. The expert system shells are often written in Lisp and therefore require a large amount of memory, and execution is slow. Fortunately, the expert system shells are increasingly being written in other languages and are therefore becoming available for more machines.

There are advantages and disadvantages to both solutions to the design problem. Successful design packages of both types exist. The design problem has been addressed by AI research and solution by expert systems is possible and often desirable. A good example is the expert system XCON which configures VAX computer systems based on purchase information[Waterman, 1986]. XCON was written in OPS5 (a forward chaining shell) and is very successful. The decision of whether to build an expert system or a FORTRAN program may rest heavily on the experience of the author and his preference for one or the other. One final note about expert system solutions for a given problem domain is that expert systems can handle problems where the solution space is combinatoric much more easily than algorithmic programs. Since the design problem is often combinatoric in nature, an expert system may provide a better solution.

Assuming that an expert system solution to the problem is chosen, the next step is to determine the requirements of the expert system. The scope

of the problem must be clearly defined and the solution fully understood in a systematic way. Once the problem is well structured and well understood, then an appropriate form of knowledge representation must be selected. It is important to stress the necessity of organizing or structuring the knowledge. Unstructured knowledge is complex and difficult to understand. Since the expert system could be used for education by students as well as process and control design engineers and even process operators, it is imperative that the knowledge representation be as clear as possible and that the expert system be able to justify the solution with an explanation of its decisions.

1.2. Expert System Shells and Knowledge Representation

Once it has been determined that an expert system should be used for the solution to the problem, the selection of an appropriate expert system shell or expert system building tool is the next step. Before listing the requirements of a shell for the design problem, some background about knowledge representation techniques, inferencing strategies, and other facilities is useful. The next sections will discuss the various knowledge representation techniques, including production rules, semantic nets, frames, access oriented programming, objects, and object oriented programming. The following sections will discuss inferencing strategies, user interfaces, and external program use.

1.2.1. Knowledge Representation

The form in which knowledge is represented in an expert system may not have a profound effect on the execution of the expert system, and may not effect the user's opinion concerning the expert system. However, the form of knowledge representation can have an effect on the ease of creating and maintaining the knowledge base. The form of knowledge representation within the expert system can provide convenient methods of using external programs, representing objects, presenting a user interface, and decrease the total number of rules. The following discussion is an attempt to introduce the most common forms of knowledge representation in a simplified manner, yet describe the basic features, advantages and disadvantages of each.

1.2.1.1. Production Rules

The most primitive and oldest form of knowledge representation for expert systems is production rules. Production rules represent the most basic form of heuristic knowledge, the *if - then* relationship. Production rules consist of two parts: the antecedent, where preconditions are represented, and the consequent, where the results are represented. If the conditions of the antecedent are met, then the rule *fires* – the actions of the consequent are taken. Typically, the antecedent checks the value of an attribute or variable, and the consequent assigns a value to another attribute. The antecedent can consist of multiple con-

ditions in conjunctions or disjunctions, and the consequent can contain multiple actions. The following is an example of a typical rule.

> If a tire is flat and a spare is available,
> then appropriate action is replace flat with spare.

In many shells which use production rules, the use of certainty factors is provided. Certainty factors are useful when many sources of knowledge exist for a single result. For example, consider the knowledge contained in an expert system for diagnosing a medical condition. Typically, a doctor examines a patient, takes samples for tests and then bases his diagnosis on his observations of the patient and on the test results. An expert system must combine multiple test results with a list of symptoms to infer a diagnosis. Certain tests may indicate more than one condition to differing degrees. Combination of the test results will provide more than one diagnosis, but evidence should build toward one as more likely than the others. Certainty factors can be used effectively for this purpose. One test result in the antecedent of a rule will provide a diagnosis of several conditions, each with a different certainty. Another test result will provide other diagnoses with differing certainty. In all probability, as more test results are used, and symptoms are provided, evidence will build toward one diagnosis with a higher degree of certainty than the others.

Productions rules are simple and easily understood, provided that the rule syntax used by the expert system shell is understandable and care is taken on the part of the knowledge base author to use descriptive attribute names and values. Unfortunately, most knowledge bases for expert systems contain hundreds if not

thousands of rules. Taken one at a time, they are understandable, but together, the overall effect of the rules is often difficult to comprehend. Some other methods of knowledge representation exist which can simplify and condense the knowledge.

1.2.1.2. Semantic Nets

A semantic net is a group of points called nodes, connected by links called arcs which define the relationship between nodes. Nodes are often objects and the arcs are relationships like *is-a*, *has-a*, etc. Thus, semantic nets establish an inheritance and structure hierarchy which can save on knowledge base size since it is not necessary to explicitly state all of the properties of a node down in the net. An example is shown in Figure 1.2.

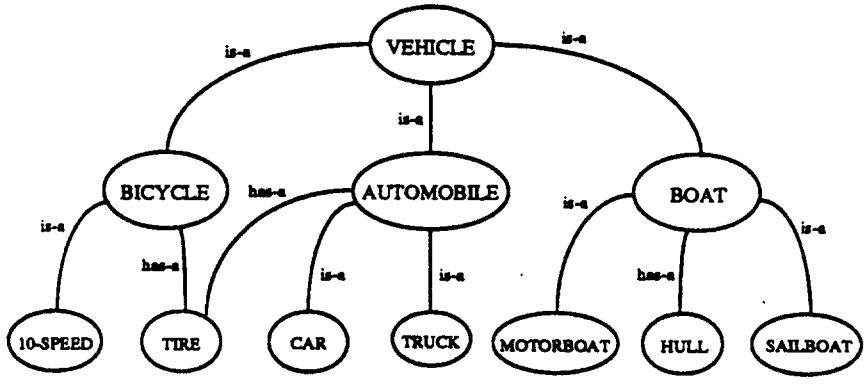


Figure 1.2. Example of a semantic net.

In this example, although no *has-a* arc links hull to sailboat, it is clear that a sailboat has a hull since a sailboat *is-a* boat, a boat *has-a* hull and sailboat

inherits this relationship from boat.

1.2.1.3. Frames

One of the most popular forms of knowledge representation other than production rules is the frame. A frame is actually a node in a semantic net. In its most basic form, a frame is an entity with a name and several slots. The slots can be attributes which are associated with the frame, or they can contain other information pertaining to the frame. Along with the frames, rules must be present which operate on the attributes. Typically, frames are referred to as objects in many expert system shells when in fact they are simply static data structures. Usually, frames do represent objects in expert systems, with the attributes describing the object to some extent. A typical example is a car represented as a frame. Associated attributes could be color, age, make, and model. Knowledge representation using frames and associated attributes is commonly referred to as object-attribute-value (O-A-V) triples or triplets.

Frame based knowledge representation also provides inheritance features. Inheritance is a facility which is useful for many applications, particularly where many very similar concepts must be considered in the expert system. For instance, if many cars must be represented, then a parent frame *car* may be defined with the various important slots. Then, specific instances of *car* may be created, all of which will automatically have the slots defined for *car*. Slots can also represent more complex entities such as procedures to assign subgoals to be

pursued or procedures to execute external programs. Furthermore, some shells provide rules as slots in the frame. Unfortunately, a tremendous amount of confusion exists concerning frames, objects, frame based shells, access oriented programming, and object oriented programming. Figure 1.3 shows a frame with its parent frame, demonstrating the concept of slots and inheritance.

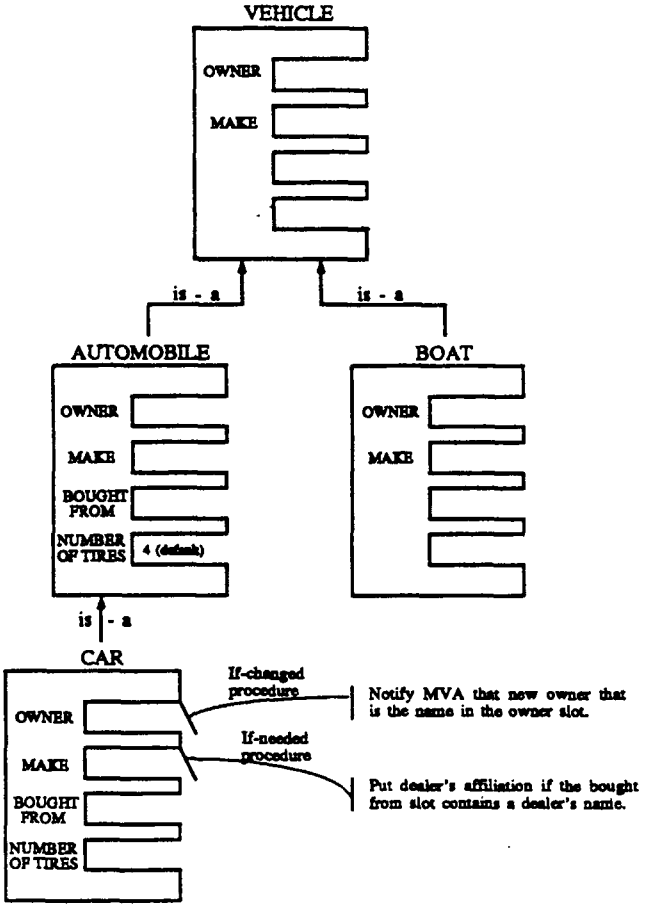


Figure 1.3. Example of a frame representation for the concept of car.

The frame representing a car is not an object in the object oriented pro-

gramming sense, it is simply a frame. If, within the frame, slots contain procedures which can be executed when information is changed or read, (so called demons) then the system provides access oriented programming. Object oriented programming is discussed in the following section.

1.2.1.4. Object Oriented Programming

In object oriented programming, objects are completely distinct entities which can contain a large amount of knowledge including rules and procedures. The difference between object oriented programming and access oriented programming is that in object oriented programming, the objects can only *talk* to each other. That is, they communicate by posting information in the *mail box* of another object. All procedures, rules, attributes, etc. are contained within the object and pertain only to that object. Object oriented expert system shells also provide for inheritance. SMALLTALK™ is an example of a truly object oriented expert system shell.

For most applications, access oriented programming is the most flexible and appropriate. Organization of the knowledge as frames provides some degree of clarity within the expert system. The use of inheritance can effectively reduce the number of rules necessary in the expert system by allowing one rule to apply to all instances of the parent frame.

1.2.2. Inferencing Strategy

Another consideration when choosing an expert system shell is the inferencing mechanism. Since the inference engine is the executable part of the expert system, it is separate from the knowledge base and usually inaccessible for modification.

1.2.2.1. Backward Chaining

Most expert system shells provide backward chaining. That is, the user specifies a goal to be attained (usually an attribute whose value is to be obtained). The inference engine then looks for all rules whose consequents will determine the value of the attribute. When a rule is encountered whose consequent assigns a value for the attribute, the antecedent is examined. If the antecedent is successful, that is, the conditions are met, then the original goal is attained and a value is obtained for the attribute which was specified. If however, not enough information is available to evaluate the antecedent of the rule, then subgoals for obtaining the values of the attributes in the antecedent become active, while the original goal is put on hold. The inference engine then attempts to attain the subgoals in the same way it attempted to attain the original goal. This 'chaining' procedure is known as backward chaining, and is often referred to as goal driven. When the lowest subgoals are finally satisfied (usually after asking the user for the necessary information), the associated

rules 'fire' giving values to the subgoal attributes, and thus result in obtaining a value for the attribute specified by the user. Figure 1.4 shows the backward chaining inference strategy.

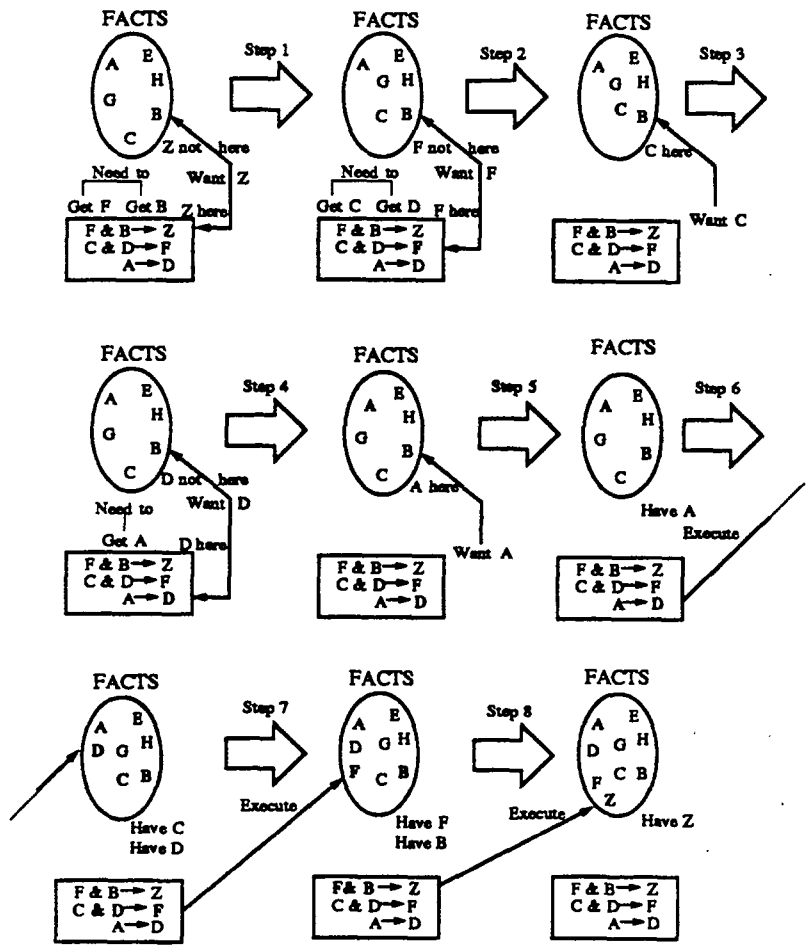


Figure 1.4. Backward chaining inferencing strategy.

1.2.2.2. Search Strategy

Variations of the basic backward chaining mechanism exist. Two variations are depth first and breadth first searches. In the depth first search, subgoals are created and pursued immediately. The depth first search results in considering the first rule which could supply a value to the attribute, creating subgoals, and sub-subgoals, until the rule succeeds or fails. If it succeeds, then inferencing is discontinued. If it fails, then the next rule encountered which could supply a value to the specified attribute is similarly considered. Figure 1.5 shows the depth first search order, if the ovals are considered rules.

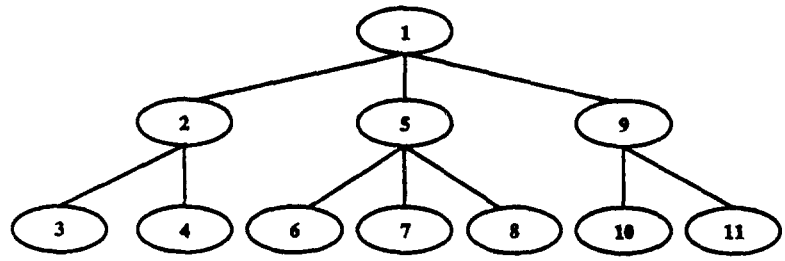


Figure 1.5. Depth first search strategy.

In the breadth first search, every rule which could supply a value to the attribute is considered before any subgoals are created. Thus, if one of the rules succeeds without further backward chaining, then a great deal of inferencing could be saved. If the antecedent of none of these first level rules is satisfied, then subgoals are created. Each of these subgoals is pursued in the same manner, by checking rules whose consequents could supply a value for the attributes of the subgoal. Figure 1.6 shows the order of the breadth first search.

Combinations and variations of depth first and breadth first searches also

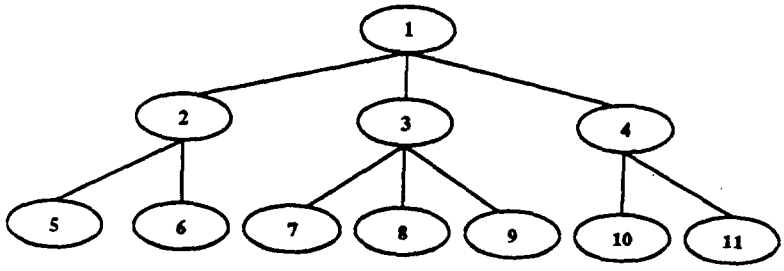


Figure 1.6. Breadth first search strategy.

exist. They may search depth first until it seems that the inference path may not succeed, and then pursue another path. Many search strategies exist, but since they are not necessarily important in choosing an expert system shell, and can be rather complicated, they will not be discussed further. The interested reader is referred to [Charniak and McDermott, 1985].

1.2.2.3. Forward Chaining

Some expert system shells provide forward chaining inference engines. Forward chaining is often referred to as data driven. In forward chaining, all input data is used to find all rules which could succeed. Then, a conflict resolution strategy is used to determine which rule should be 'fired'. The rule (only one) fires, adding to or changing the data. After the rule 'fires', the inference engine starts the search for rules again, with the new set of data. This cycle continues until no rules can fire with the current data. The result is a new data set hopefully containing the problem solution. Figure 1.7 shows the forward chaining inference strategy.

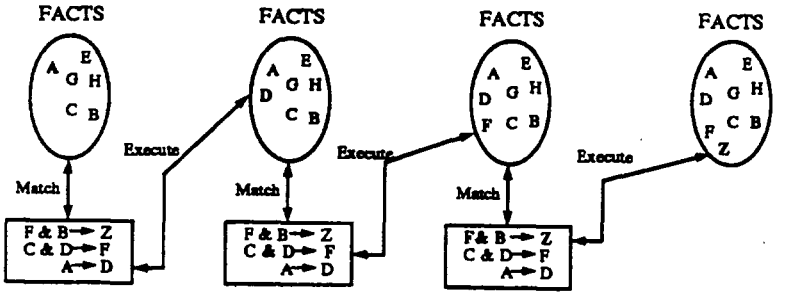


Figure 1.7. Forward chaining inference strategy.

1.2.3. User Interface

Another important aspect of an expert system shell is the end user interface. Most shells allow custom design of the user interface. Some use the question/answer mode while others use menu and mouse driven interfaces. One very desirable feature is explanation facilities. When the user is asked to provide information, he should be able to ask for a more detailed explanation of the question, advice as to what response is most appropriate, and why that information is necessary. Some shells provide for this type of interrogation, but others do not and therefore require extra effort to provide such facilities. When an answer is reached, the system should be able to provide information as to why that particular answer was reached. Some systems even allow interrogation into why another answer was not reached. Success of the expert system depends directly on the confidence of the user in the system's solution to the problem. Explanation facilities are invaluable in gaining the confidence of the user.

Some expert systems shells use windows for graphics and text display during expert system building and execution. This type of user interface has some

advantages and some disadvantages. The advantages are that shells with graphics tend to more user friendly than those which do not use graphics. Furthermore, if graphics will be a requirement of the expert system, then it is unnecessary to provide separate graphics capabilities. The disadvantages include the fact that these shells often are not as flexible as those which use a knowledge base closer to a programming environment or language. Another disadvantage is that using the graphics which are provided may limit the graphical possibilities for the expert system, while use of external graphics can provide a much more desirable graphics result.

1.3. Expert System Shell Requirements for Design

The task required of an expert system is that it mimic the solution of the expert. In the case of design, specific procedures are often followed. In this sense, one might wonder whether using an expert system is appropriate for the design problem. Numerous successful computer aided design packages which use algorithmic programming demonstrate the applicability of algorithmic programming to design. For a further discussion of this aspect, refer to Chapter 1, section 1.1. The utility of the expert system solution is for design problems in which the solution space is combinatoric, as can often happen. The design problem also contains both shallow knowledge, and deep knowledge. Shallow knowledge is knowledge which is the result of years of experience in the prob-

lem domain, and is heuristic in nature. Deep knowledge is knowledge which is gained through an understanding of the underlying phenomenon which creates a cause and effect relationship.

In design, deep knowledge includes calculational procedures and quantitative theoretic knowledge. Shallow knowledge is the knowledge which is used to make decisions which lead to the final answer, based on evidence provided by the deep knowledge as well as directly from user provided information. Here then is where the use of external routines must be possible in conjunction with the expert system.

1.3.1. Chaining

The design problem imposes some specific requirements upon the expert system shell. In the past, it has been accepted that the design problem typically requires a forward chaining inference engine. In forward chaining, a set of data or facts is used as input, and all possible resulting facts or data is inferred based on rules or procedures provided in the knowledge base. The reason that design has been classified as a problem for solution by forward chaining is that the solution or answer is actually an accumulation of facts, not a single answer. In backward chaining, a single answer or fact is sought as the result of a consultation. It is not necessary to solve the design problem with forward chaining, as the example described in this dissertation proves. The required inferencing

mechanism is influenced as much by the organization of the knowledge as by the problem domain. Further discussion of the knowledge organization is addressed later. One advantage of backward chaining is that during execution, the expert system asks only for necessary information. In a forward chaining shell, special effort must be made in order to avoid asking for unnecessary information.

1.3.2. Controlling Inferencing

Perhaps the most obvious requirement of the expert system shell stems from the fact that most design problems may be solved using a design procedure in which goals and objectives have definite priority and order. The design should proceed step by step along a rather well defined path toward the answer. In apparent conflict with the procedure based design methodology is the solution method provided by an expert system. Earlier it was stated that one of the salient features of an expert system is the separation of the executable part and the knowledge. If this separation is indeed complete, then how can an expert system be built which proceeds along a predefined path? The answer is that it need not proceed along a completely predefined path, but rather ask for information in an order which seems logical to the user, and allow the user to direct it through various stages of the design.

Most expert system shells allow giving priorities to rules in order to effectively take some control of the inferencing process. Unfortunately, giving

priorities to rules may not be adequate to effectively guide execution through the procedure properly. Assigning priorities to conflicting goals must be possible in order to reduce improper inferencing. Some shells allow procedural programming to create and satisfy goals in the proper order as well as allow the user to guide execution according to his preference. We have found the provision of this procedural programming to be instrumental in building a successful expert system for design.

1.3.3. Explanation Facilities

Explanation facilities are absolutely necessary in an expert system for design. Although the knowledge base author may understand the importance and significance of a particular piece of information, the end user may not. Explanation of the question and its relevance to the problem at hand (as well as logical ordering of questions to the user) creates an environment in which the user gains confidence in the expert system. Explanation of the final result is also important in order to lend credibility to the answer.

1.3.4. Access Oriented Programming

Another capability which we have found useful is access oriented programming facilities. In access oriented programming, facets are provided on the slots

of frames. The facets can be described by such titles as *when changed* or *when needed.* The facets are procedures or rules which are executed when certain conditions are met. Facets are often referred to as demons or guards. Demons are like rules, containing a rule like antecedent, but are passive, monitoring the values of the attributes in its antecedent. When values are changed and the antecedent is true, the demon fires just like a rule. Demons are in some sense forward chaining rules, although the consequent often contains a procedure. Demons are particularly useful in providing warnings and recommendations, including trapping inappropriate or inconsistent information provided by the user.

1.3.5. External Program Use

During the design procedure, it is often desirable if not necessary to run programs which already exist as executables in order to obtain more information about the particular problem. Expert system shells must provide for the use of executable programs which already exist as sources of information for the design. Typical examples include simulation programs for dynamic processes, stress analysis for load bearing structures, and data analysis programs. The expert system shell should allow specification of external programs as sources of knowledge or sources of attribute values. Then, when the value of an attribute whose value may be determined by an external source is needed, the external

program will automatically be executed. The expert system shell should allow the user to specify if he would like to use the external or an alternative such as provide the needed attribute value himself.

Another typical requirement of expert systems for design is a graphical representation of the result. Some shells allow definition of graphic objects as part of the shell and therefore can be used for graphic display of the design result. These shells often use windows and graphics for knowledge base building as well as end user interface, and are typically expensive. An alternative is to use externals for graphic display of the design result by passing necessary information to an executable program which uses a graphical kernel system (GKS).

CHAPTER 2

IDIOMATIC CONTROL

In the end of section 1.1, it is stated that it is important that the problem to be solved by an expert system be well structured and the solution fully understood. Organization of the knowledge is critical. Idiomatic control is a structured method for analysis and synthesis of control systems. It provides a starting point for organization of the problem solving knowledge in the control design domain. As such, it can provide the backbone for construction of an organized knowledge base for control system design. The idiomatic control design method has been represented in a form (goal tree - success tree) which facilitates construction of the knowledge base for an expert system in control system design. The goal tree - success tree (GTST) model for knowledge is discussed in detail in Chapter 3.

Idiomatic control, introduced in 1980 by E. Bristol[Bristol, 1980]was the first attempt at structured analysis and synthesis of control systems. In idiomatic control analysis, the "layers" of a P/I diagram are "peeled" away in a manner which reduces a very complex diagram such as that in Figure 1 to an

understandable form. During the analysis procedure, idioms are defined which describe the mini-inventions that the control expert uses to construct a control system. Idiomatic control synthesis matches the control objectives for the process under consideration with idioms defined during previous analysis of several control systems[Prassinos, 1982, Prassinos, et. al., 1984]. The idiomatic control synthesis methodology provides a structure for knowledge in control system design. It also seemed to match a method for representing knowledge in a model (the goal tree - success tree) which has been used to represent deep knowledge for use in expert systems.

2.1. Idiomatic Control System Analysis

The concept of control idioms was introduced as a way of defining mini-inventions which control design engineers use when designing a control system for a process. Idioms can be specialized such as a specific control loop which connects a manipulated variable and a controlled variable, or they can be general, such as a flow control loop used as an inner loop for a cascade arrangement. The idea of idiomatic control analysis and synthesis was borrowed from structured analysis where a complex problem is decomposed into smaller, more understandable sub-problems. Control idioms can be identified by applying the idiomatic control analysis methodology to various processes, and compiling a table of the resulting idioms with their purposes. Many control idioms are system

independent, however some are unique to specific processes. These idioms are defined in such a way that a control system may be designed by defining control objectives, and then building the control system by combining the idioms which satisfy the control objectives.

Idiomatic control analysis consists of two parts. The first part is the process of "peeling" away the layers of the control structure in order to gain insight into the purpose of the control system. The second part is a more detailed analysis which defines the idioms relating specifically to each actuator (valve). The results of the second part are the level 1 and level 0 control flow diagrams describing the relationships between measurements and actuators. Another result of the second part of idiomatic control analysis is a list of idioms and their purposes. By combining the idioms from analysis of several control systems in a table with their purposes and diagrams, a source of general and process specific idioms is obtained which can be used for idiomatic control synthesis.

2.1.1. Idiomatic Control System Decomposition

For the first part of idiomatic control analysis, the secondary or helpful idioms are removed as the first step in simplifying the complex P/I diagram. Figure 2.1 is a typical example of a complex P/I diagram. These secondary control idioms are those which are not absolutely necessary for control, but help to reduce the transient effects of disturbances and set point changes to the

system. Secondary control idioms include inner loops of cascade arrangements (which include BTU controllers), and feed forward. Figure 2.2 shows the system of Figure 2.1 after the secondary idioms are removed.

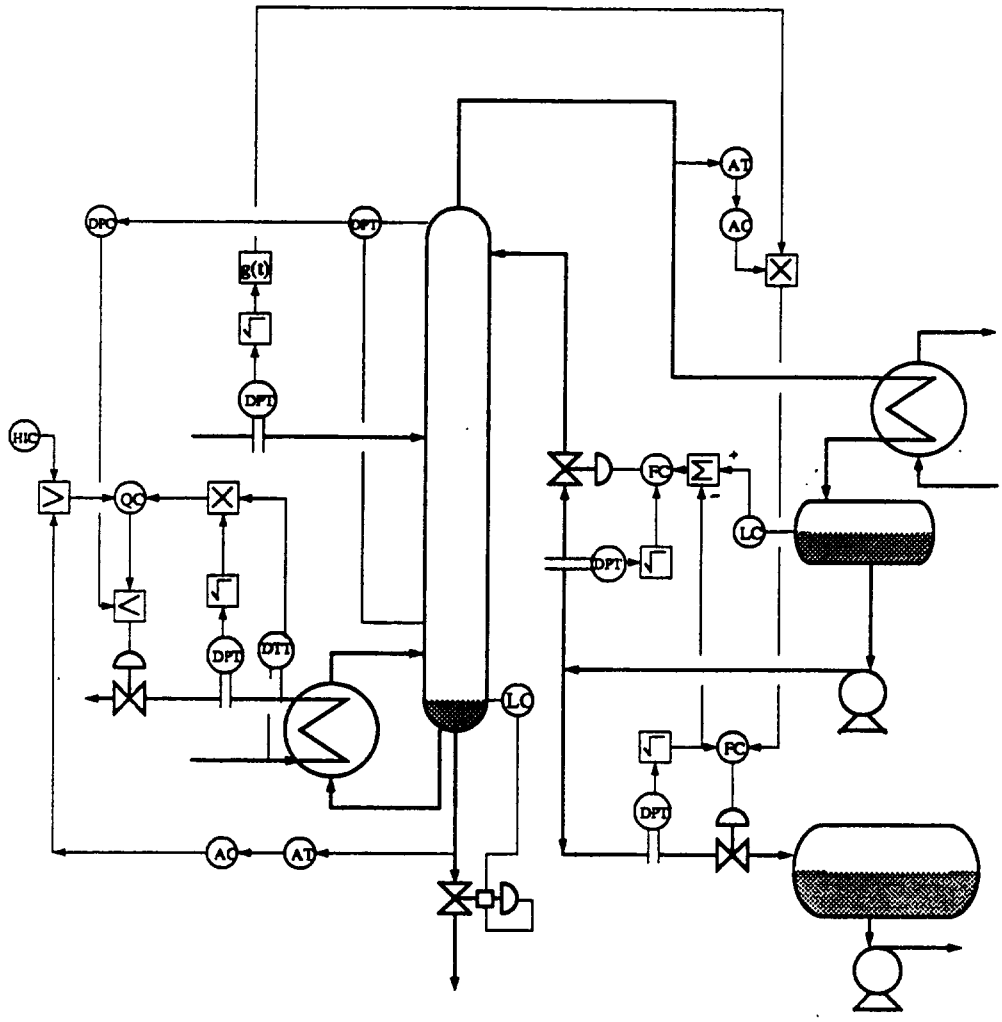


Figure 2.1. Industrial control system for column 1.

The remaining control structure consists of the primary control idioms responsible for normal operation, constraint operation, and optimizing control. The next step is to remove the idioms associated with constraint operation.

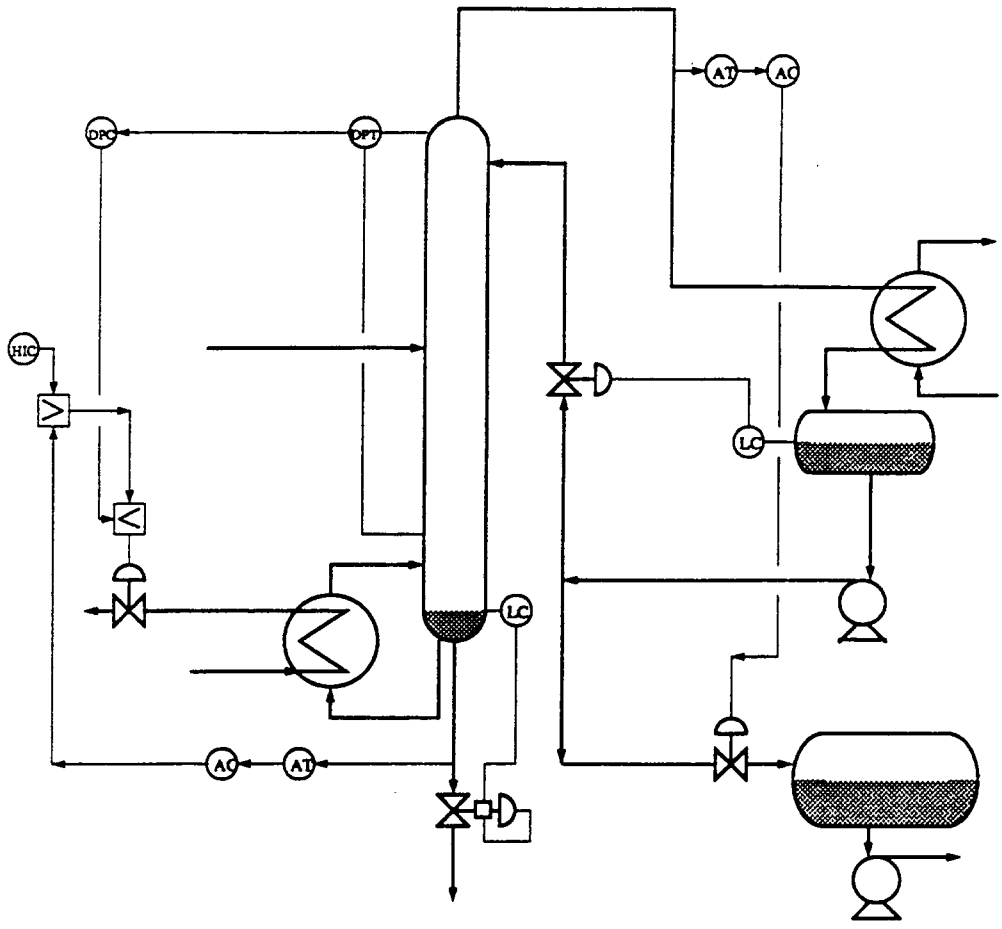


Figure 2.2. Control system with secondary idioms removed.

The constraint operation control idioms are those which take over control of the process when it approaches the limits of normal operation. Constraint control idioms are often process specific such as controlling the vapor rate in a distillation column between the flooding and weeping constraints.

Once the constraint operation control idioms have been removed, the remaining control structure consists of the primary control idioms for basic reg-

ulation and optimizing control. Figure 2.3 shows the control system of Figure 2.1 with the secondary and constraint idioms removed. The P/I diagram has been reduced to a form from which the original purpose of the control system is more easily understood.

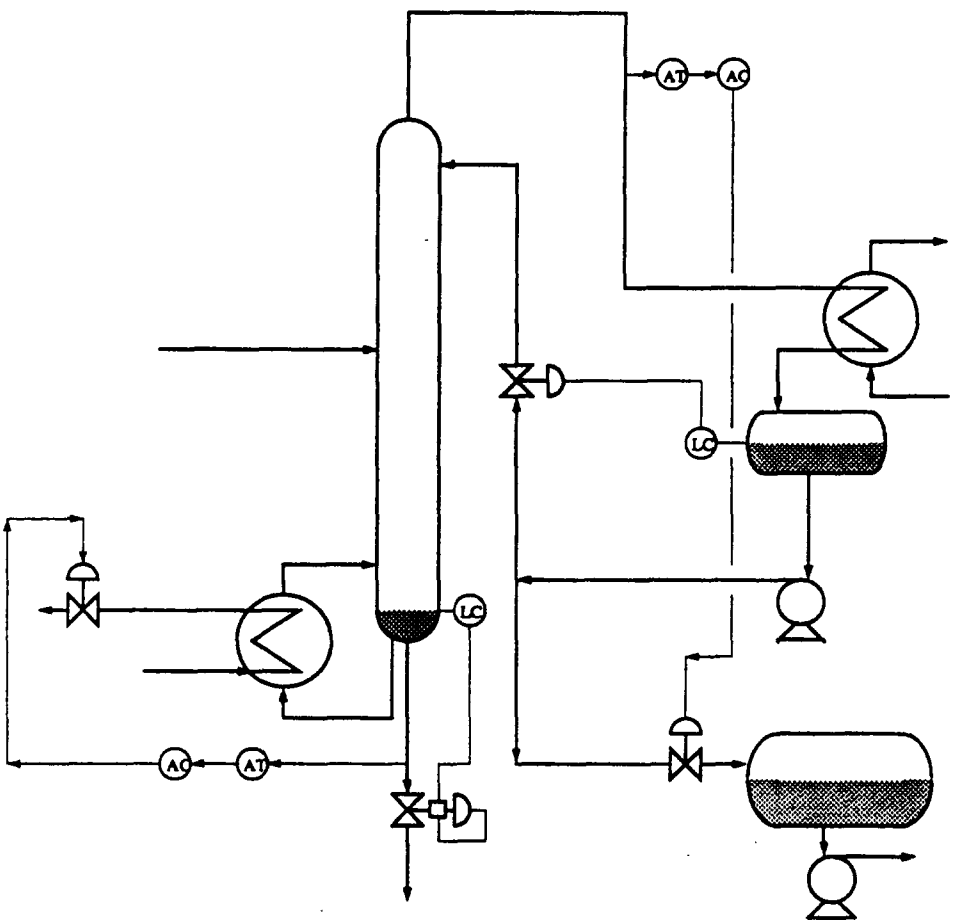


Figure 2.3. Control system with secondary and constraint idioms removed.

2.1.2. Idiom Definition

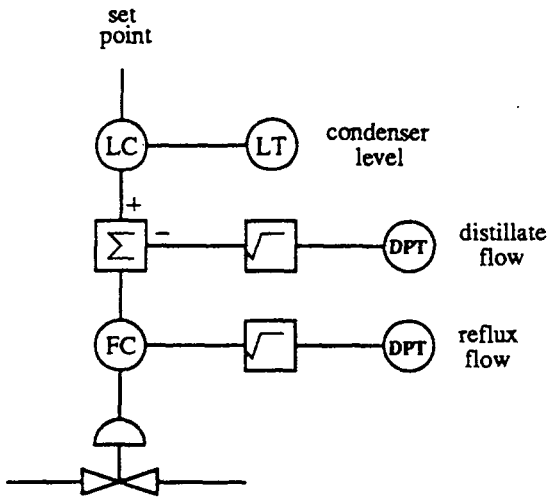


Figure 2.4. Level 1 control flow diagram for reflux valve.

The second part of the idiomatic control analysis methodology begins by defining the idioms which relate specifically to each actuator. The control signal is traced backward from the actuator (valve) to the measurement(s) and set points, including all intermediate selectors, multipliers, dividers, and summers. The resulting diagram is called a level 1 control flow diagram and shows the measurements on the right side with the signals flowing toward the main control flow path or trunk and joining it at collecting nodes. Figure 2.4 shows a level 1 control flow diagram for the example in the next section. The main control flow path is shown as the vertical path on the left. The horizontal and vertical paths consist of circles to represent sensors and controllers, and squares which represent algebraic operations, combined with lines representing the flow path for the signals. The horizontal paths join the vertical trunk at collecting nodes represented by smaller circles. Each horizontal path, which represents the flow

of a control signal to the trunk, is defined as a control idiom.

A convenient representation of an idiom is a notation similar to a general mathematical function. We use the I(A,B) notation where A and B are the elements of the horizontal control path in the level 1 control flow diagram including summing elements, multipliers, comparators, and other algebraic ma nipulations[Prassinos, et. al., 1984]. The level 0 control flow diagram is simply the level 1 control flow diagram with the idioms represented in the I(A,B) manner inside blocks which are connected vertically. Figures 2.4 and 2.5 show the level 1 and level 0 control flow diagrams respectively for reflux flow in a distillation column.

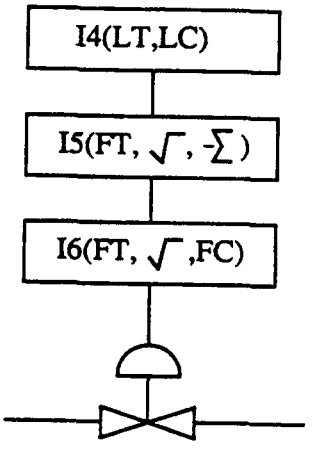


Figure 2.5. Level 0 control flow diagram for reflux valve.

2.1.3. Example of Control System Analysis

As an example of idiomatic control system analysis, consider the first dis-

tillation column shown in Figure 1.1. The same column is shown in Figure 2.1 with the second column removed except for the heat exchanger which integrates the columns.

2.1.3.1. Control System Decomposition

The control system decomposition for the first column in Figure 1.1 has been well described in section 2.1.1. The result of this analysis is the basic regulatory control system shown in Figure 2.3. From the diagram in Figure 2.3, it is much easier to determine the purpose of the basic regulatory control system. It is fairly clear that the regulatory control system consists of controlling bottom composition with heat input (or vapor boilup), controlling top composition with distillate flow (this pairing is the conventional energy balance control scheme), controlling the bottom level with bottom flow, and controlling the accumulator level with the reflux flow. Figure 2.6 shows the distillation column with the measurements and manipulators (valves) only. Figure 2.6 is the starting point for control system design.

2.1.3.2. Idiom Definition

To generate the level 1 control flow diagrams, the control signals are traced from measurements to manipulators. Starting with the hot oil valve in Figure 2.1, the first element is a low selector which selects between the signals from

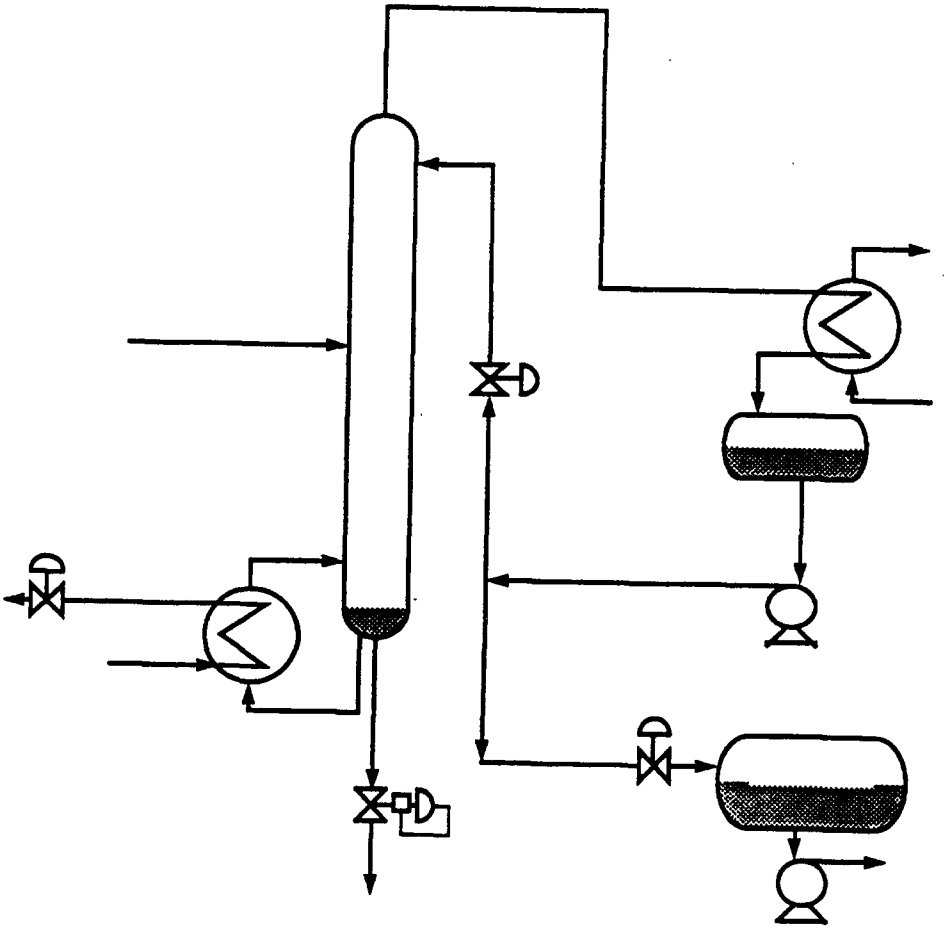


Figure 2.6. Column 1 with no controls.

the BTU controller and the differential pressure controller (DPC). The DPC is traced back to the differential pressure transmitter (DPT). The BTU controller is traced back to the multiplier of a differential temperature transmitter (ΔTT) and a flow measurement ($\sqrt{DPT}$). The setpoint of the BTU controller is traced back to a selector between the signal from the analyzer controller (AC) and the heat input controller (HIC). The signal to the AC is traced back to the analyzer

transmitter (AT), and the HIC setpoint is given as a constant. The resulting level 1 control flow diagram is shown in Figure 2.7.

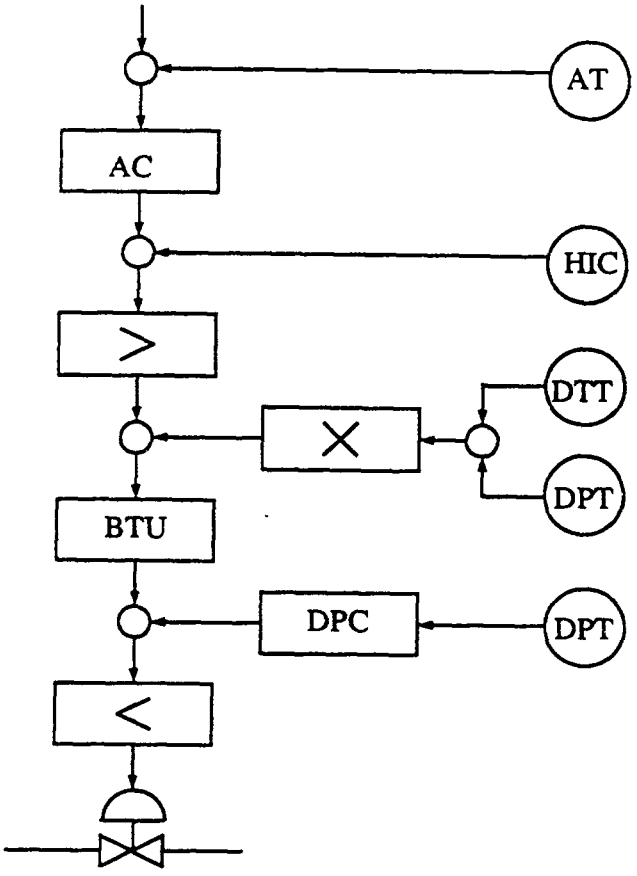


Figure 2.7. Level 1 control flow diagram for hot oil valve.

Each horizontal path of the level 1 control flow diagram is given by an idiom and represented in the level 0 control flow diagram shown in Figure 2.8.

The control flow diagrams for the other three valves may be developed in a similar manner. The development of these diagrams is not presented here since each is less complicated than the diagrams for the hot oil valve given in Figures 2.7 and 2.8 and explained in the paragraph above. The results of analysis

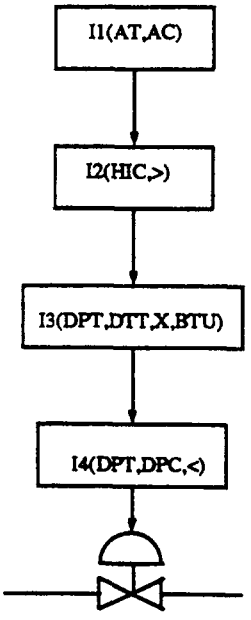


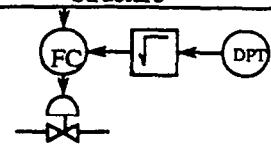
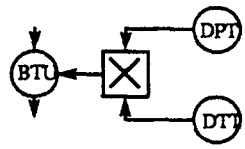
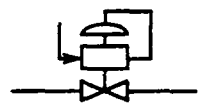
Figure 2.8. Level 0 control flow diagram for hot oil valve.

of several control systems leads to Tables 2.1 and 2.2. Table 2.1 shows some general process idioms with their purposes, objective representation, functional representation, and structure as it would appear in a P/I diagram. These tables are extremely useful during control system design by idiomatic synthesis.

2.2. Idiomatic Control System Synthesis

Idiomatic control system synthesis consists of two parts. The first is to define the control objectives for the process under consideration. The control objectives must be expressed in an unambiguous manner and includes regulatory, constraint, and supportive controls. The second part is translation of

Table 2.1: General Process Idioms

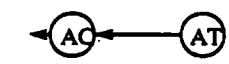
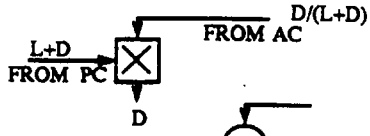
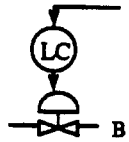
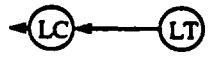
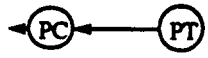
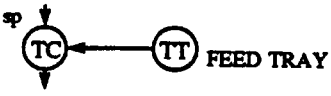
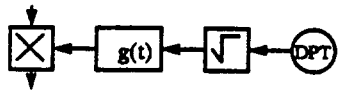
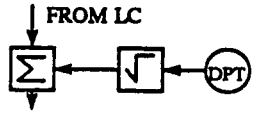
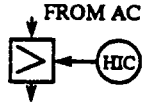
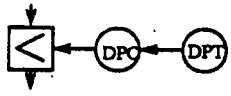
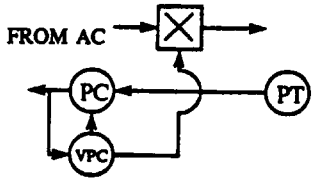
Idiom	Purpose	Variables	Components	Structure
1.	Stabilize flow	S(F,V)	I(DPT,⌐ ,FC)	
2.	Stabilize heat input	S(FDT,V)	I(DPT,DTT,X,BTU)	
3.	Eliminate stability problems because of the hysterisis of a valve.		Valve position controller	

control objectives into control idioms. A previously compiled table of control idioms and their associated purposes is extremely valuable for the second part of the synthesis procedure.

2.2.1. Control Objective Definition

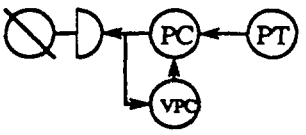
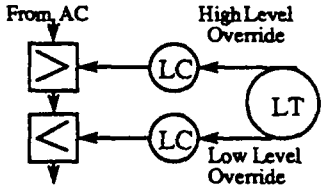
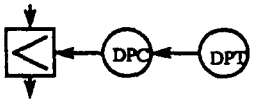
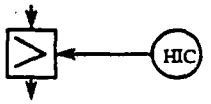
To use idiomatic control to build a control system, the control objectives must be defined completely. First, the basic regulatory control system is specified, that is, pair manipulated and controlled variables for regulation during normal operation and during constrained operation. The specification of the basic control system usually involves some optimization of the process such as

Table 2.2: Unit Process Idioms (Distillation)

Idiom	Purpose	Variables	Components	Structure
4.	Regulate composition	R(C,F)	I(AT,AC)	
5.	Regulate top composition using (L/D)	R(TC,DF)	I(X)	
6.	Regulate bottom level	R(BL,BV)	I(LC)	
7.	Regulate level of reflux drums	R(L,V)	I(LT,LC)	
8.	Regulate pressure at point in tower	R(P,OV)	I(PT,PC)	
9.	Regulate top pressure	R(P,RF or RF and DF)	I(PT,PC)	
10.	Regulate reflux flow	R(RF,RV)	I(Σ)	
11.	Regulate temperature near the feed	R(T,FV)	I(TT,TC)	
12.	Anticipate effect of feed flow on composition	A(FF,C)	I(DPT,√, g(t),X)	
13.	Anticipate effect of distillate flow on distillate composition	A(DF,DC)	I(DPT,√, Σ)	
14.	Constraint to achieve a minimum heat input to the tower	C^{LO}(FDT)	I(HIC,>)	
15.	Constrain differential pressure to avoid flooding	C^{HI}(DP,V)	I(DPT,DPC,<)	
16.	Constrain and stabilize pressure with a bypass valve to achieve minimum energy consumption	CS(P,BP)	I(PT,PC,VPC,X)	

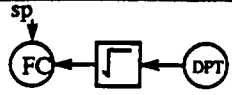
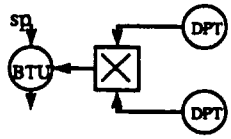
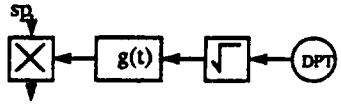
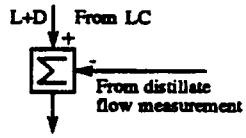
dual composition control, floating pressure control, or the use of valve positioners. Next, the secondary objectives of stabilization and anticipation (i.e. cascades and feed forwards) are specified. Table 2.3 shows the primary control objectives for a distillation column, and Table 2.4 shows the secondary control objectives for the same column.

Table 2.3: Tabulation of Design Issues and Solutions
Primary Issues (Goals)

Issue	Type	Solution	Idiom
1. Minimize Tower Pressure to Minimize Energy Consumption	Regulation	Use of valve position controller (VPC) on signal to pressure control valve.	
2. Control Composition at Both Ends of Tower to Minimize Energy Consumption	Regulation	Use two standard analyser controllers	
3. Level Control Must Never Be Lost	Constraint	A level controller must always be operational. Override composition controllers.	
4. The Tower Must Not Flood	Constraint	Use a differential pressure controller (DPC) to measure the approach to flooding. The DPC takes over heat input if tower starts to flood.	
5. The Tower Must Not Weep	Constraint	A minimum heat input must be supplied to the reboiler.	

These objectives can be expressed in an algebraic manner similar to the expressions used for the idioms. The form of the expressions is given by $X_i(C, M)$

Table 2.4: Tabulation of Design Issues and Solutions
Secondary Issues (Goals)

Issue	Type	Solution	Idiom
1. Correct For Flow Upsets Quickly	Cascade	Cascade other slow controllers to flow loop.	
2. Correct For Heat Input Disturbances Quickly	Cascade	Cascade other controllers to BTU loops.	
3. Adjust Tower Operation for Feed Flow Upsets	Feedforward	Ratio composition controllers to feed flow.	
4. Avoid Lag in Reflux Accumulator When Manipulating Distillate Flow For Composition Control	Feedforward	Use sum of reflux and distillate flows for accumulator level control.	

where X can be C for "constrain", R for "regulate", CS for "constrain and regulate", S for "stabilize", A for "anticipate", or D for "decouple"[Prassinos, et. al., 1984]. In most cases, (when $X = C, R, CS, or\ S$) C represents the controlled variable, and M represents the manipulated variable. For $X = A$, the relationship can be expressed as "Anticipate the effect of C on M." For $X = D$, the relationship can be expressed as "Decouple C and M."

2.2.2. Translation of Objectives to Idioms

After the objectives have been specified, they are translated into idioms

with a one-to-one pairing of objectives and idioms. The idioms can be taken from tables generated during the analysis of several control systems as described earlier. The purpose of the idioms match the objectives defined during the first two parts (definition of primary and secondary objectives) of the synthesis procedure.

2.2.3. Example of Idiomatic Control System Synthesis

The second distillation column of Figure 1.1 is taken as an example for application of the Idiomatic Control Synthesis methodology. The same column is shown in Figure 2.9 with no controls.

This column is used to separate a mixture of hydrocarbons and has the following characteristics.

Design Information:

Reflux Ratio (L/D) = 16

Distillate to Feed Ratio (D/F) = 0.22

Vapor Boilup to Feed Ratio (V/F) = 4

Purpose – hydrocarbon splitting

Cooling System – condenser with vapor bypass

Heat Source – hot oil

2.2.3.1. Control Objective Definition

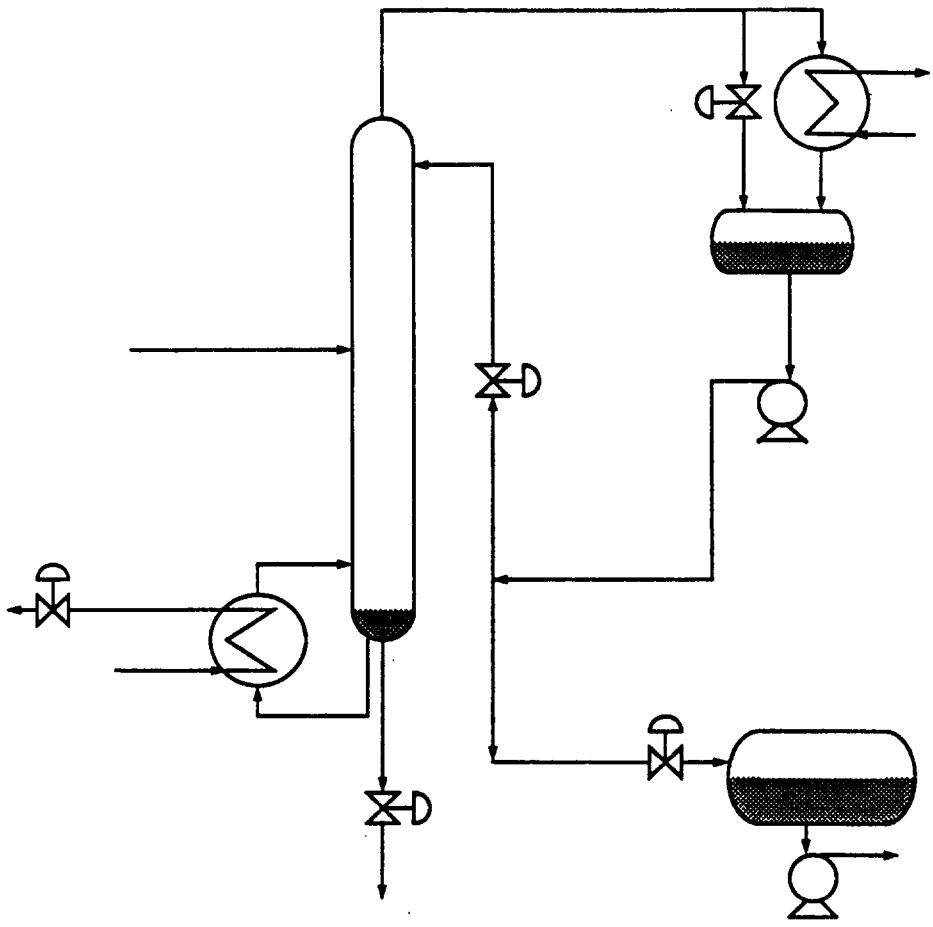


Figure 2.9. Column 2 with no controls.

The first step in idiomatic control system design is to define the control objectives. It is desired to control the compositions of both the distillate and bottoms streams. In addition, the pressure, reflux drum level, and column base level must be controlled. Composition, pressure and material balance controls are all primary regulatory control objectives. Primary constraint objectives are to maintain column differential pressure below the flooding limit, and to avoid weeping in the column due to insufficient vapor flow up the column.

The choice of manipulated variables for composition and material balance control variables is not addressed by the idiomatic control design methodology. The variable pairing must be accomplished using some control effectiveness measure such as the relative gain (RGA), relative disturbance gain (RDG) or perhaps some heuristics. For the distillation column discussed, the large reflux ratio (16) indicates that reflux flow is better suited to level control of the reflux drum than the distillate flow due to the relative sizes of the streams. (It is better to use the larger stream for level control unless conditions in the column base indicate that vapor boilup must be used for level control of the column base.) The vapor boilup to bottoms flow ratio is relatively small (V/B=5) and therefore, the bottoms flow may be used for level control of the column base. Table 2.5 summarizes the variable pairings for material balance control for varying values of reflux ratio and vapor boilup to bottoms flow ratio. For the current example, the table indicates that the pairing LB for material balance control is appropriate.

The choice of LB for material balance control leaves DV as the choice for composition control. It is worthwhile to note that the RGA for this particular pairing is 0.22, indicating that significant interaction will occur between the composition control loops. In this case, it is desirable to consider some alternatives. If one product of the column is less important than the other, it may be possible to implement only single ended composition control on the more important product. If dual composition control is desired, the control

Table 2.5 Material balance control variable pairings.

V/B \ L/D	<5	5 ≤L/D<10	≥10
<10	DB	LB	LB
10≤V/B<20	DV	DV \ LB	LB
≥20	DV	DV	DV^{t} \ LB^{tt}

The diagonals with two entries indicate that a comparison of L/D and (V/B)/2 must be made. The pairing on the left is recommened when (V/B)/2 > L/D, and the pairing on the right is recommended for the opposite case.

[t]Overrides must be provided for accumulator level.
[tt]Overrides must be provided for column base level.

loop for the composition of the less important product should be detuned, or a slow acting optimizing controller should be substituted. If tight control of both compositions is desired, some form of decoupling should be implemented.

The column shown in Figure 2.9 has a condenser which is provided with a vapor bypass, intended for pressure control. This means of controlling pressure can cause severe dynamic problems, particularly if inerts are present in the system. However, the equipment design obviously intends manipulation of the

vapor bypass flow for pressure control.

The second section of primary control objectives is concerned with constraint operation. Since we have chosen appropriate material balance control loop pairings, level overrides are not necessary, and constraint controls are limited to avoiding flooding and weeping in the column. The pressure drop across the column can be used as a measure of approach to flooding. Column differential pressure may be constrained below the flooding point by limiting the heat input below the amount which could cause flooding. Weeping may similarly be avoided by constraining heat input and thereby vapor boilup above the minimum necessary to ensure adequate vapor flow up the column. The primary control objectives are summarized in Table 2.6.

Table 2.6 Primary control objtectives.

Purpose	Type	Manipulated Variable
1. Control Distillate Composition	Regulatory	Distillate Flow
2. Control Bottoms Composition	Regulatory	Vapor Boilup
3. Control Reflux Drum Level	Regulatory	Reflux Flow
4. Control Column Base Level	Regulatory	Bottom Flow
5. Control Column Pressure	Regulatory	Vapor Bypass Flow
6. Avoid Flooding	Constraint	Override Heat Input
7. Avoid Weeping	Constraint	Override Heat Input

The secondary or supportive control objectives must also be defined in a similar manner. One of the most common supportive control loops is the flow controller used as the inner loop of a cascade arrangement. The flow controller can compensate quickly for flow changes due to pressure fluctuations and therefore stabilize overall control. Flow controllers should be provided on all the liquid flow valves: distillate, reflux, and bottoms flow valves. A similar controller should be provided on the heat input. If steam or other condensing medium is used, then a flow controller should be provided. In this example, hot oil is used for the heating medium. Since the purpose of the regulatory control loop for bottoms composition is to manipulate heat input and thereby vapor boilup, a BTU or heat input controller should be used. The BTU controller measures hot oil flow and temperature drop across the reboiler. The product of these two measurements is proportional to the heat absorbed in the reboiler. The BTU controller can quickly compensate for changes in heat input due to fluctuations is hot oil pressure and temperature.

Also included in the secondary or supportive control objectives are the controls which anticipate effects on control. Changes in feed flow rate have a profound effect on the distillate and bottoms compositions. By setting the manipulated variables for composition control in ratio to feed flow, the effects of feed flow variations can be largely eliminated. Another improvement can be made by anticipating the effect of variations in distillate flow for composition control on the reflux accumulator level. By using the level controller output as

the sum of reflux and distillate flows, this effect can also be minimized. The secondary control objectives are summarized in Table 2.7.

Table 2.7 Secondary control objectives.

Purpose	Type	Manipulated Variable
8. Stabilize Distillate Flow	Cascade	Distillate Valve
9. Stabilize Reflux Flow	Cascade	Reflux Valve
10. Stabilize Bottom Flow	Cascade	Bottoms Valve
11. Stabilize Heat Input	Cascade	Hot Oil Valve
12. Anticipate Effects of Feed Flow Changes on Distillate Composition	Feed Forward	Distillate Flow
13. Anticpate Effects of Feed Flow Changes on Bottom Composition	Feed Forward	Heat Input
14. Anticipate Effects of Distillate Flow Changes on Reflux Drum Level	Feed Forward	Reflux Flow

2.2.3.2. Control Objective Translation

Translation of control objectives to idioms requires the definition of variable for use in the quasi-algebraic representation of objectives and idioms. Table 2.8 defines the necessary variables. Table 2.9 shows the objectives and idioms in their algebraic representations. The representation of objectives is X(A,B..) where X is R for regulate, C for constrain, S for stabilize, or A for anticipate.

A, B... are the variables involved in the objective. The idioms are represented by I_n(M,N...) where I indicates an idiom, n identifies the idiom, and M, N... are the elements involved in the idiom. The idioms are taken from Tables 2.1, and 2.2 provided during idiomatic control system analysis of distillation columns in [Prassinos, et.al. 1982]. Note that the entries in Table 2.9 correspond numerically to the objective descriptions in Tables 2.6 and 2.7.

Table 2.8 Variables for idiomatic control.

Variable	Definition	Variable	Definition
x_D	distillate composition	D	distillate flow
x_B	bottoms composition	F	feed flow
l_R	reflux drum level	B	bottoms flow
l_B	column base level	O	hot oil flow
p	column pressure	DV	distillate valve
dp	column differential pressure	LV	reflux valve
V	vapor boilup	BV	bottoms valve
VB	vapor bypass flow	OV	hot oil valve
L	reflux flow	FV	feed valve

The next part of the idiomatic control design is to build or draw the control system. Figure 2.9 shows the distillation column with no controls. Figure 2.10 shows the column with the primary regulatory controls, Figure 2.11 shows the column with the primary controls including constraint control. Figure 2.12 shows the column with primary and secondary controls.

2.2.4. Regulatory Control Variable Pairing

Table 2.9 Objective - Idiom pairing.

Idiom	Objective	Type	Idiom
1.	$R(x_D,D)$	Regulatory	I_1(AT,AC)
2.	$R(x_B,O)$	Regulatory	I_2(AT,AC
3.	$R(l_R,L)$	Regulatory	I_3(LT,LC)
4.	$R(l_B,B)$	Regulatory	I_4(LT,LC)
5.	R(p,VB)	Regulatory	I_5(PT,PC)
6.	C^{Hi}(dp,O)	Constraint	I_6(DPT,DPC,<)
7.	C^{Lo}(V,O)	Constraint	I_7(HIC,>)
8.	S(D,DV)	Stabilize	I_8(DPT,√,FC)
9.	S(L,LV)	Stabilize	I_9(DPT,√,FC)
10.	S(B,BV)	Stabilize	I_{10}(DPT,√,FC)
11.	S(V,OV)	Stabilize	I_{11}(DPT,√,DTT,X,BTU)
12.	$A(F,x_D)$	Anticipate	I_{12}(DPT,√,g_1(t),X)
13.	$A(F,x_B)$	Anticipate	I_{13}(DPT,√,g2(t),X)
14.	$A(D,l_R)$	Anticipate	I_{14}(DPT,√, Σ

While idiomatic control addresses the problem of synthesizing a control structure, it does not specify the method for determining the basic regulatory control structure. The problem in determining the best variable pairing is one which grows exponentially with the number of controlled and manipulated variables. The combinatorial problem is difficult to address at best, and usually requires at least a simple model of the process or some specific experience with control systems of similar processes. The selection of variable pairing is usually accomplished by using some interaction measures as well as some heuristic rules. In distillation, the relative gain array (RGA) has been used to select the composition control scheme which minimizes interaction between the control

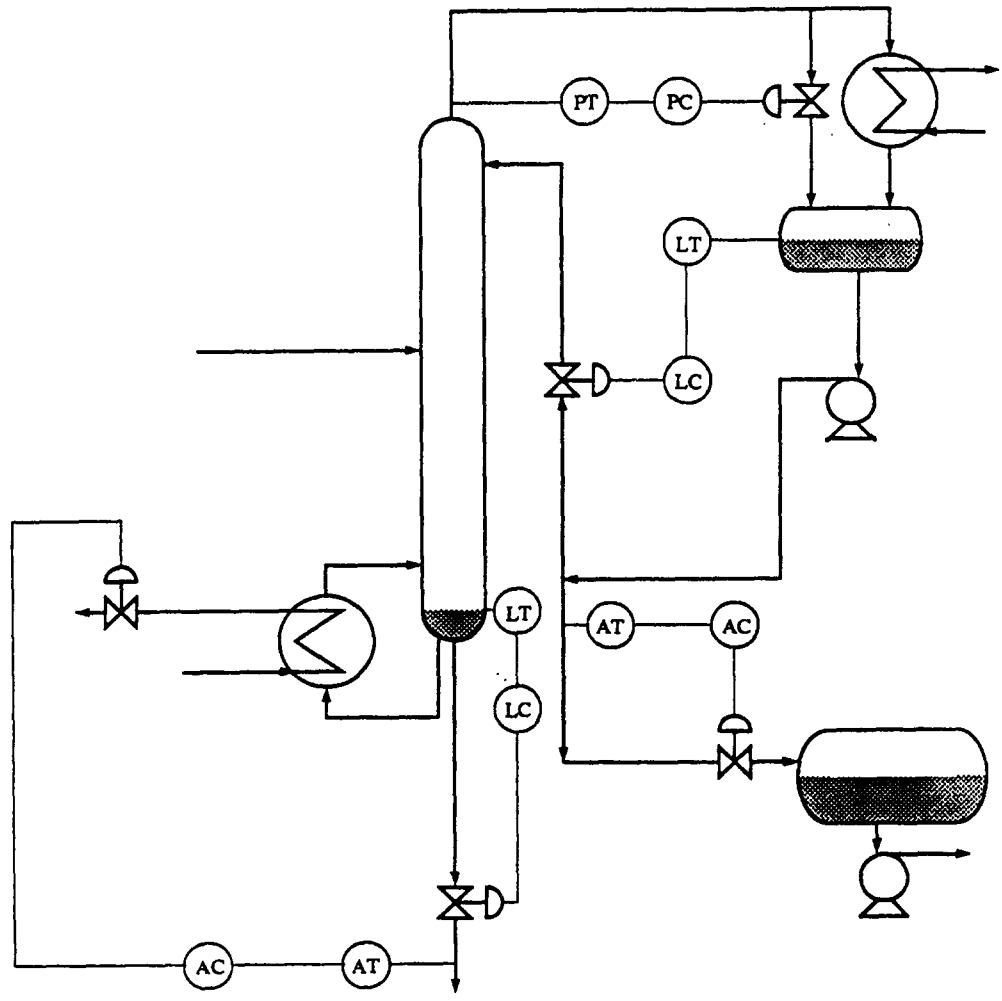


Figure 2.10. Column 2 with primary regulatory controls.

loops[McAvoy, 1983]. Heuristics are often used to complete the basic regulatory system when closing the inventory loops.

In the preceding discussion, it is implied that material balance control of the column is given precedence over composition control. This is in apparent conflict with other design procedures which for instance choose the composition control loop pairings based on the RGA. The reason for the conflict is that from

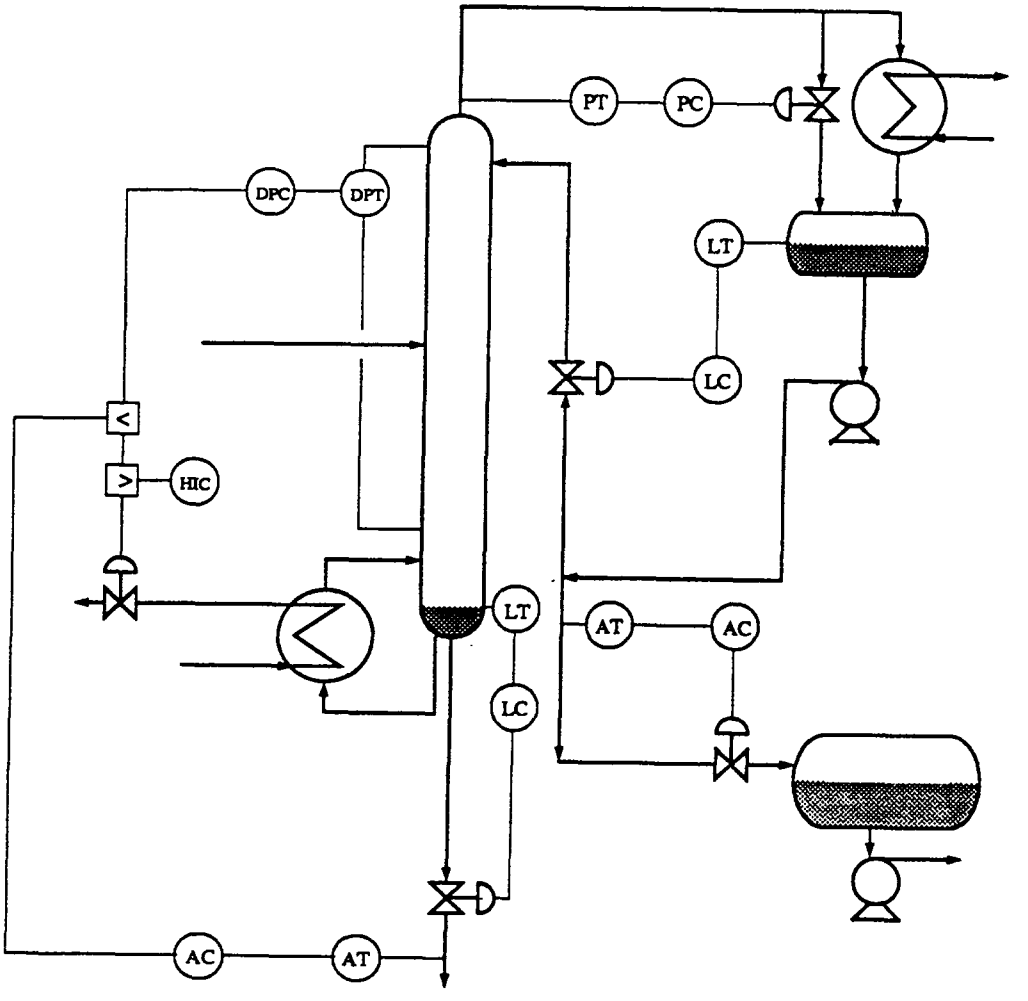


Figure 2.11. Column 2 with primary regulatory and constraint controls.

an operating and safety standpoint, material balance control is more important than composition control. Product which is off specification, while undesirable, will not generally cause operating problems, while running dry or overflowing the reflux accumulator or column base can cause interruptions in operation. By choosing composition control loops first, many times, more overrides must be implemented in order to ensure control of the levels at all times. In the case

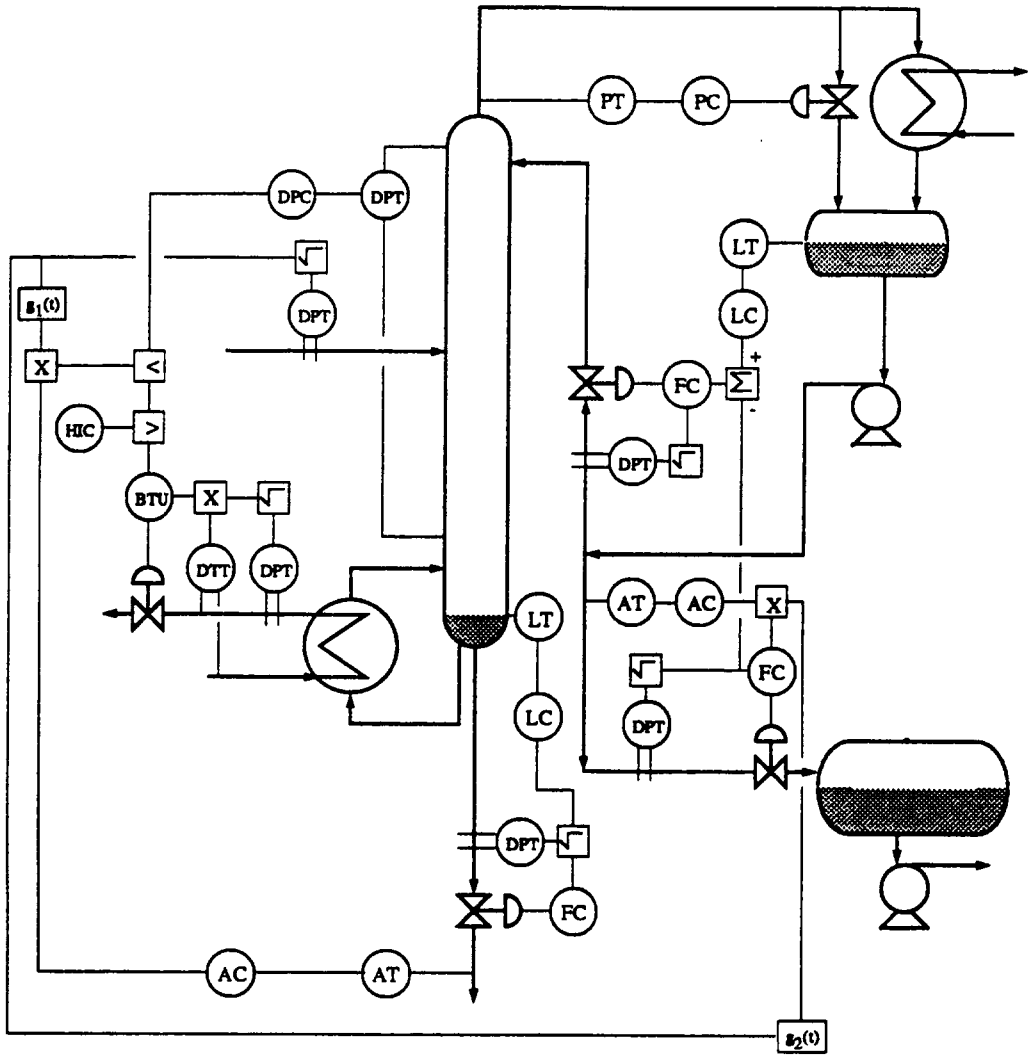


Figure 2.12. Column 2 with primary and secondary controls.

where composition control loop pairing is given priority over material balance control, upsets can cause the controllers for material balance to saturate more quickly and more often than in the case for material balance control priority. The overrides will kick in, effectively turning off the composition control loops and thereby producing product which is not within specifications. On the other hand, giving priority to the material balance control loops can result in produc-

ing product which is not within specifications due to composition control loop interaction. The difference is that by giving priority to the material balance control loops, the resulting control system can be less complicated due to fewer override controls.

CHAPTER 3

GOAL TREE - SUCCESS TREE MODEL

The previous chapter describes idiomatic control analysis and synthesis as a structured procedure by which control systems may be understood and designed. How then can one take advantage of this structure in order to build an expert system for control system design? The first step is to represent the structured procedure in a form which can be used more readily to build an expert system.

For expert system applications, it is useful to classify knowledge into two types. These two types are shallow knowledge and deep knowledge. Shallow knowledge is knowledge which is the result of experience in the problem domain, and is heuristic in nature. Shallow knowledge is easily represented in the form of if-then (or production) rules. Deep knowledge is knowledge which is based on an understanding of the underlying phenomena which create cause and effect relationships. The first expert systems used production rules as the only form of knowledge representation. Deep knowledge is more difficult to represent in the form of production rules because it is generally associated with objects or

concepts. The GTST model was developed as a convenient way to represent deep knowledge [Birky, et. al., 1989, Modarres and Cadman, 1986]for expert systems for fault diagnosis in the nuclear power industry. The GTST model is not an expert system shell, but a convenient way to model knowledge which provides a better understanding of the problem domain. The GTST model also leads directly to a specific programming solution technique; the use of a primarily backward chaining expert system shell.

The GTST model consists of a hierarchical decomposition of the problem into smaller sub-problems in a systematic, top down problem solving approach. The GTST model is one example of a systems engineering tool which can be used to effectively model problem solutions. For further information concerning systems engineering principles, the interested reader is referred to [Eisner, 1987, and Blanchard and Fabrycky, 1981]. The GTST model is not the only one which can be used, but it does provide several advantages specifically related to expert system development.

3.1. Goal Tree Structure

The goal tree model for knowledge representation consists of a tree of goals with the root goal defined at the top. The root goal is then decomposed into a set of necessary and sufficient subgoals, the satisfaction of which guarantees the success of the root goal. The first level of subgoals is further decomposed into

subgoals, and the process of subgoal definition continues until a physical entity (usually equipment) is needed to satisfy the lowest goal. There may be multiple paths which will satisfy the lowest goal, each path being a success path for the goal.

3.2. Building a Goal Tree

When building a goal tree, two necessary and sufficient rules must be followed in order to ensure its accuracy and completeness [Modarres and Cadman, 1986]. These rules are:

1 Upon looking upward from any subgoal toward the objective or tree-top, it is possible to define explicitly why the specified goal or subgoal must be satisfied.
2 Upon looking downward from any goal toward the bottom of the tree, it is possible to define explicitly how the specific goal or subgoal is satisfied.

Each block of the goal tree represents a goal which can have conditions and attributes associated with it. The conditions create a goal tree which is dynamic in nature, changing as conditions, data, or facts change. The attributes allow further description of the goal such as its priority or the order in which it must be satisfied relative to the others of its level.

To obtain the goal tree for a problem domain, the root goal is specified in an unambiguous manner. The first layer of subgoals is obtained by answering the question "How is the root goal satisfied?", or equivalently, "What must be

true in order for the root goal to be satisfied". By answering this question in an unambiguous statement, the first level of subgoals is defined. If the answer is accurate and complete, then completeness of the goal tree and knowledge is ensured. If the answer contains expressions such as "if A then B", or "first A must be satisfied, then B", then conditions and priorities must be attached to the goals. By attaching conditions and priorities to the goals, conflicts and ordering of a procedure may be represented. Verification of the subgoals can be accomplished by answering the question "Why must this subgoal be satisfied?". If the answer is clear, then the subgoal is relevant, thus ensuring accuracy of the goal tree. The construction of the goal tree continues until it is no longer possible to decompose a subgoal into other subgoals.

At the point where a subgoal can no longer be broken down into other subgoals, a success path begins. In order to construct the success tree or path, all different paths by which the subgoal can be satisfied must be represented. The difference between the success tree and the goal tree is that only one of the success paths must be satisfied for the subgoal to be satisfied. Implementation of the endnodes or success paths is provided by equipment, measurements, algorithms or perhaps rules. In all cases evidence is necessary for a success path to be satisfied.

The GTST model is well suited to represent deep knowledge and is easily translated into a frame based knowledge representation. The frame is now a common form of knowledge representation for expert systems due to the

convenience and flexibility of its object oriented nature. A frame is a type of data representation which describes an object or concept. The data includes the object or concept name and several slots for attributes. Each attribute describes a specific property of the object in an object-attribute-value triplet. For example, the object may be a car with the attribute color. The value of the attribute color could be red for instance. In expert systems which use this type of knowledge representation, it is the attribute or attributes of objects which are determined through interaction with the user and application of rules associated with the objects. Therefore, not only does the goal tree model represent deep knowledge in an organized manner, it leads directly to a convenient method of knowledge representation for the final expert system.

3.3. Idiomatic Control Synthesis as a Goal Tree

The process of idiomatic control design is easily conformed to the goal tree structure due to its inherent hierarchical structure. The following paragraphs describe the goal tree for designing the control system for a binary distillation column. The goal tree for a binary distillation column is chosen as a relatively complicated yet presentable example. The goal tree is based primarily on the idiomatic control methodology. Note that at some points, the goals include conditional statements. Furthermore, priorities apply to some goals in the same level of the tree. Conditions and priorities do not step out of the bounds of the

GTST model, but can be included as attributes of the goals in a frame – like representation.

3.3.1. The Root Goal

The root goal of goal tree is to find the "best" overall control system for column operation at all times. This goal is broken into the first level of subgoals:

1. Determine the "best" regulatory control system (for normal operation), including some optimization,
2. Determine the control system for constraint operation,
3. Determine the supportive control system for superior control during transients,
4. Implement or construct of the control system (diagram).

These goals have priorities during control system design. Subgoal 1 must be satisfied first and then subgoal 2. Subgoal 3 must be satisfied next, and finally, the control system is constructed. The goal tree showing the first level of subgoals is shown in Figure 3.1. Note that subgoals 1 and 2 correspond to the definition of the objectives for the primary control idioms, subgoal 3 corresponds to defining the objectives for the secondary or supportive control idioms, and subgoal 4 corresponds to the final step of idiomatic control design which is translation of objectives into idioms.

3.3.2. Subgoal 1: Determine the Best Control System

The goal of determining the "best" regulatory control system for normal

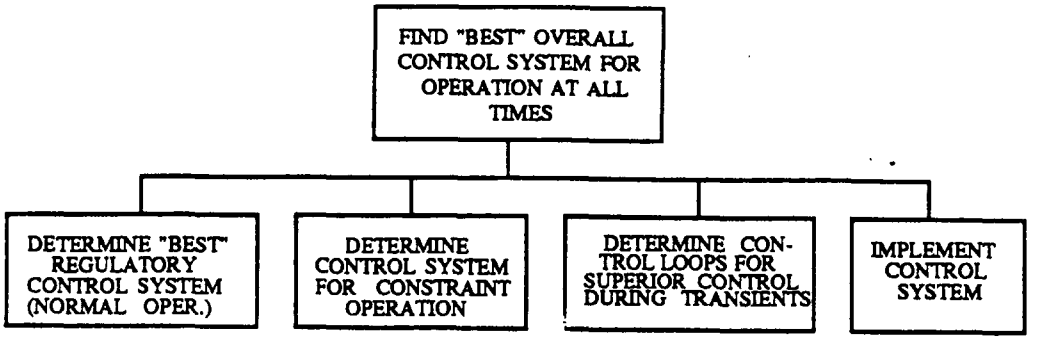


Figure 3.1. Root goal and level 1 subgoals.

operation normally involves optimizing the column operation. Dual composition control minimizes the heat input necessary for a given product specification by allowing tight control very close to set points. However, control loop interaction can be significant. Selection of the best pairing of manipulated variables for composition control is the first subgoal under subgoal 1. The remaining control loops are associated with material balance control in the column. The pressure in a distillation column affects the separation and must be controlled in order to ensure good control of the compositions. The second subgoal under subgoal 1 is control of the pressure in the column. One way to optimize the operation of a distillation column is to minimize the operating pressure. The lower the pressure, the easier the separation becomes, resulting in less heat input. However, minimum pressure operation is not always worthwhile depending on how easy the separation is and how much the relative volatility varies with pressure. If minimum pressure control is worthwhile, then maximum use of the condenser

will be necessary. Maximum use of the condenser can be accomplished by using a valve position controller to change the set point for pressure. Also included under material balance controls are level control in the reflux accumulator and column base. Subgoal 1 of Figure 3.1 is broken into the following second and third level subgoals:

1.1. determine variable pairing for composition control, including decision for dual or single point composition control,
  1.1.1. determine manipulated variable for distillate composition control,
  1.1.2. determine manipulated variable for bottom composition control.
1.2. determine variable pairing for material balance control.
  1.2.1. determine manipulated variable for pressure control, including minimum pressure control if desirable[Kister and Doig, 1981]
  1.2.2. determine manipulated variable for reflux drum level control,
  1.2.3. determine manipulated variable for column base level control.

The goal tree for normal operation (1.) is shown in Figure 3.2. The goals expressed in Figure 3.2 are related to one another because they share a common set of possible manipulated variables. Normally, some kind of priority or importance is associated with the selection of manipulated variables for composition and material balance control. The expert system described by Shinskey [Shinskey, 1986]uses the RGA to determine the manipulated variables for composition control, then proceeds to deduce the manipulated variables for material balance control. Others [Buckley, et. al., 1985]put heavier priority on material balance control, selecting manipulated variables for reflux drum and column base levels first based on column design information and other calculations.

There are probably as many methods of determining the initial variable pairings as there are control system designers. If detailed dynamic simulations are available, then they can be helpful in determining the best configuration, if not, then some simple dynamic or steady state models can provide useful information. The manipulated variable for pressure control can be selected based on information such as inerts in the system, provision for condenser flooding, and the condition of the coolant water. The value of minimizing column pressure can be determined by examining the properties of the process materials such as relative volatility, and the change in relative volatility with pressure [Kister and Doig, 1981]. Beneath the third level subgoals in Figure 3.2, various calculational or heuristic procedures make the success trees. Since the success paths vary from one designer or company to another, no specific success paths are shown. The discussion above describes the various methods which could represent the success paths.

3.3.3. Subgoal 2: Constraint Operation

Subgoal 2 for constraint operation is satisfied by subgoals which guarantee safe operation at constraints. One primary subgoal is to maintain all inventory controls. Inventory control must never be lost during operation of the column. Loss of inventory control could result in overflow of the column base, overflow of the reflux accumulator (or condenser if a flooded condenser is used), a dry

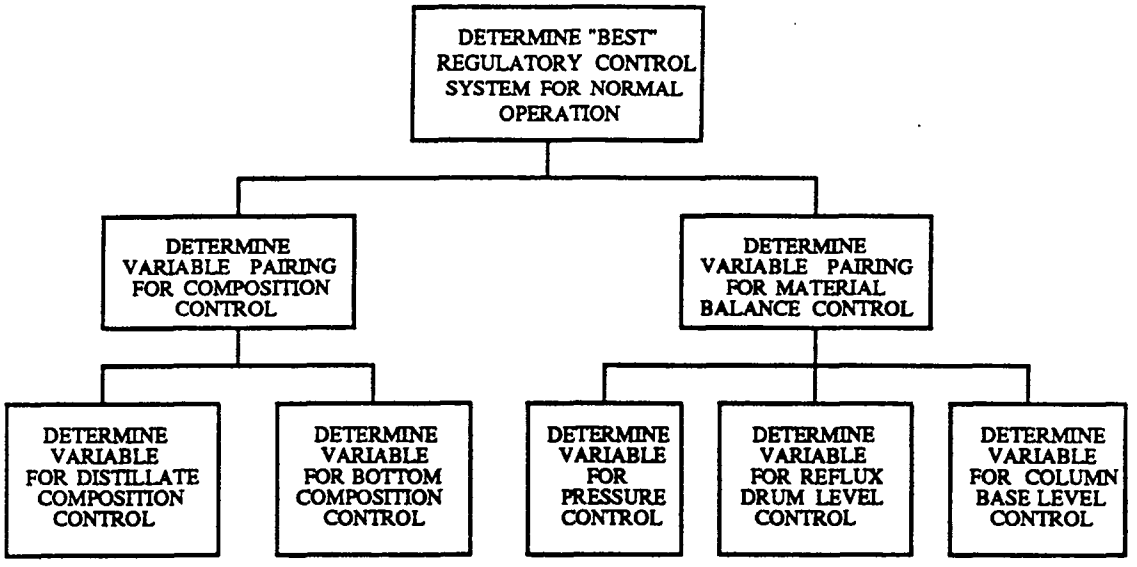


Figure 3.2. Goal tree for subgoal 1.

reboiler, or no reflux to the column. All of these conditions can be avoided if the control system is designed so that the level in the column base and reflux accumulator are always under control even during constraint operation. Another subgoal is that the column must not flood. Flooding can occur if column throughput is increased resulting in a higher volume of vapor boilup. The higher vapor rate can cause liquid backup on the trays as the vapor attempts to flow up the downcomers. Flooding can be avoided by limiting the heat input to values below the flooding point. Pressure drop across the column is normally used as a measure of approach to flooding. As flooding is approached, the pressure drop across the column increases dramatically. A third subgoal is to avoid weeping.

Weeping can occur when vapor boilup decreases below the amount necessary to keep the liquid from descending through the holes in the trays through which the vapor is intended to pass. By limiting the heat input to a value above the minimum necessary to provide enough vapor, weeping can be avoided. The minimum heat input is a function of the column design and is nearly constant. Subgoal 2 of Figure 3.1 is broken into the following second level subgoals:

2.1. Determine overrides for level control at constraint,
  2.1.1. determine override for level control of reflux drum at constraint,
  2.1.2. determine override for level control of column base at constraint.
2.2. Determine overrides to avoid flooding.
2.3. Determine overrides to avoid weeping.

The goal tree for the subgoal for constraint operation (2.) is shown in Figure 3.3. Subgoal 2.1 is further reduced to controlling the bottom level and accumulator level at all times. Control of these levels is satisfied by the success paths shown below them. The success paths for overrides on level control consist of overriding the manipulated variables for composition control. In this manner, when levels rise above or fall below allowable limits, level control will be assisted by the manipulated variables normally used for composition control. Therefore, quality control is sacrificed for safe operation when constraints are encountered. Subgoal 2.2 is satisfied by maintaining the column differential pressure below the flooding point. The success path involves using a differential pressure controller to override the heat input to the column when flooding is approached. A minimum selector is used to select the signal to be transmitted to the heat

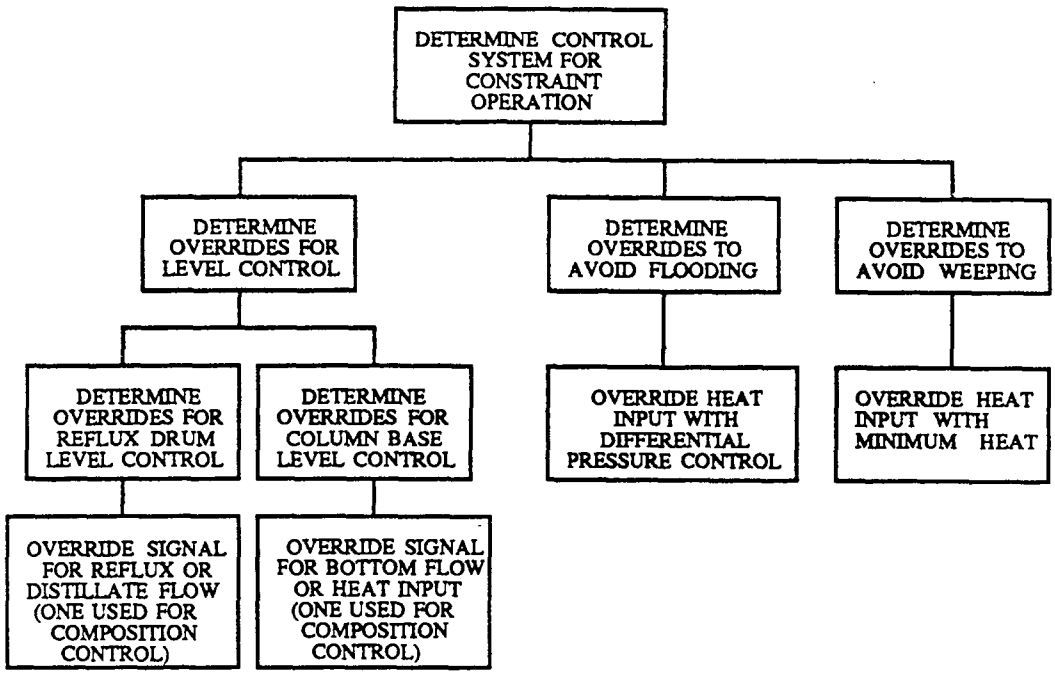


Figure 3.3. Goal tree for subgoal 2.

input system. Subgoal 2.3 is satisfied if a minimum heat input is maintained. The success path consists of a heat input controller with a constant minimum value of heat input with a maximum selector to select the signal to be transmitted to the heat input system. Since level control must be maintained, and one manipulated variable at the bottom of the column is no longer available, the bottom composition controller must be selected out of service in each case.

3.3.4. Subgoal 3: Supportive Control System

The basic regulatory and constraint controls (primary control idioms) can be improved by using some supportive control systems. Determination of the supportive controls for improved performance includes use of cascade and feed-forward control. The appropriate use of these supportive or secondary idioms is determined largely by heuristics or "rules of thumb." One common supportive control loop is the flow control loop used as the inner loop of a cascade arrangement for a slower loop such as composition. The purpose of a flow loop is to increase performance of the outer loop by rejecting disturbances in the inner loop. The flow loop can compensate very quickly for upsets in flow due to pressure changes. Another similar loop is the BTU controller which calculates the heat input by multiplying the flow of heating medium by the temperature drop across the heat exchanger. Used as an inner loop in a cascade arrangement, the BTU controller can quickly compensate for changes in medium input temperature and flow. Note that a BTU controller is possible only if the heating medium is not a condensing vapor. Another area in which improvement is possible is the level control in the reflux accumulator. If the distillate flow changes due to the action of another controller, the level in the accumulator will begin to change. The level controller will detect the change and compensate by changing the reflux flow. The level controller must sense the change before it can take corrective action. This "lag" can be avoided by using the sum of distillate and reflux to control the level in the reflux accumulator. Column operation is also improved if the control system is designed to adjust for feed

flow (throughput) upsets. Subgoal 3 of Figure 3.1 is broken into the following second level subgoals:

3.1. Design control system to avoid lag in reflux accumulator level response,
3.2. design control system to adjust for feed flow upsets,
3.3. design control system to correct for heat input (medium temp.) upsets,
3.4. design control system to correct for flow upsets quickly.

Figure 3.4 shows the goal tree for subgoal 3. Subgoal 3.1 is satisfied by a success path which specifies that the reflux level controller use both reflux (L) and distillate (D) flows ($V = L + D$). Subgoal 3.2 is satisfied by a success path which specifies that the heat input to the column be set in ratio to the feed flow (another feed forward arrangement). Subgoal 3.3 is satisfied by a success path which specifies that the controller which manipulates heat input be cascaded to a BTU controller if the heating medium is hot oil. Subgoal 3.4 is satisfied by a success path which indicates that control signals for manipulating flows should be cascaded to flow controllers. These success paths are the result of experience or knowledge gained from the literature.

3.3.5. Subgoal 4: Control System Implementation

In order to implement the control system which has been outlined above, the objectives must be translated into the idioms and placed in the proper arrangement. The first step in the implementation process is to determine the necessary algebraic operations which will translate the manipulated variables

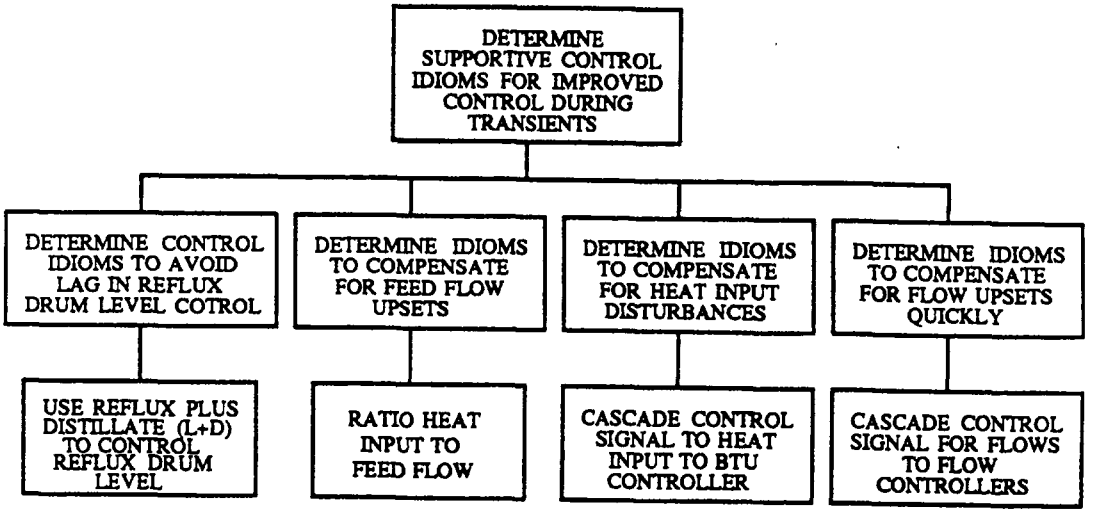


Figure 3.4. Goal tree for subgoal 3.

into the appropriate signals to be sent to the manipulators (valves). Determining the necessary operations may seem rather trivial, but in actuality, it is dependent on the choice of manipulated variables for composition control, the column material balances, and the choice of manipulated variables for inventory control. In most cases, it is straight forward to reduce the manipulated variables into the flows D, L, V, and B. In other cases (when variables such as separation factor and other transformed variables for decoupling are used [McAvoy, 1983]) it may require a rather complex computation.

Another subgoal in the implementation is to provide the proper selectors for constraint operation. Control signals must be selected properly such that control of the compositions is selected out of service rather than the inventory

controls. The final subgoal is to provide for the supportive control structures which improve control during upsets. Subgoal 4 of Figure 10 is broken into the following second level subgoals:

4.1. Determine operations necessary to transform composition and material balance control variables into the four flows D, L, V, and B,
4.2. determine necessary selectors for constraint control,
  4.2.1. determine selectors for flooding and weeping,
  4.2.2. determine selectors for level control
4.3. determine feed forward and cascade control loops to enhance control during transients.

The goal tree for subgoal 4 is shown in Figure 3.5. Subgoal 4.1 is satisfied by multiplying and adding control signals. Subgoal 4.2 is satisfied by using a minimum selector between the control signal from the differential pressure controller and the signal from the regulatory control loop and using a maximum selector between the control signal from the regulatory control loop and the minimum heat input. If the level in the column base is controlled by heat input to the column, then the level control loop is overridden by the flooding and weeping controls. Overrides must be provided for level control by the bottom flow when heat input is manipulated by an override. A high level override controller and low level override controller must be provided to override the regulatory control signal to the bottom flow. A maximum selector should select the larger of the high level override and the regulatory signals, and a minimum selector should select the smaller of the regulatory and low level override signals. Subgoal 4.3 is satisfied by multiplying the heat input signal by feed flow, implementing a BTU controller for heat input if a hot oil is used, and using flow controllers on

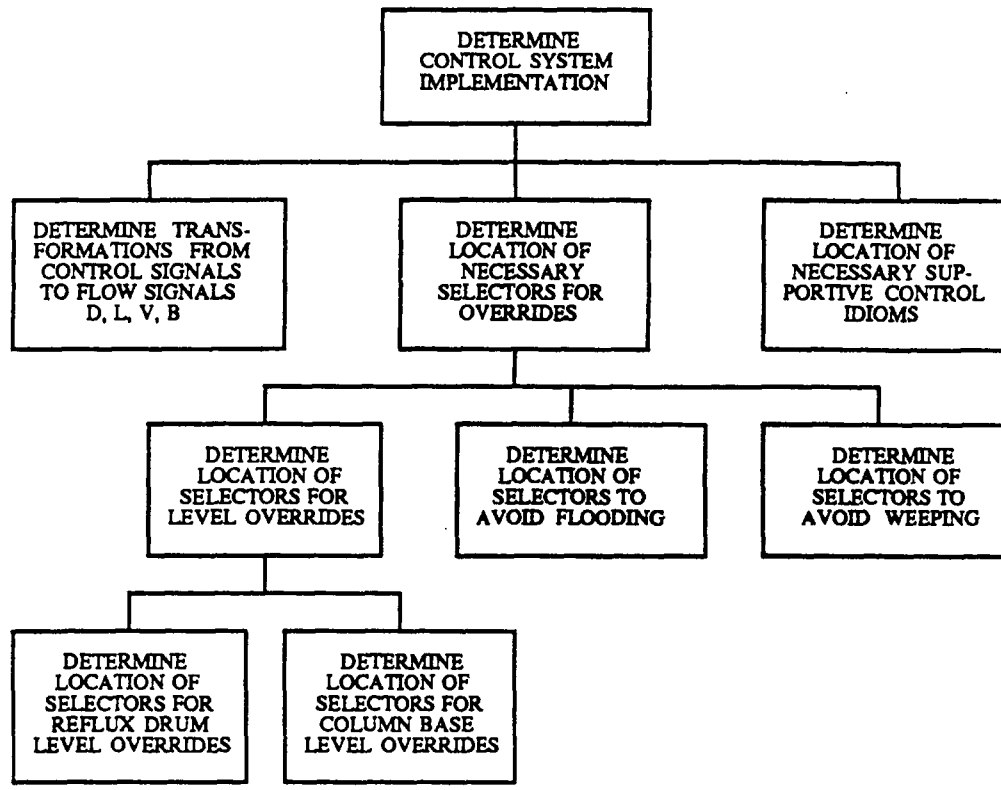


Figure 3.5. Goal tree for subgoal 4.

L, D, and B (and steam flow if steam is used as the heating medium).

The preceding discussion demonstrates that the GTST model satisfies the requirements for modeling the idiomatic control design methodology. Similarly, it provides a framework for modeling the calculational part of control system design as implementation of endnodes or success paths, and has been used extensively as a representation of knowledge for production rule based expert

systems. The calculational procedures can be modeled as the success paths to satisfy a low level goal.

Due to its generality, the GTST model can be used as the backbone for the knowledge base of the expert system for analysis and design of process control systems. The GTST model is easily converted into a frame based knowledge representation. Since the frame is a very convenient and desirable form of knowledge representation for expert systems, the initial representation of knowledge by the GTST model serves as a tool to organize the knowledge into a highly structured form. The high degree of structure facilitates building the expert system particularly if a frame based representation is used. Another advantage to this method is that the idioms are easily represented by frames.

CHAPTER 4

DICODE: DISTILLATION CONTROL SYSTEM DESIGN

In order to demonstrate the applicability of the goal tree - success tree for use as an intermediate form of knowledge representation during expert system construction, a control design application was sought. From discussions with E.I. du Pont de Nemours & Co., it became known that their leading expert in distillation control design was to retire in the end of 1987. It was desired to retain some of his knowledge in the form of an expert system for use by DuPont. The distillation control design problem was chosen as the example to demonstrate the proposed expert system construction methodology. The application is now known as DICODE for DIstillation COntrol DEsign.

4.1. Project Definition

Early in 1987, DuPont found that the criteria set forth by D.A. Waterman [Waterman, 1986]for the possibility, justification, and appropriateness of expert

system development fit the task of designing control systems for distillation columns [Birky, McAvoy, and Tyreus, 1988]. Other criteria were specified for the applications and capabilities of the expert system. The system would help the control engineers design the controls for a distillation column, giving sufficient warnings and recommendations to ensure that poor choices of control objectives and equipment configurations were avoided. The results of the consultation were to include a graphic representation of the design and a text description which included justifications. Another requirement was the ability to provide explanations during consultation as to the meaning of, need for, and relevance of information asked of the user.

Since it is virtually impossible to capture all of the knowledge of an expert in distillation control design, the scope of the problem was originally limited to the overhead system of the distillation column which consists of the condenser and associated equipment. The control objectives were to provide level control of the reflux accumulator or flooded condenser, pressure control if necessary, condensate temperature control if possible and desired, and finally distillate composition control. Later, the system was extended to include design of the controls for the entire column, including the column base.

4.2. Goal Tree - Success Tree for DICODE

The first step in designing the expert system, was to model the problem

solution technique in the form of a goal tree. As described in Chapter 3, the goal tree represents the problem solving methodology. The goal tree shows the decomposition of the root goal, which is to design the "best" control system for the distillation column within the scope of the knowledge available.

Figure 4.1 shows the goal tree for solution of the control design problem of a binary (two product) distillation column. Note that the goal tree does not contain any specifics of the final solution. That is, it does not specify how the lowest level goals are satisfied.

Figures 4.2 through 4.7 show the success paths for the regulatory control system of the binary distillation column. Note that the success paths specify how the lowest level goals are satisfied. All possible ways that the lowest level goals can be satisfied are shown as success paths, but no information is provided to determine which success path will be the one to satisfy the goal. The information which determines which success path is to be used to satisfy a specific lowest level goal is unique to the expert who solves the problem, while the goal tree - success tree is used to represent or organize the problem solution method without specifics.

The success paths which satisfy the lowest level goals for constraint operation are similar to those of the lowest level goals for regulatory control. For reflux drum level control, the manipulated variable used for distillate composition control is overridden. Therefore, the success paths below the reflux drum level constraint consist of the reflux and distillate flows. For column base level

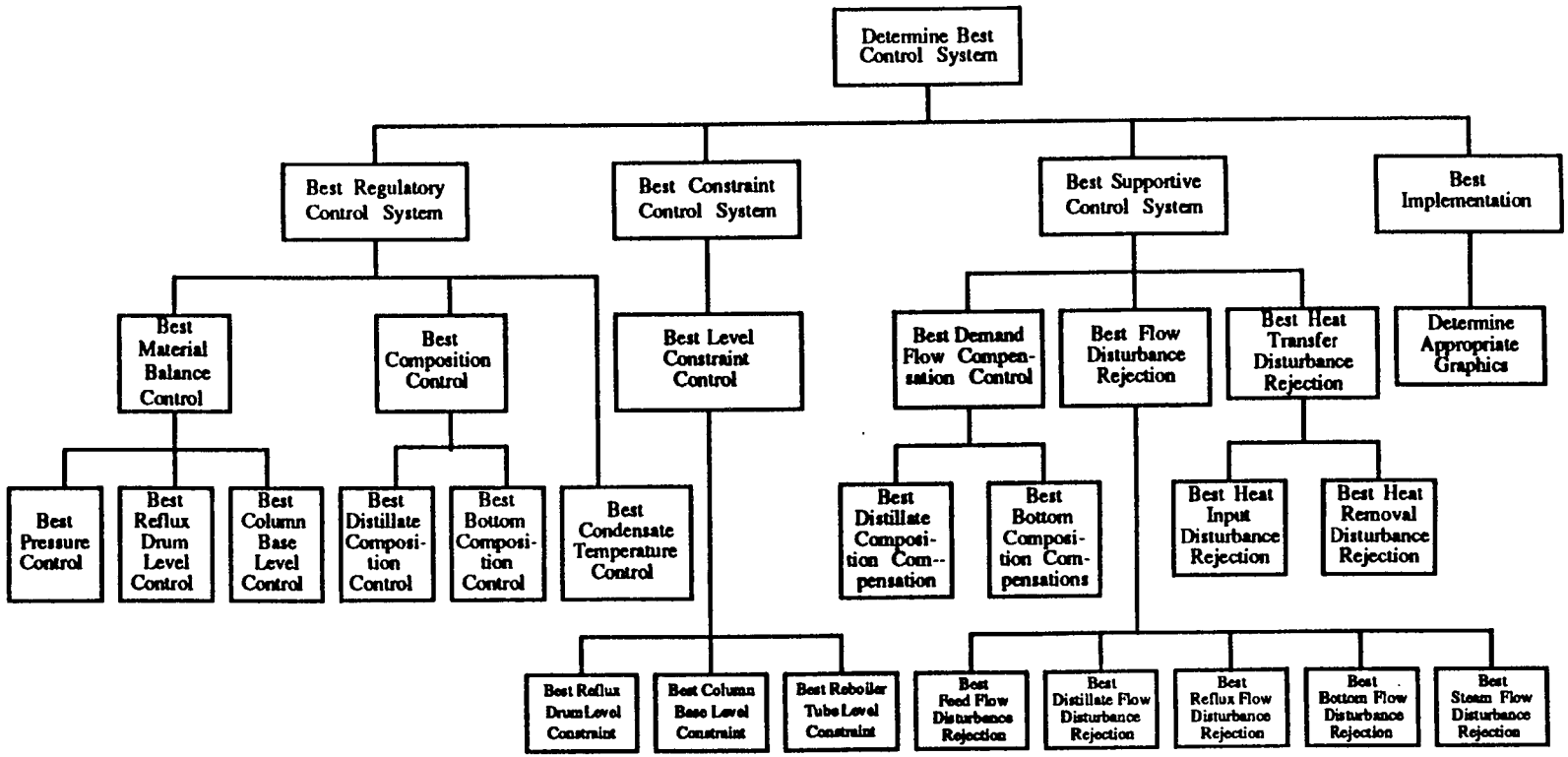


Figure 4.1 Goal Tree for DICODE

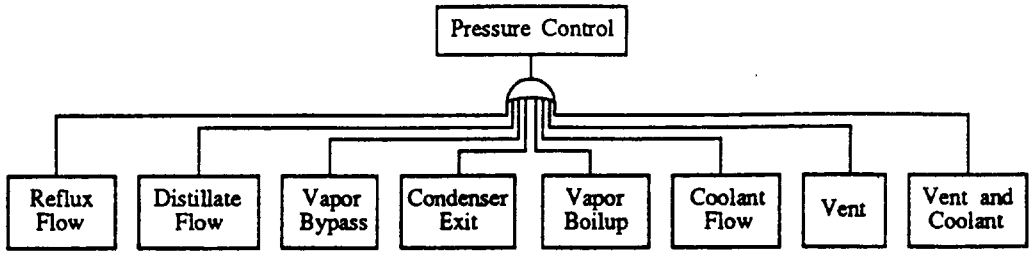


Figure 4.2. Success paths for pressure control goal.

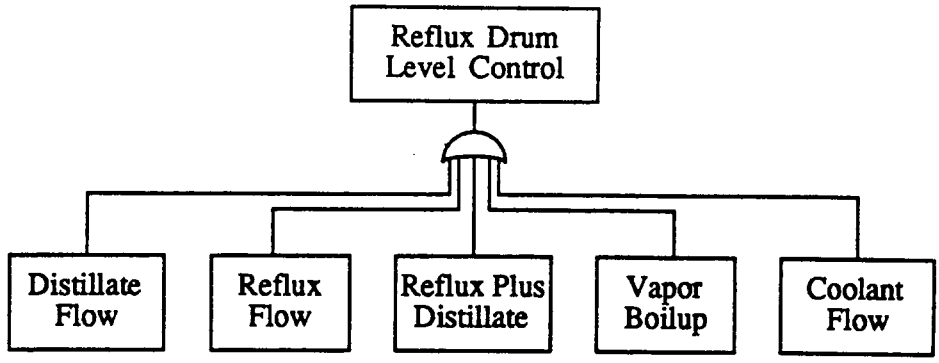


Figure 4.3. Success paths for reflux drum level control.

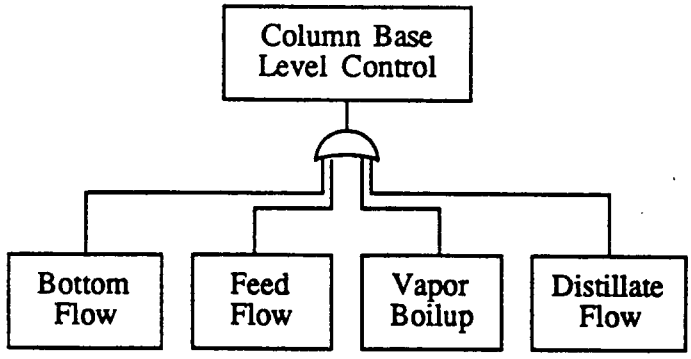


Figure 4.4. Success paths for column base level control.

control, the bottom column manipulated variable used for composition control

success paths must have some priority associated with it. Implementation of priority and ensuring that each success path satisfies a single lowest level goal in the regulatory control branch of the goal tree is discussed further in the following sections. A generalization of the method used to solve this problem is presented in chapter 5.

4.3. Choosing an Expert System Shell

A particularly difficult element involved in any expert system project is choosing an appropriate shell for development. Many aspects of the project should be considered during the shell selection process. A major division of shells can be based on the chaining method - backward or forward. Another consideration is the method of knowledge representation provided by the shell. The frame is a convenient and powerful form of knowledge representation available in most shells. The necessity of access oriented programming (use of demons) must be considered. If certain portions of the problem solution require procedures, access oriented programming is very useful. If knowledge sources exist in the form of information provided by programs such as simulations, or spread sheets, then the ability to execute and communicate with external programs must be provided.

4.3.1. Expert System Function

One major concern is the function of the expert system. In the past, if the function were design, then the choice was for a forward chaining shell. With the knowledge gained through application of the GTST model to the distillation control design problem, it has been determined that a backward chaining can also be used. With the knowledge represented in the form of a goal tree, an expert system shell which provides backward chaining will be adequate to execute the goal tree part of the problem. For DICODE, the goal tree is represented directly in the knowledge base in the form of rules. The following rule is the one which satisfies the root (highest level) goal for DICODE.

```
Root goal satisfied:
if
     Subgoals:Best Regulatory Controls>Status = satisfied and
     Subgoals:Best Constraint Controls>Status = satisfied and
     Subgoals:Best Supportive Controls>Status = satisfied and
     Subgoals:Best Implementation>Status = satisfied
then
     Goals:Best Control System>Status = satisfied.
endif.
```

The syntax of the rule is explained later, but the important feature is that the root goal (Goals:Best Control System) is satisfied if all of its four subgoals are satisfied.

4.3.2. Frame Representation for Knowledge

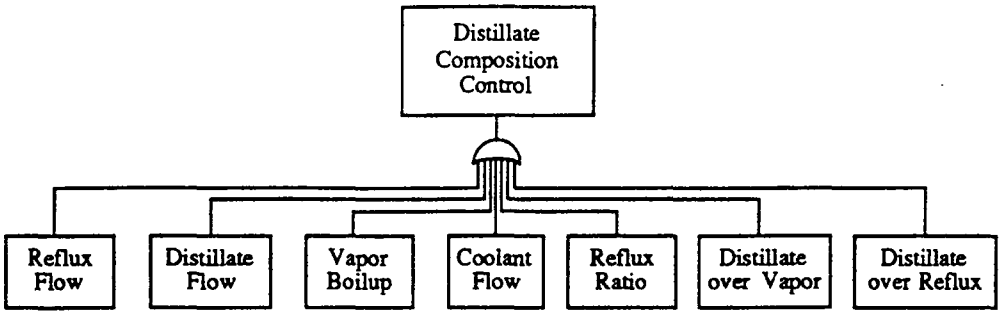


Figure 4.5. Success paths for distillate composition control.

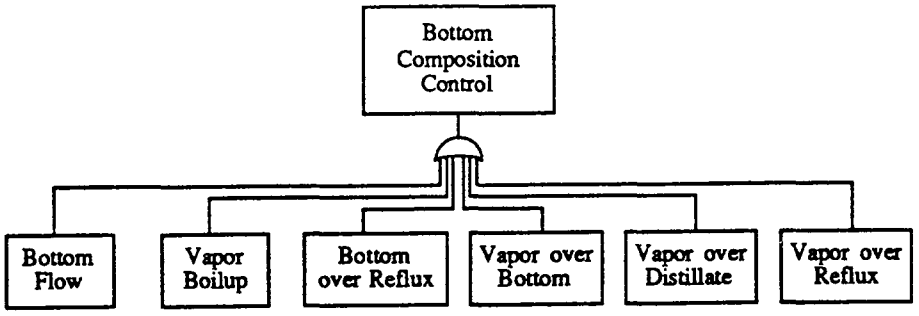


Figure 4.6. Success paths for bottom composition control.

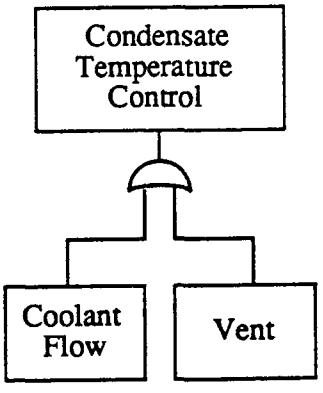


Figure 4.7. Success paths for condensate temperature control.

is overridden. The success paths for column base level constraint are the bottom

flow and vapor boilup or heat input. The goal of maintaining liquid level above the tubes of the reboiler is satisfied by a success path consisting of overriding the heat input to the column.

For the lowest level goals associated with supportive control, compensation of the distillate control for changes in demand flow is satisfied by success paths consisting of feed forward of feed flow, distillate flow, or bottom flow. The feed forward controller outputs are used as the inputs to ratio controllers that receive set points from the composition controllers. Each of the lowest level flow disturbance rejection goals has one success path below it consisting of a flow controller manipulating the valve on its stream. The heat input disturbance rejection goal is satisfied by one of two success paths which are a BTU controller and a steam flow controller. The heat removal disturbance rejection goal is satisfied by one success path, a heat calculation controller.

The lowest level goal of determining the appropriate graphics consists of matching the correct graphical representation to each of the controllers obtained in satisfying the lowest level goals for regulatory, constraint, and supportive control. This graphics goal could be further broken into subgoals if desired and each lowest level goal would then be described as determining a graphical representation of a controller in the context of the other controllers.

One very important point to note is that many of the lowest level subgoals have common success paths. In the regulatory control design, any one success path can satisfy only one lowest level goal. Therefore, the goal satisfaction by

As mentioned earlier, the frame method of knowledge representation for expert systems is powerful. It allows definition of a parent or generic frame of which instances are created during run time, with as many separate instances of the frame as is necessary to solve the problem. Each instance of the parent frame inherits all of the slots of the parent, as well as any facets of the slots. Therefore, if demons are provided by the shell, the demons apply to all instances of the frame.

For the design problem, access oriented programming (provided through demons) is useful. Demons can be used to provide recommendations and warnings during consultation. Providing recommendations based on previously provided design information is extremely difficult if the shell does not allow the use of demons. Furthermore, demons are absolutely necessary for the design problem in which success paths appear below more than one lowest level goal, but may be used to satisfy only one of the goals under which they appear. In order to appreciate the reason for the need for demons, a lengthy explanation is required. That explanation appears in Section 4.5. In the problem addressed by DICODE, the need for demons is a direct result of the multiple appearance of success paths under different lowest level goals.

4.3.3. External Call Capability

Since knowledge in the form of simulations and other involved calculations

is used by experts to design controls for distillation columns, the ability to run and communicate with external programs is necessary for DICODE. Also, since the end result of a consultation is a graphical representation of the equipment and control system, graphics is necessary. If the shell does not provide graphics, then an external graphics program must be executed. DICODE uses a FORTRAN executable to display the graphics results of a design.

4.3.4. Flexibility

One consideration which is often overlooked is the flexibility of the expert system shell. That is, how easily can it be modified or extended in order to provide problem domain specific facilities. Many of the very expensive expert system shells (commonly referred to as expert system building tools) require the addition of these modifications in the form of LISP functions. Unless the knowledge base author knows LISP, he must not only learn the syntax of the shell and how to use it, he must also learn to program in LISP to provide modifications which cause the shell to behave as he needs. Some of the less expensive shells allow the knowledge base author to guide the expert system execution according to his needs without programming outside of the shell. Simply because the shell is more expensive and claims to provide many extras does not mean that the knowledge base author will be able to use the shell in the most effective way.

4.3.5. Cost

The final consideration when selecting a shell for expert system development is the cost of the shell if it must be purchased. Another consideration is the amount of time needed to learn the shell. This amount of time can be considerable. If an inadequate shell is chosen, the time used to learn the shell and develop a prototype is lost. A shell which provides all the features necessary to implement the expert system at the lowest price is of course the best choice.

4.4. KES^{TM} PS

After careful consideration of the requirements of the expert system shell, KES^{TM} PS was chosen for DICODE implementation. KES^{TM} is primarily backward chaining, provides knowledge representation in frames, allows the use of demons, can communicate with and run external programs, and is reasonably priced. The following sections describe the salient features of KES^{TM} using examples from the DICODE knowledge base [KES, 1988].

4.4.1. Classes

KES^{TM} uses frames to represent objects and concepts. In the KES^{TM} syntax, a *Class* of objects is defined in the knowledge base. The class has slots

for attributes, and facets to describe the slots. One facet is the value of the attribute, another describes the type of attribute - real number, integer, string, single valued, or multi-valued, another can provide a default value, another can calculate a value from other numeric attributes, and another can give constraints on the value of the attribute. A type of facet called a textual attachment is provided for each attribute. The attachments include a question to be asked when the value of the attribute is needed, an explanation for the attribute, and a reason why a value for the attribute is needed. Other slots in the frame can specify default members of the class (instances of the frame)and a class from which it inherits attributes. The following is an example of a class definition from DICODE.

```
Manipulated Variables: [inherits: Variables]
    [default: reflux, distillate, coolant, vent, bottom,
        vapor, feed, vapor bypass, condenser exit,
        vent coolant]
    attributes:
        paired: truth [default: false].
    %
endclass.
```

During execution of the expert system, instances of a frame (members of a class) can be created and destroyed. If default members are provided, then the class has members when execution begins. Otherwise, members are created in rules and demons, and destroyed in demons.

Since one of the slots of a frame or class is the name of a class from which it inherits attributes, the inheritance structure is allowed. Any class can inherit

from only one other class. Inheritance allows the members of a class and its subclasses to be referenced as a group in rules and demons. Functions are provided to determine if a member is from the superclass or a specific subclass. In KES^{TM}, an attribute of a specific member of a class is referenced as

classname : membername > attributename.

4.4.2. Attributes

Besides the attributes attached to frames as slots, KES^{TM} allows the use of free or *global* attributes. Global attributes are not attached to a frame, but in every other respect are identical to the attributes of a frame. In fact, the syntax for defining global attributes is identical to defining an attribute of a frame, except the definition is not within the definition of a frame. Each attribute can have many facets including demons, default values, textual attachments, and attribute type.

The attributes can be used to describe a piece of information which does not describe a specific concept or object. In DICODE, the global attributes are used to describe concepts such as the process stream sensitivity to the atmosphere. A major use of the global attributes is for string manipulation. One result of the consultation is a text description of the equipment and control system with explanations and justifications. The descriptions for each piece of equipment, sensor, and control loop are contained in string attributes which are slots of

frames. In order to display and save the descriptions in a specific format, global string attributes are used.

4.4.3. External Program Use

KES^{TM} allows the use of external programs. An expert system built using KES^{TM} can write communication files and then exit and run an external program which uses the communications file and perhaps writes its own communication file to be read by the expert system. When the external program execution is complete, the expert system resumes execution. The expert system does not care if the external program is another expert system or a FORTRAN executable. KES^{TM} is also embeddable in C programs and FORTRAN programs if the FORTRAN allows subroutine and functions to pass by value. Embeddability allows application programs to contain KES^{TM} expert systems.

All external programs are defined in a section of the knowledge base called the *externals* section. When the external program is called during expert system execution, the shell submits a string to the operating system to tell it what to do. If the string is an operating system command such as copy a file, then the command is executed and control is returned to the expert system. If the string is a command to run a FORTRAN executable, then the program is executed and control is returned to the expert system. The following is an example definition of an external program definition from DICODE.

```
graph:
    [program: "RUN DECODE"].
```

The external named *graph* submits a command to the operating system which is an instruction to run the program DECODE. DECODE is the FORTRAN executable which reads a communication file written by DICODE containing the names of data files which contain information to draw pictures. DECODE then uses a graphical kernel system (GKS) to draw the graphical results which were inferred by DICODE.

4.4.4. Rules

Rules are defined in a KES^{TM} knowledge base within a section called the rules section. Rules consist of a name, an antecedent, and a consequent. The name identifies the rule for reference during explanation. The antecedent contains the *if* part and can be made of conjunctions and disjunctions of tests on attribute values. If the antecedent is evaluated as true, then the consequent is executed. The consequent contains assignments of members to classes, assignment of values to attributes, and commands to display messages or stop.

The rules are used to backward chain to obtain the value of an attribute. The back chaining can be started by the need for an attribute value in a command from the actions section, or in a demon. Backward chaining continues until no source of information exists for the value of an attribute. The user is

then asked to provide a value for the attribute. If along the way, the conditions of a demon are met, then the demon fires and the actions of the demon could instigate backward chaining along another path. The following is an example of a rule from DICODE.

```
Root goal satisfied:
if
     Subgoals:Best Regulatory Controls>Status = satisfied and
     Subgoals:Best Constraint Controls>Status = satisfied and
     Subgoals:Best Supportive Controls>Status = satisfied and
     Subgoals:Best Implementation>Status = satisfied
then
     Goals:Best Control System>Status = satisfied.
endif.
```

In the rule shown above, specific members of a class of objects are referenced. It is also possible to write rules which refer to the entire class of objects. The rule below gives an example of a rule which refers to an entire class of objects.

```
Inner cascade flow loop on reflux:
c:Controllers
if
     c>manipulated = reflux
then
     assertclass Supportive Controllers = FC reflux.
     Supportive Controllers:FC reflux>manipulated =
          reflux valve.
     Supportive Controllers:FC reflux>Short Description =
          scfcre.
endif.
```

The rule above binds the variable *c* to all *Controllers*, and is considered as many times as there are *Controllers*. If any one *Controllers*' attribute *manipulated*

has the value *reflux*, then a *Supportive Controller* named *FC reflux* is created and values are given to its attributes *manipulated* and *Short Description.*

4.4.5. Demons

Demons are defined in another section of the knowledge base. They are attached to the frames by binding a variable to the class name. Then the demon applies to all instances of the class or frame. The demons can be thought of as guards. They remain passive until the antecedent is satisfied. When the antecedent has been satisfied, the actions or procedures specified in the consequent are executed immediately while inferencing is put on hold.

The consequent of a demon can contain any combination of many commands available in KES^{TM}. Examples of commands are to display a message, obtain the value of an attribute, assign the value of an attribute, conditional or *if* blocks (if a condition is true then actions), erase the value of an attribute, assign a member to a class, erase the members of a class, etc. The following is an example of a demon from DICODE which obtains the value of an attribute when no rules are able to assign one.

```
Get heating medium:
when
    status(Heating Medium) = unknown
then
    erase Heating Medium.
    askfor Heating Medium.
endwhen.
```

The need for this demon may not be clear. The KES^{TM} shell tries to assign values to attributes by firing the rules. If no rules exist which could assign a value to the attribute, then the user is asked to provide a value. If however, rules exist which can assign a value to the attribute, but none of them succeed, then the attribute receives a value of *unknown*. If the value is needed, then it is necessary to provide a demon such as the one given above in order to ensure that a value is obtained for the attribute.

4.4.6. Actions

The actions section of a knowledge base for KES^{TM} contains algorithmic programming in KES^{TM} syntax. When an expert system is run in KES^{TM}, execution begins with the actions section. Typically, the expert system is menu driven by using attributes as instructions. Backward chaining is started by a command to obtain the value of an attribute, or a command which contains an attribute whose value is undetermined. The actions section allows extensive customization of the expert system execution. The knowledge base author is able to force the expert system to do what is needed besides backward chaining when necessary. The actions section allows if-then structures similar to FORTRAN, while looping like PASCAL, looping through the members of a class, giving messages, erasing attribute values and class members, asking for attribute values, writing information to files, reading information from files,

running external programs, and many other options.

The flexibility of KES^{TM} is superior to most expert system shells due to the wide range of commands available in the procedural actions section. The consequent of a demon can contain any command which can be used in the actions section. The demon facility used in this way can provide wide excursions from the original goal of the expert system, thereby allowing flexibility and control of the expert system execution.

4.5. DICODE Implementation

Once KES^{TM} was chosen for expert system implementation, the GTST model of the problem solution had to be translated into a knowledge base. As was stated earlier, KES^{TM} was chosen because of its apparent compatibility with the GTST model.

4.5.1. Role of the GTST

The goal tree for design of controls for a binary distillation column is directly implemented in the DICODE knowledge base. The goals are represented by four classes, *Goals, Subgoals, Subsubgoals,* and *Subsubsubgoals.* Subsubsubgoals inherits from Subsubgoals, Subsubgoals inherits from Subgoals, and Subgoals inherits from Goals. The class Goals has a single member named *Best*

Control System, and only one attribute named *Status*. The attribute Status is single valued, and can take one of two values *satisfied*, or *unsatisfied*, with a default value of unsatisfied. Since each subgoal class inherits from the class of goals directly above it, all goals have the attribute Status. The lowest level subgoals each represent one control objective – to select a manipulated variable to control a single measured process variable.

At this point it is useful to outline two different interpretations of the purpose of the success tree which lies beneath the goal tree for DICODE. In strict adherence to the GTST model, the success paths would describe ways in which the lowest level goals can be satisfied. Since these goals are described as "Determine the best manipulated variable to control distillate composition" for instance, the success paths should describe the ways in which this goal can be attained. In selecting the manipulated variable for a controlled variable in the SISO (single input single output) control system, each expert would have his own criteria which could include results from a dynamic simulation, estimates of integrated error associated with a given disturbance, RGA analysis, RDG analysis, SVD, or perhaps just heuristics based on design data. In this way, the implementation of the end node subgoals is accomplished through use of tools which are applied by the expert.

Definition of the GTST model states that the success paths are joined to the lowest level subgoals by *or* gates. In the case of control system design, any combination of tools can be used in order to satisfy the lowest level subgoal of

choosing the appropriate manipulated variable. Furthermore, the success tree is specific to the expert whose knowledge is being represented. Therefore, it is difficult to represent the success paths in a general way which would apply to the design of control systems independent of the expert whose knowledge is implemented. A much different interpretation of the purpose of the success paths is used in the representation of the knowledge for the expert system DICODE.

If the lowest level subgoals are interpreted as "Control distillate composition" for example, then the success paths would represent the ways in which this goal can be satisfied. With this interpretation, the success paths would simply represent the various manipulated variables available for controlling the process. Thus, the success tree is generalized to a form which is expert independent at the expense of loss of structure. That is, the success tree is simply one layer, which could be quite large, repeated (theoretically) under each lowest level subgoal.

In the DICODE knowledge base, the success paths are represented as manipulated variables. The lowest level subgoals are then satisfied when a manipulated variable or success path is selected to be used to control the controlled variable or lowest level subgoal. In adherence to the GTST in the knowledge base, a class called *Success Paths* would have been defined which represent the manipulated variables.

Once the class structure had been created to represent the goals and success

paths, the rules which link the goals to their respective subgoals were easily written. The rules to determine which success path satisfies which lowest level subgoals were a bit more difficult to write. The section which describes the rules addresses the subject in detail.

4.5.2. Classes in DICODE

DICODE uses the class or frame to represent many of the concepts involved in the distillation control system design problem. Other than the goals and success paths mentioned above, DICODE uses frames to represent variables (both manipulated and controlled), controllers, equipment (including condenser, reboiler, and transmitters), and the idioms needed to build the control system. (For an in depth discussion of idiomatic control, see chapter 2.) Each of these classes has an attribute named *Description* which is used to give a description of the member. The description is used for textual results of the consultation.

4.5.3. Global Attributes

DICODE uses global or free attributes to describe information which is not necessarily attached to a particular concept or object. These attributes allow the user to employ DICODE in the way he likes best by selecting options from menus. The global attributes are also used to represent concepts such as the

amount of inerts in the process stream, the sensitivity of the process stream to the atmosphere, the process operating pressure, etc. The global attributes are for the most part input provided by the user, although some are calculated and a few are inferred.

4.5.4. Rules

In DICODE, rules fall into five groups. The first group links the goals together in the goal tree. The second satisfies the lowest level subgoals due to the success of any of the success paths below it. The third group deals with the success path selection for specific lowest level subgoals. The fourth group determines what graphical and text descriptions are needed to represent the equipment and control system design. The fifth group of rules represents the relationships between user provided information. The next paragraphs describe each group of rules in detail and provides examples.

The rules which link the goals together into the goal tree are simple and quite easy to write. The antecedent of each rule contains the conditions for one single goal to be satisfied. Each of the goal's subgoals must be satisfied for the goal to be satisfied. The consequent establishes the satisfaction of the goal if the antecedent is true. The following rule is an example taken from DICODE.

Root goal satisfied:
if
    Subgoals:Best Regulatory Controls>Status = satisfied and
    Subgoals:Best Constraint Controls>Status = satisfied and

Subgoals:Best Supportive Controls>Status = satisfied and
Subgoals:Best Implementation>Status = satisfied
then
Goals:Best Control System>Status = satisfied.
endif.

One rule similar to the one above is provided for each goal which is not a lowest level subgoal.

The next group of rules establishes the satisfaction of the lowest level subgoals based on one of two conditions. The lowest level subgoal is satisfied if a controller has been found which links the controlled variable represented by the subgoal with a manipulated variable which represents a success path. Since the lowest level subgoal is a control objective, there arise cases where the subgoal is satisfied by conditions which make the control objective unnecessary or impossible. An example is that it is not necessary to control the accumulator level when an internal condenser is used. The following rule implements both ways of satisfying the goal to control reflux drum level.

Accumulator level subgoal satisfied by internal or control:
if
Condensers:overhead condenser>Type = internal or
status(Material Balance Controllers:LC reflux>manipulated)
= known
then
Subsubsubgoals:Best Accumulator Level>Status = satisfied.
endif.

The rule above states that if a manipulated variable has been established to be used to control the accumulator level, then the subgoal has been satis-

fied. The next group of rules selects which of the possible and available success paths is to be used to satisfy the subgoal. The rules of this type contain the experts knowledge which is case specific. Under what conditions should the reflux be used to control the accumulator level? The following rule from DICODE establishes when the reflux flow should be used to control accumulator level.

```
Reflux drum level controlled by reflux flow:
if
    Controlled Variables:reflux level>paired = false and
    Top Product = liquid and
    Reflux Ratio ge 5.0 and
    Scaled Vapor to Bottom Ratio le Reflux Ratio and
    Manipulated Variables:reflux>paired = false
then
    assertclass Material Balance Controllers = LC reflux.
    Material Balance Controllers:LC reflux>manipulated
        = reflux.
    Material Balance Controllers:LC reflux>controlled
        = LC reflux.
endif.
```

As can be seen, the rule above is quite involved. It states that if the reflux drum level has not been paired to a manipulated variable, the distillate product is a liquid, the reflux ratio is greater than five, the scaled vapor to bottom ratio (vapor boilup to bottoms flow ratio divided by two) is less than or equal to the reflux ratio, and the reflux flow has not been used for another purpose, then create a new member of the class *Material Balance Controllers* named *LC reflux* and give its attributes *manipulated* and *controlled* the values *reflux* and *LC reflux* respectively. Two of the conditions in the antecedent deal with ensuring that the controlled variable and manipulated variable being considered are valid.

That is, the controlled variable has not been paired (in this case meaning it is currently an active goal), and the manipulated variable *reflux* has not been paired (in this case meaning it is still available to be used for control). The need for these conditions will become clear in section 4.5.7 which describes DICODE execution.

Another group of rules determines which files need to be used by the graphics program in order to properly display the graphical results of the design and which descriptions should be displayed as the text results. The following rule is an example from the DICODE knowledge base.

```
Level control by reflux flow controller:
if
     Condenser Use = normal and
     Material Balance Controllers:LC reflux>manipulated
          = reflux
then
     assertclass Loops = LEVREF.
     Material Balance Controllers:LC reflux>Description = levref.
endif.
```

The rule above states that if the condenser is used in a normal manner (not flooded), and the reflux drum level is controlled by the reflux flow, then create a new member of the class *Loops* named *LEVREF*, and give the value of *levref* to the attribute *Description* of the member *LC reflux* of the class *Material Balance Controllers*. The member of *Loops* is later written to a file which is read by an external FORTRAN executable which draws the graphics. The file named *LEVREF* contains information to draw the control loop which connects

the reflux accumulator level controller to the reflux flow controller. The second consequent assigns the string attribute *Description* a value of *levref*, which is actually a previously defined string constant describing the reason that the reflux flow was chosen to control the reflux drum level. The value of *Description* is used as part of the description of the control system.

The fifth group of rules describes relationships among user provided information. The purpose of these rules is to avoid asking the user to provide redundant information, or information which can cause conflicts. The logic contained in the rules of this group make them particularly difficult to compose. The shell is backward chaining, and the rules are more forward chaining in purpose. The following is an example of the rules from this group.

```
Condensate temperature not controlled for vapor product:
if
    Top Product = vapor
then
    Condensate Temperature = is not controlled.
endif.
```

Controlling the condensate temperature is a possible lowest level subgoal. If the distillate product is in the vapor form, then any inerts leave the system with the vapor product. Since controlling the holdup of inerts in the column is necessary in order to control the condensate temperature, the condensate temperature cannot be controlled on a distillation column with a vapor distillate. This rule specifies that if the distillate product is in the vapor form, then the condensate temperature is not to be controlled.

4.5.5. DICODE Demons

DICODE uses demons for three purposes. The first is to provide warnings and recommendations, the second is to prompt the user for information when the rules are unable to determine it. These two purposes are relatively simple to implement and understand. The third use of demons is to keep track of which success paths (manipulated variables) have been used, and which are still available for satisfying lowest level subgoals (control objectives).

The first group of demons provides information to the user which informs him about possible poor choices of control objectives and equipment design. They also give precautions and recommendations when poor choices are made and not changed by the user. The following demon is used to present a warning about controlling condensate temperature on an atmospheric column.

```
Warning about controlling Condensate temp on
atmospheric columns:
when
     Operating Pressure = atmospheric and
     Condensate Temperature = is controlled and
     determined (Condensate Temperature Demon) = false
then
     Condensate Temperature Demon = completed.
     display attach warning of kb.
     askfor submenu1.
     if
          submenu1 = pressure
     then
          erase Operating Pressure.
          askfor Operating Pressure.
     endif.
     if
```

```
        submenu1 = temperature
    then
        erase Condensate Temperature.
        askfor Condensate Temperature.
    endif.
    erase submenu1.
    if
        Operating Pressure = atmospheric and
        Condensate Temperature = is controlled
    then
        display attach precautions of kb.
    endif.
endwhen.
```

The demon behaves in the following way. When the user specifies that he would like to control the condensate temperature on an atmospheric column and the demon has not already fired (the determined(...) = false), the user is presented with a warning called *attach warning of kb* which states that he has chosen a poor combination of control objectives for the column under consideration. Next, he is asked to choose from three options (asked by the *askfor submenu1* command). Based on his response, he then may choose to operate the column under a slight pressure or vacuum, or he may choose not to control the condensate temperature. If the user decides to keep the options he had previously chosen, the *display attach precautions of kb* command presents the user with precautions which should be taken when the condensate temperature is controlled on an atmospheric column.

The next group of demons asks the user to provide information about the system when it cannot be inferred from rules. The KES^{TM} shell tries to

determine the value of an attribute by backward chaining. If no rules exist which could determine the value, then the user is asked to provide a value. If rules do exist which could determine the value, but none succeed, then the attribute is assigned a value of *unknown*. If the information is necessary, then a demon must be provided to ensure that a value is obtained. The situation when an attribute is assigned a value of *unknown* occurs in DICODE due to the rules which represent the relationship among user input. These rules ensure that inconsistent or redundant information is not entered by the user, but they also cause some necessary information to become unknown if none of the rules succeed. The demons erase the attribute value (resetting it to *undetermined*) and then ask the user to provide a value. The following demon is an example taken from the DICODE knowledge base.

```
Get heating medium:
when
     status(Heating Medium) = unknown
then
     erase Heating Medium.
     askfor Heating Medium.
endwhen.
```

The rule above states that when the value of *Heating Medium* becomes *unknown*, reset its value to *undetermined* and ask the user to provide a value.

When DICODE begins to try to satisfy the goals, it establishes the lowest level subgoals as ones to satisfy. These subgoals are satisfied by success paths which can be common to multiple subgoals. Since each success path represents a single manipulated variable, each success path can satisfy only one lowest level

subgoal by being assigned to a controlled variable. The rules which establish the pairing of manipulated and controlled variables must therefore check to make sure that the controlled variable is currently a subgoal to be satisfied, and that the manipulated variable is available for use (not yet paired to a controlled variable) before it assigns a pairing. The third group of demons is used to keep track of which manipulated and controlled variables have been paired. The following is one of the two demons used in DICODE to keep track of which manipulated and controlled variables have been paired. The other demon deals with a special case which could not be dealt with by the more general demon.

```
Demon:
c:Controllers
when
    status(c>manipulated) = known
then
    forall mv:Manipulated Variables do
        if
            c>manipulated = mv>manipulated
        then
            erase mv>paired.
            mv>paired = true.
        endif.
    endforall.
    forall cv:Controlled Variables do
        if
            c>controlled = cv>controlled
        then
            erase cv>paired.
            cv>paired = true.
        endif.
    endforall.
endwhen.
```

This demon is quite complicated and requires some explanation. Immediately under the name of the demon (*Demon*), the line *c:Controllers* binds the variable c to all *Controllers*. The demon is then executed as many times as there are *Controllers*. In the antecedent, any time the value of the attribute *manipulated* of any controller becomes *known*, two actions occur in the consequent. First, KES^{TM} loops through all members of the class *Manipulated Variables* (each being bound to the variable *mv*). When it finds the member of *Manipulated Variables* whose attribute *manipulated* equals the value of the attribute *manipulated* of the member of the class *Controllers* bound to *c*, it changes the value of the attribute *paired* of *mv* to *true*, indicating that the manipulated variable has been used and is no longer available. The second action is completely analogous, changing the attribute *paired* of the correct member of *Controlled Variables* to true. The utility of this demon will be explained in section 4.5.7, at which time its necessity should become apparent.

4.5.6. DICODE Actions

When a KES^{TM} expert system is run, execution begins at the top of the actions section. In the actions section, algorithmic programming guides execution of the consultation. In this section, goals are created through the use of commands. Attributes can be assigned values, erased, class members can be created and destroyed, messages can be displayed to the user, information can

be written to files, information can be read from files, external programs can be run, etc. The actions section allows the knowledge base author to tailor execution of the expert system to his own preference through the use of algorithmic programming.

In DICODE, the actions sections is used primarily to allow the user to choose options during a consultation. The options allow the user to choose what he would like to do next, such as start a consultation, quit, review previous results, see a graphical representation of the control system and equipment design, or see a text description of the control and equipment design. These options are presented to the user through the use of menus. Menus are actually attributes whose values are asked of the user and then used to determine what actions are to be taken by the expert system. The following is an example of a small section of the actions taken from DICODE.

```
while resultmenu # return do
if
     resultmenu = graphics
then
     message " PLEASE WAIT".
     write "CONSYS.DATA", Drawings.
     run graph.
     .
     .
     .
endif.
if
     resultmenu = save graphics
then
     erase Filename.
     askfor Filename.
```

```
        write Filename, Drawings.
    endif.
    .
    .
    .
    endwhile.
```

The above part of the actions section allows the user to choose options of seeing a graphical representation of the control system design by choosing the value *graphics* for the attribute *resultmenu*, or save the graphics results by choosing the value of *save graphics* for the attribute *resultmenu*. The part shown loops until the value of the attribute *resultmenu* becomes *return*. Then the loop is exited and execution continues to other actions. The section shown above has been abbreviated, but comes directly from the actions section in DICODE.

4.5.7. DICODE Execution

DICODE execution begins by presenting the user with a menu of three options. If the user selects the option to create a new design, then a command in the actions section sets up the the goal to obtain the *Status* of the root goal *Best Control System*. KES™ then backward chains down the goal tree until a lowest level subgoal is found. Since the lowest level subgoals can be satisfied based on equipment and process design information, DICODE asks the user for information about the equipment and process. Rules which describe the relationship between user provided information are used to determine the

necessity of certain lowest level subgoals. If the rules do not succeed, then the demons ask the user for the needed information.

In general, the lowest level subgoals are satisfied by controllers. The rule presented above named *Accumulator level subgoal satisfied by internal or control* typifies the rules which satisfy the lowest level subgoals. Note that the antecedent of the rule continues the backward chaining process to try to determine the value of the attribute *manipulated* of the controller. Backward chaining then finally ends at rules which contain the conditions which specify when a certain manipulated variable (or success path) should be used to control the controlled variable (represented by the lowest level subgoal) and thus satisfy the lowest level subgoal. The rules which determine the pairing of manipulated and controlled variables are the ones which reflect the knowledge specific to the expert used as the knowledge source. The same goal tree could be used to design the controls for the distillation column, but the rules such as the one mentioned above would be different for different experts.

Once the best manipulated variable has been determined for the controlled variable, the demon which monitors the attribute *manipulated* of the members of the class *Controllers* fires. The demon changes the attribute *paired* from its default value of *false* to *true*. In the case of the rule mentioned above, the demon changes the attribute *paired* of the manipulated variable *reflux* to *true*. Any other rule subsequently examined for another controller to use the manipulated variable *reflux* will not succeed if the antecedent checks checks the

attribute *paired* for a value of *false*. In the manner described, DICODE ensures that any manipulated variable can be paired with only one controlled variable, and thus each success path satisfies only one lowest level subgoal.

At this point, it may not be clear why the demon which changes the value of the attribute *paired* to *false* needs to be implemented as a demon and not a rule. For instance, why not have the rule which pairs a manipulated and controlled variable change the value of the attribute *paired* to *true*? The answer is that in KES^{TM}, the inference engine uses short cut evaluation. That is, it obtains values for attributes which are necessary, so not all of the assignments in the consequent of a rule are executed unless that assignment is necessary to obtain the value of the attribute which is being sought. Adding an assignment to the consequent of a rule which is being fired does not guarantee that the assignment will occur. Therefore, a demon which monitors the attribute which caused the rule to be evaluated fires each time the attribute changes to fulfill the antecedent of the demon.

Another reason that the demon must be implemented as a demon and not as a rule is that if it were in a rule, assigning the attribute *paired* a value would cause circular inferencing in an infinite loop. Consider the following example. The rule which gives the value *reflux* to the attribute *manipulated* of the controller requires the value of the attribute *paired* of the manipulated variable *reflux*. Therefore, the value of the attribute *paired* is sought by the inference engine. If the demon were a rule, it might look like the following

hypothetical rule.

```
Hypothetical rule:
if
    Material Balance Controllers:LC reflux>manipulated
        = reflux
then
    Manipulated Variables:reflux>paired = true.
endif.
```

The rule above would be executed when the value of the attribute *paired* of the manipulated variable *reflux* is sought. Its antecedent needs to know the value of the attribute *manipulated* of the controller *LC reflux* which can be determined by the previous rule. Each rule referring to the other involves circular logic and such logic will cause severe problems during expert system execution. The demon solves the circular logic problem. Furthermore, since no rules assign values to the attributes *paired* of the manipulated variables, they receive the default value of *false* provided in the definition of the frame *Manipulated Variables*. The same circular logic problem is encountered if the attribute *paired* of the class *Controlled Variables* is referenced in the antecedent of a rule. The demon which assigns values to *paired* of the manipulated variables, does the same for *Controlled Variables*.

Another feature of DICODE is the priority of the lowest level subgoals. In the design of control systems for distillation columns, different control experts assign different priorities to the control loop determination. P.S. Buckley is the expert whose knowledge is used in DICODE. His design methodology gives

priority to the regulatory control design in the following order, from highest to lowest:

1 Material Balance Control
  a Pressure Control
  b Reflux Accumulator Level Control
  c Column Base Level Control
2 Composition Control
  a Distillate Composition Control
  b Bottom Composition Control
3 Condensate Temperature Control

DICODE implements this priority through ordering the goals in rules appropriately. KES^{TM} attempts to satisfy the antecedents of a rule in the order in which they appear. The following three rules ensure that the priority given above is implemented.

```
Regulatory control goal satisfied:
if
     Subsubgoals:Best Material Balance>Status = satisfied and
     Subsubgoals:Best Composition Controls>Status
          = satisfied and
     Subsubgoals:Best Condensate Temperature>Status
          = satisfied
then
     Subgoals:Best Regulatory Controls>Status = satisfied.
endif.
```

```
Material balance control subgoal satisfied:
if
     Subsubsubgoals:Best Pressure Control>Status
          = satisfied and
     Subsubsubgoals:Best Accumulator Level>Status
          = satisfied and
     Subsubsubgoals:Best Bottom Level>Status = satisfied
```

```
then
    Subsubgoals:Best Material Balance>Status = satisfied.
endif.


Composition control subgoal satisfied:
if
    Subsubsubgoals:Best Distillate Composition>Status
        = satisfied and
    Subsubsubgoals:Best Bottom Composition>Status
        = satisfied
then
    Subsubgoals:Best Composition Controls>Status = satisfied.
endif.
```

Since KES^{TM} attempts to establish the antecedents in the order they appear, the material balance control is given priority over composition and condensate temperature. The next rule gives priority to pressure control, then accumulator level control, and then column base level control. The next rule gives priority to distillate composition over bottom composition, although priority here is somewhat arbitrary since the choice of manipulated variables is usually mutually exclusive between the composition controllers.

Ultimately, the primary result of the execution of backward chaining is that the root goal *Best Control System* is satisfied. This single result is typical of the backward chaining paradigm. The important point for DICODE is that as a side effect, all of the necessary and recommended controllers are established during the process of obtaining the root goal satisfaction. Furthermore, the members of classes which are used for the graphical representation of the design are also obtained. The text descriptions which are another result have actually

not been obtained. However, all rules which determine the proper text results have been executed in order to obtain other information. The KES^{TM} shell does not execute the rules again in order to obtain the description when it is needed, rather the shell knows which rules have succeeded and it simply assigns values to attributes according to the consequents of rules which have already succeeded.

At this point in DICODE execution, all information in the form of the design results has been obtained, or at least all rules which can determine the proper results have succeeded. The user is presented with a menu which allows him to select the type of results he would like to see, or to return and start a new design. DICODE allows the user to select a quick design result which includes only the level and composition controls without specifics of the equipment design. The preliminary result will not be completely accurate since it cannot consider equipment design variations. Another option is to examine the overhead or reboiler section designs in detail. These two options present the control and equipment design in detail with all override and cascade controls.

4.6. Insights Gained Through DICODE

The real test of any methodology is whether it works in reality. The objective was to provide a method by which the knowledge in control design can be translated from expert to knowledge base through the use of an intermediate

representation. The method was to be independent of the shell used for implementation but provide information on what features are necessary in the shell. The methodology which is proposed in this thesis and demonstrated through the implementation of DICODE was found to clarify the knowledge base, simplify writing the knowledge base, and hopefully help ease the process of modification of the knowledge base through addition of rules. The methodology also delineated the requirements of the shell in terms of the fact that access oriented programming was found necessary.

The next chapter describes the methodology in detail, and discusses the general aspects of the methodology. The impact of the use of the GTST model as the intermediate knowledge representation before expert system construction is described as well as major simplifications which could be made to the knowledge base through the use of certain expert system shell provided features. A generalization and categorization of the rules needed to implement the knowledge based on the GTST model is presented.

For those interested in the course of development of DICODE, Appendix B contains a history of the project. Also, Appendix C is a users guide which describes the details of using and tuning DICODE in order to tailor its use of the RGA, RDG, relative stream sizes, and the estimate of integrated error in

ways other than the two defaults provided.

CHAPTER 5

KNOWLEDGE BASE STRUCTURING

Chapter 3 introduced the concept of the Goal Tree – Success Tree (GTST) model for deep knowledge. The example presented in Chapter 3 illustrates the method for constructing a goal tree for the design of control systems for a distillation column. Chapter 4 describes the application of that GTST to implementation of a real expert system for designing the controls for a distillation column. Using the GTST model as an intermediate form of knowledge representation before entering that knowledge into the knowledge base was found to have distinct advantages over direct translation of knowledge from source to expert system. This chapter discusses the method for using the GTST as a guide to constructing the knowledge base for an expert system in an attempt to draw some generalizations from the experience gained through construction of the knowledge base described in Chapter 4. The advantages of using the GTST as an intermediate and the requirements the method imposes on the expert system shell are discussed.

5.1. Knowledge Organization in a GTST

The organization of the GTST forces the knowledge base author to gain a better understanding of the problem solution method in order to represent the knowledge in a GTST. A better understanding of the problem solution leads to a knowledge base which is more structured, efficient, and understandable. Furthermore, if the GTST structure is actually used in the knowledge base, it provides direct implementation of troubleshooting and debugging facilities for use during expert system development. The GTST model leads to representation of the knowledge in the form of frames.

If the GTST is directly represented in the knowledge base of the expert system, the goals can be represented as instances of frames. That is, each goal or subgoal is a child of a generic or parent object whose attributes have been defined. Each success path below any lowest level subgoal may also be represented as an instance of a frame. The goals and success paths must have an attribute to indicate that they have either been satisfied or not satisfied. The goals are linked by rules which state that if all subgoals of a particular goal have been satisfied, then the goal has been satisfied. The success paths are also linked to their respective lowest level subgoals by rules which state that if any success path is satisfied, then that lowest level subgoal is satisfied. A further discussion of the rule implementation appears in section 5.2.

An advantage to representing the GTST directly in the knowledge base is that it lends its own structure to the knowledge base. The hierarchical structure

of the GTST outlines the knowledge in a way which illustrates the importance and significance of each goal and success path. Goal priority is represented by position in the goal tree, and the necessity of its satisfaction is given by the rules. The knowledge base will be much more organized with the presence of the GTST, and therefore, the knowledge base will be more easily understood by anyone who wishes to modify the knowledge.

Another advantage of direct representation of the GTST structure in the knowledge base is that facets may be used on the slots of frames to inform the user when goals have not been satisfied, indicating that a bug is present in the knowledge. For instance, if a facet is attached to the attribute *status* of a *goal*, then when *status* is changed, the *when changed* facet will be executed. The procedure could check the value of *status* and if it is *not satisfied*, then a search for the cause could be executed. Access oriented programming is necessary to implement this trouble shooting procedure.

Probably the most useful advantage of the GTST implementation in the knowledge base is the fact that it is expert independent. The GTST does not include the knowledge by which the lowest level subgoals are satisfied for a given set of input data. That is, the GTST describes the *what* part of the problem solution or the objectives (goals) for solving the problem. The GTST for designing the controls for a reactor system would be different from the GTST described in the previous chapters for designing the controls for a distillation column. However, the basic breakdown of the design or root goal is the same.

There will exist many similarities between the two GTST's.

The fact that the GTST is independent of the expert is the result of the fact that the GTST models the process related or deep knowledge involved in the problem solution. This deep knowledge is common to all experts but depends on the problem domain. The end node implementation of the goal tree (selection of an appropriate success path to satisfy a given lowest level subgoal) is the part of the knowledge base which is expert dependent. This knowledge describes the *how* part of the problem solution. It is difficult to draw specific generalizations as to a recommended implementation structure for the knowledge of this type. However, a few guidelines can be given as to the form of the rules, frames, and procedures used to implement the knowledge which is expert dependent. Section 5.3 describes the recommended structure to use in the knowledge base in order to conform to the GTST model. It also provides some guidelines as to the form of the rules, frames, and procedures used to implement the expert dependent knowledge.

5.2. Process Specific Knowledge Representation

Aside from the knowledge (rules, frames, attributes, demons) used to represent the GTST model, other knowledge which is domain specific must be represented within the expert system. This knowledge represents the concepts related to the physical objects, concepts manipulated as physical objects during

problem solving, information used as inputs to the problem solution, any relationships existing between inputs, and appropriate precautions necessary under specific conditions (combinations of input data and solution objectives).

5.2.1. Frames

Any concepts which are manipulated during the problem solution should be represented as frames within the knowledge base. For the control design problem, some of the equipment should be represented as frames. If multiple pieces of very similar equipment are considered, then creating a frame to represent that type of equipment facilitates knowledge base construction when rules must be written. For example, if the equipment is a heat exchanger network, then a frame representing heat exchangers will make writing rules easier if they apply to several of the exchangers. It is worth noting that it is generally not desirable to create a frame which will only have one instance. It is usually easier to create global or free attributes to represent the attributes associated with the one instance of the frame. Using global attributes to describe the single instance not only could reduce the memory required by the system, it also simplifies the rules.

As an example, consider an expert system which deals with a single heat exchanger. One could create a frame whose name is *Heat Exchangers* with attributes of *Cold-Side Outlet Temperature*, *Cold-Side Inlet Temperature*, etc.

Then during execution, a single instance of *Heat Exchangers* named perhaps *Heat Exchanger-1* could be created. Rules which deal with *Heat Exchangers* would be used to infer the result(s) of the consultation. The rules could look like the following.

> High cold side inlet temperature:
> If the Cold-Side Inlet Temperature of Heat Exchanger-1 is greater than 100
> then Problem is Warm Coolant.

Or equivalently,

> High cold side inlet temperature:
> If the Cold-Side Inlet Temperature of any Heat Exchanger is greater than 100
> then Problem is Warm Coolant.

Note that the antecedent must refer not only to the cold side inlet temperature, but also specifically the cold side inlet of a heat exchanger. Rather than using this kind of representation, it would be better to simply use a global attribute called *Cold-Side Inlet Temperature* realizing that this is the cold side inlet temperature of the only heat exchanger being considered. The rule would then be simpler.

> High cold side inlet temperature:
> If the Cold-Side Inlet Temperature is greater than 100
> then Problem is Warm Coolant.

Using this representation not only simplifies the rule, it reduces the representation structure by eliminating the frame *Heat Exchangers*. It may not

always be advantageous to use this second representation, particularly if the shell being used does not allow the use of global or free attributes. There may also be other reasons for using a frame representation for a single instance such as clarity of the knowledge base depending on the shell used.

Other concepts should also be represented as frames within the expert system. For the case of control system design, the controllers should be represented as frames. Today, the controller is usually implemented as a computer program, but the control design engineer thinks of a controller as a physical object. Using the frame representation in this case is an extremely powerful implementation since it is possible that many instances of the frame *Controller* will undoubtedly be created during a single consultation. Furthermore, the controllers have properties by which they can be classified for purposes of control system design. For instance, composition controllers are typically slow. It is therefor desirable to cascade the composition controllers to flow controllers if a flow is being manipulated. Using flow controllers in this way is almost always desirable for any controller which is slow compared to a flow controller.

In this case, a frame called *Controller* should be created with attributes *speed* and *output* (other attributes would generally be attached as well). Then a rule can be written which deals with all *Controllers* based on their speed. For example, the following rule may be useful.

Cascade slow controller to distillate flow controller:
If the speed of a Controller is slow and
the output of the Controller is distillate flow,
then create instance Distillate Flow Controller of Controllers

and the output of Distillate Flow Controller is distillate valve.

It may actually be possible to make the rule even more general by not specifying the output of *Controller* but by creating the other controller based on the output of the first controller. That is, in the rule above, *output* of *Distillate Flow Controller* could be assigned the value *distillate valve* through the association of *distillated valve* with *distillate flow*. The ability to create a new controller based on the output of the first is a feature which is definitely shell dependent, and may or may not be useful. In any case it is not a critically important feature. However, it is definitely useful to be able to write general rules which function like the one above.

It should be mentioned that sometimes the frames actually contain the rules which determine the behavior of the frame. Rules contained withing the definition of frames is an undesirable limitation which constrains the use of rules drastically. The constraint may not be severe if the frames are allowed to communicate freely, however it is nonetheless limiting.

5.2.2. Attributes

There are usually many attributes which need not be attached to or associated with objects or frames. For instance, input data may be associated with what could be considered the problem environment. Some shells allow use of these global or free attributes, others do not. Examples of attributes of this

kind include such descriptions as the amount of inerts in the feed to a distillation column or the operating pressure of the process. Note that these attributes could be associated with a frame, although it is not necessary and not usually desirable.

5.2.3. Rules

A typical use of rules which deal specifically with the problem to be solved is to describe relationships between the various input data. Certain characteristics of a problem often infer other characteristics. In order to ensure that inconsistent data is not provided to the system, rules can be used to express the relationship between the input data. An example of this kind of relationship is the fact that if very little inerts is present in the feed to a distillation column, it will not be possible to control the temperature of the condensate leaving the condenser. The fact that no inerts are in the feed means that an inert bleed is not used, and therefor there are not enough manipulated variables available to control the condensate temperature (unless it is desirable to introduce inerts into the process).

Some care must be taken in using rules which describe the relationship between inputs in an expert system. It is possible and quite easy to create circular logic paths with rules which represent relationships among the input values. It also may be necessary to implement some of these relationships as

demons rather than rules in order to ensure that the value is not asked of the user, or to ensure that circular logic paths are not created.

The remainder of this chapter describes the recommended methodology to be used to create a knowledge base for the purpose of solving a design problem using the GTST model. Specific steps are outlined to create the knowledge base which contains the GTST model in the form of frames and rules. The need for demons in specific cases is discussed and finally the requirements imposed on the shell by the problem as modeled in the GTST are presented.

5.3. Knowledge Base Building Methodology

The methodology for building the knowledge base for an expert system based on the GTST model requires a backward chaining shell. The fact that the method is intended to be used to create expert systems for backward chaining shells has the advantage that if the problem solution can be modeled in a GTST, then it can be solved by a backward chaining shell. This is true for any problem, including the design problem which has been typically solved only in forward chaining shells. The following discussion assumes that the method will be used to create an expert system in a primarily backward chaining shell.

5.3.1. Building the GTST

5.3.3. Rules

Three distinctly different types of rules are necessary to implement the GTST in the knowledge base of an expert system. These rules have the function of linking the goals in the proper way, linking the success paths with lowest level subgoals, and satisfaction of the success paths. Each type of rule is presented with possible generalization based on the capabilities of the shell in which the knowledge base is implemented.

5.3.3.1. Type 1 Rules

The first type of rule links the subgoals of a particular goal to their goal. For the goal tree shown in Figure 5.1 below, the rule takes the form:

Rule 1:
If the Status of G2 is satisfied and
the Status of G3 is satisfied and
the Status of G4 is satisfied
then the Status of G1 is satisfied.

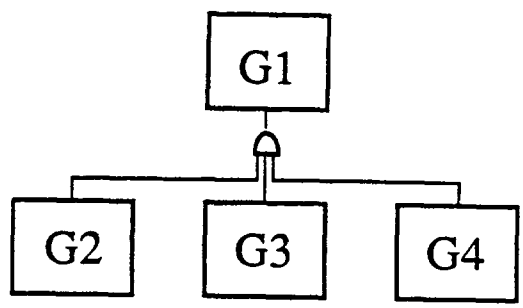


Figure 5.1. Example goal tree.

Equivalently, the rule could deal with non-satisfaction of the goals. In this case, the rule takes the form:

> Rule 2:
> If the Status of G2 is not satisfied or
> the Status of G3 is not satisfied or
> the Status of G4 is not satisfied
> then the Status of G1 is not satisfied.

The second form is more useful for using the *has a* relationship and scanning through the instances of a frame. For example, the rule could take the form:

> Rule 3:
> If the Status of any Goal of G1 is not satisfied
> then the Status of G1 is not satisfied.

In this case, *Goal* is the name of the parent frame and *of G1* indicates that only the goals which are *parts* of goal G1 are considered. This rule still represents only a single goal (G1) and if all rules were written in this manner, then there would be as many rules as there are goals (excluding the lowest level subgoals). Further generalization may be possible by not specifying G1, but using the name of the parent frame. In this case, the rule could take the following form:

> Rule 4:
> If the Status of any Goal2 of Goal1 is not satisfied
> then the Status of Goal1 is not satisfied.

In this case, a distinction must be made between the two goals so that the satisfaction of the higher level goal is based on the satisfaction of its lower level goals. In order to implement this distinction, it may be necessary to establish

Constructing the knowledge base for an expert system using the GTST model as an intermediate representation of the knowledge starts by constructing the goal tree. Section 3.2 discusses the goal tree construction. Once it has been determined that the goal tree is complete and accurate, the success paths which satisfy the lowest level subgoals must be determined. Typically the success paths reference equipment, procedures, measurements, etc. Once the GTST is completed, construction of the knowledge base can begin. Objects or frames must be defined to represent all the goals and success paths with necessary attributes. Finally, rules must be written to link the success paths to their subgoals, and to link the subgoals to their goals. The next sections discuss each step of the conversion of the GTST to a knowledge base in more detail.

5.3.2. Defining Goals and Success Paths as Frames

Each goal is represented as an instance of a generic frame named *goals* with an attribute which will take one of two values indicating either satisfied or not satisfied. The values could be truth values (*true* or *false*) or could be other values indicating the satisfaction of a goal. The attribute values allowed is a function of the shell in which the expert system is implemented. It is suggested that one use truth values for an attribute named *satisfied* if possible or an attribute named *status* taking a value of satisfied or unsatisfied. The syntax of the shell should allow the use of descriptive attribute names and values. If

the knowledge base author takes advantage of descriptive attribute names and values, the rules will be more understandable and the amount of documentation needed within the knowledge base will be decreased.

The *has a* relationship between frames is useful for implementing rules which are general. The *has a* feature describes the relationship between objects in which one is a part of the other. For instance, in the goal tree structure, a *goal* has *subgoals*. In the definition of *goal*, it is specified that *goal1 has a subgoal1*. If the shell provides features which allow the frames or objects to be linked by the *has a* relationship, the goals should be linked in the following manner. Each goal or subgoal should have as *parts* all of its subgoals. Each lowest level subgoal should have as *parts* all of its success paths. This relationship among the goals provides a direct implementation of the goal tree using shell provided features. If the *has a* relationship is not provided, it can be implemented using attributes which describe the relationship. The *has a* feature is used in the rules which link the goals to establish satisfaction. One rule may then be used to link all goals to their subgoals by referencing all goals and checking for satisfaction of those subgoals which are its *parts*. If the goal tree is large, this one rule could replace tens or hundreds of more specific rules. In the worst case, one rule must be written for each goal, specifying that if all its subgoals are satisfied, then the goal is satisfied. The next section describes the rules and gives examples which demonstrate the use of the *has a* relationship.

a hierarchy of goals by creating a parent frame for each level of subgoals. For example, consider a frame base in which the parent frames *Goals* and *Subgoals* indicate two levels of goals in the goal tree. The rule could take the following form:

> Rule 5:
> If the Status of any Subgoal of any Goal is not satisfied
> then the Status of Goal is not satisfied.

With any of the rules which assign a value of *not satisfied* to the attribute *Status*, it may be necessary to provide a default value of *satisfied* for *Status* when no rules assign it a value. The assignment of a default value of *not satisfied* will force a goal to be unsatisfied when rules cannot establish satisfaction. Otherwise, depending on the particular shell which is used, the value of *Status* could become *unknown* rather than *not satisfied.*

5.3.3.2. Type 2 Rules

The second type of rule links the success path to its subgoal. These rules satisfy the lowest level subgoals when any of its success paths are satisfied. Figure 5.2 shows a lowest level subgoal and its associated success paths. The rules which satisfy the lowest level subgoal take the following form:

> Rule 6:
> If the Status of S1 is satisfied or
> If the Status of S2 is satisfied or
> If the Status of S3 is satisfied
> then the Status of G1 is satisfied.

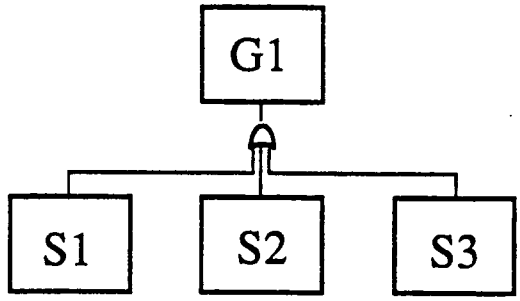


Figure 5.2. A lowest level subgoal and its success paths.

Equivalently, if S1, S2, and S3 are specified as *parts* of goal G1, then the following rule may be used.

> Rule 7:
> If the Status of any Success Path of G1 is satisfied
> then the Status of G1 is satisfied.

Further generalization results in the following rule.

> Rule 8:
> If the Status of any Success Path of any Goal is satisfied
> then the Status of the Goal is satisfied.

5.3.3.3. Type 3 Rules

The third and final type of rule establishes the satisfaction of the success paths. These rules contain the knowledge which deals with specifics of the problem solution, while the first two types of rules deal more with the solution

method. These success path rules contain attributes and objects in the antecedent which are specific to the application. The following rule is an example which illustrates the use of the third type of rule. In this example, the success path *S1* consists of *Pump1* being *available*, and *Valve1* being *closed.*

Rule 9:
If the Status of Pump1 is available and
the Position of Valve1 is closed
then the Status of S1 is satisfied.

Rules such as the one above establish the success of a particular success path which lies below a single lowest level subgoal. The situation arises where a success path lies below more than one lowest level subgoal. In the case where this occurs and a success path is allowed to satisfy only a single lowest level subgoal, an accounting must be used to keep track of the availability of each success path. When a success path is *bound* to a goal, an attribute of the success path called perhaps *availability* should be given a value to indicate its unavailability. This is best implemented using a demon. The demon would look like

Demon 1:
When the Status of S1 is satisfied
then the availability of S1 is false.

When this demon is used, the rules which satisfy success paths must contain a reference to the availability of the success path. The rule above (rule 9) would then be

Rule 10:

If the availability of S1 is true and
the Status of Pump1 is available and
the Position of Valve1 is closed
then the Status of S1 is satisfied.

It is important to realize that the demon used above could not be used as a rule for backward chaining because it causes circular logic. When backward chaining indicates that the value of *Status* of *S1* is needed, rule 10 indicates that the *availability* of *S1* must be obtained. The demon (if it were a rule) would be used to try to determine the *availability* of *S1* based on the *Status* of *S1*. Since the *Status* of *S1* is the attribute being sought, a circular path is established, causing inference problems. The demon (used as a demon) cuts the backward chaining inference process by immediately executing as soon as a rule determines the value of the *Status* of *S1*, giving the *availability* of *S1* a value of *false*. It is also necessary to realize that the *availability* of all success paths must be set to *true* at the beginning of inferencing, and the demon must have the ability to reset the value of *availability* since *availability* will already have a value before execution of the demon.

5.3.3.4. Type 2a Rules

Another consideration when success paths can satisfy multiple lowest level subgoals is that it is usually important to know which success path satisfied which subgoal, not just that the subgoal was satisfied. In this case, it is necessary to also keep track of how each lowest level subgoal was satisfied. Keeping

track of which success path satisfied which lowest level subgoal can be accomplished by assigning a value to an attribute of the subgoal which indicates which success path was used to satisfy it. The attribute should be assigned a value in the same rule which satisfies the subgoal. If we use an attribute called *success*, then rule 10 above would be

> Rule 11:
> If the Status of any Success Path of any Goal is satisfied
> then the Status of the Goal is satisfied and
> the success of the Goal is the name of the Success Path.

Note that this implementation also requires the ability to equate the values of attributes of frames. The last line of the rule establishes the connection between the success path and the lowest level subgoal. Each *Success Path* must have an attribute *name* whose possible values are the same as the possible values of the attribute *success* of the frame *Goal.* If it is not possible to equate attribute values in the manner shown above, then a separate rule for each success path link to each lowest level subgoal must be written in which the attribute *success* is given a value indicating which success path was used.

5.3.4. Demons

Only one type of demon is actually needed in order to implement the GTST within the knowledge base of an expert system. The purpose of this demon is to keep an accounting of the success path availability during the process of

determining which success paths should be used to satisfy which lowest level subgoals. Other than the use of demons for this purpose, three other functions are useful for the design problem. These demons have the function of providing appropriate warnings concerning poor combinations of user input, providing conflict resolution for cases where the user would like to change input based on the warnings, and prompting the user for needed information when it cannot be inferred. These four types of demons are described in the next four sections.

5.3.4.1. Type 1 Demons

The type 1 demons provide an accounting of the availability of the success paths during inferencing. These demons were described above as part of the rules section in order to clarify the explanation of the rules. Note that this demon is necessary only if the success paths appear under more than one lowest level subgoal, but may be used to satisfy only one of the lowest level subgoals. In this case, it is also necessary to somehow specify the order in which the lowest level subgoals must be satisfied. Demon 1 above gives the form of the type 1 demons.

In comparing Demon 1 with the demons used in the DICODE knowledge base, you will note a striking contrast. Demon 1 is a complete generalization of the concept of the demon entitled *Demon* in the DICODE knowledge base. That demon is repeated here.

Demon:

```
c:Controllers
when
    status(c>manipulated) = known
then
    forall mv:Manipulated Variables do
        if
            c>manipulated = mv>manipulated
        then
            erase mv>paired.
            mv>paired = true.
        endif.
    endforall.
    forall cv:Controlled Variables do
        if
            c>controlled = cv>controlled
        then
            erase cv>paired.
            cv>paired = true.
        endif.
    endforall.
endwhen.
```

In DICODE, the value of *availability* as *false* is expressed as the attribute *paired* of *Manipulated Variable* as *true*. Since the success paths of the DICODE knowledge base are represented by manipulated variables, and the success of a lowest level subgoal is expressed as a controller linking the lowest level subgoal (controlled variable) to a success path (manipulated variable), the demon *Demon* serves the purpose of Demon 1, but also keeps an accounting of which lowest level subgoals (controlled variables) have been paired.

In DICODE it is useful to refer to the controlled variables in the antecedents of type 3 rules to ensure that rules are not used unnecessarily when a control objective has been met by the inability to provide control. An example of the

situation when a control objective becomes inactive due to inability to provide control is when condensate temperature cannot be controlled due to the lack of enough manipulated variables. It is therefore necessary to keep an accounting of the availability of the controlled variables (that is, whether a control objective is active).

5.3.4.2. Type 2 Demons

The type 2 demons provide appropriate warnings when poor choices of input values are provided. The demons of this type are not necessary for the solution to the problem, but could be specified (as they were for the DICODE project) as a criterion for the expert system. During the design process, it is generally possible to provide combinations of design objectives and other user provided information which create circumstances under which the design solution will be poor.

The type 2 demons trap these combinations of user input in order to warn the user that a poor combination has been provided. A good example of the type 2 demons can be taken from the DICODE knowledge base. The choice to control the condensate temperature on a distillation column which is operated at atmospheric pressure is a poor one. Since pressure is not controlled, the coolant stream is generally available for controlling the condensate temperature. However, controlling condensate temperature with the coolant stream causes dynamic problems which can adversely affect operation of the column.

A demon is used to warn the user of this specific poor combination of operating conditions and control objectives. The following is a translation of the demon from DICODE.

> When the operating pressure is atmospheric and
> the condensate temperature is controlled
> then display warning to the user.

In this case, *warning* is a message indicating that the user has chosen a poor combination of control objectives and operating conditions.

5.3.4.3. Type 3 Demons

The type 3 demons take the warnings of the type 2 demons one step further. The type 3 demons allow the user to change the choice of inputs in order to eliminate the poor combination described by the warning. In the case of the example above, the demon asks the user which input he would like to change, erases the appropriate attribute value, and then asks the user to provide a value. In the cases handled by this type of demon, the user should always be allowed to keep the current options keeping in mind that the solution will poorer than if he had changed options.

Type 3 demons have the following form.

> When the operating pressure is atmospheric and
> the condensate temperature is controlled
> then
> if conflict resolution is change operating pressure
> then erase operating pressure and ask for

operating pressure
if conflict resolution is change condensate temperature
then erase condensate temperature and ask for
condensate temperature.

The type 3 demons can also be used to force the user to choose different options when inconsistent input is provided. In this way, the demons can be used to resolve conflicts when they arise.

5.3.4.4. Type 4 Demons

The last function of the demons is necessary when the shell functions in a specific manner concerning the source of values for attributes. The shell used for DICODE necessitated the use of demons to ask the user for input in certain cases. In DICODE, rules are used to express relationships between possible user provided information. For example, when no inerts are present in the column feed, condensate temperature control is not possible.

The shell in which DICODE is implemented attempts to infer the values of attributes using rules. If no rules exist, then the user is asked to provide a value. If rules exist which could supply the value for an attribute, and they succeed, then the attribute gets a value. If rules exist which could supply a value, but they fail, then the attribute is assigned the value *unknown*. Any rules which use this attribute in the antecedent will provide values of *unknown* to attributes in the consequent. In order to avoid the avalanche effect of giving all attributes the value *unknown*, a demon is used.

The demon monitors the value of the attribute, and when the value becomes unknown, the demon fires. The demon erases the value of *unknown* and then asks the user to provide the value of the attribute. Since inferencing becomes inactive as soon as a demon's antecedent is satisfied, and does not continue until the demon's procedure has been completely executed, a value is provided for the attribute before any other attributes are inferred to be unknown. The following demon is a translation of one used in the DICODE knowledge base which demonstrates the form of the type 4 demons.

> When condensate temperature control option becomes unknown
> then erase condensate temperature control option and
> ask the user for condensate temperature control option.

The type 4 demons may not be necessary depending upon the features of the shell used for the expert system. Some shells behave as described above, others automatically ask the user to provide a value when one cannot be inferred. The most elegant way to handle the above problem is to use a shell which allows the knowledge base author to specify the behavior of the system with respect to handling sources for attribute values.

5.4. Shell Requirements for GTST Use

Chapter 4 presented the methodology for using the GTST to build an expert system through the use of an example which has been implemented.

It explained the GTST used, the frame system created, and the rules used to implement an expert system for the design of distillation column controls. The discussion above is a generalization of the concepts used to create that expert system. The generalization described forces some requirements on the shell to be used for the expert system.

The previous sections imply that the shell used to implement the expert system must provide certain features. First, in order to reduce the number of rules required to represent the goal tree, frame representation of knowledge must be allowed. The *has a* relationship is useful, but not necessary since it can be implemented through the use of attributes attached to the frames. The *has a* relationship allows simplification of the rules by generalizing the rules to consider as a group, only those objects which are *parts* of a specific object. Further generalization can be made by specifying the main object as any of a group (or class) of objects which have the same parent frame.

Access oriented programming as provided by the use of demons adds a facility which cannot be implemented using rules. The use of demons is necessary for the situation above where success paths can be common to several lowest level subgoals, but can satisfy only one. Access oriented programming is useful for other purposes not directly related to the GTST model for knowledge. Demons can provide excellent recommendation and warning facilities during expert system execution. They are also useful for prompting the user for necessary information which could not be obtained through inferencing, or which

is needed to resolve some conflict in user provided information.

The GTST model used directly in the knowledge base of an expert system provides the ability to trace bugs in the knowledge base. If the goal tree is properly represented, the root goal must be satisfied in order for the problem to be solved. If the root goal is not satisfied, the structure of the goal tree allows direct tracing down the hierarchy to find the lowest level subgoal which was not satisfied. The lowest level subgoal is the cause of failure to satisfy the root goal and all goals between. Once the lowest level subgoal which was not satisfied has been identified, the rules which apply to only that subgoal need be examined to determine the cause of the failure. The goal tree structure lends its own organization of the knowledge base. The rules are organized into three groups, the goal tree rules which link goals and their subgoals, the goal tree success tree rules which link the lowest level subgoals and their success paths, and the success path rules which establish the success of the success paths. The success path rules contain the problem domain specific knowledge which is used to determine under what conditions the success paths may be used to satisfy a particular subgoal. These rules are further grouped by lowest level subgoal. If used properly, the GTST model organizes the rules as well as the use of objects or frames.

CHAPTER 6

CUSTOMIZED SHELL FOR CONTROL DESIGN

In the previous chapter, the knowledge base building methodology proposed as the subject of this thesis was outlined in detail. This chapter describes the generalizations drawn from the results of the project aimed at formalizing a method for knowledge base construction in the control design domain. Once a formal method has been established, the requirements of the shell used to implement the expert system can be determined. Furthermore, a specific representational form of the knowledge within the expert system can be established. This chapter describes the shell requirements and the form of knowledge representation within the expert system. Other suggested shell features will also be discussed.

Customization of a shell which facilitates the construction of a knowledge base for control system design is also discussed. The customization described in this Chapter provides the facilities to enter the goal tree into the expert system in a simple manner. However, the goal tree must be defined for the problem as the customization does not have the ability to construct the goal tree through

interaction with the expert. The customization also provides the rules to link the goals to each other and their success paths. Another facility provided by the customization is the form of the rules which use external information to determine the control loops based on criteria for control loop adequacy. The customization also includes demons which are necessary for the regulatory control design problem and to relax the criteria on measures of control adequacy in a user defined manner. The resulting partial knowledge base can be used to create expert systems for control design for any process more rapidly than if the knowledge base author had to start with only the shell.

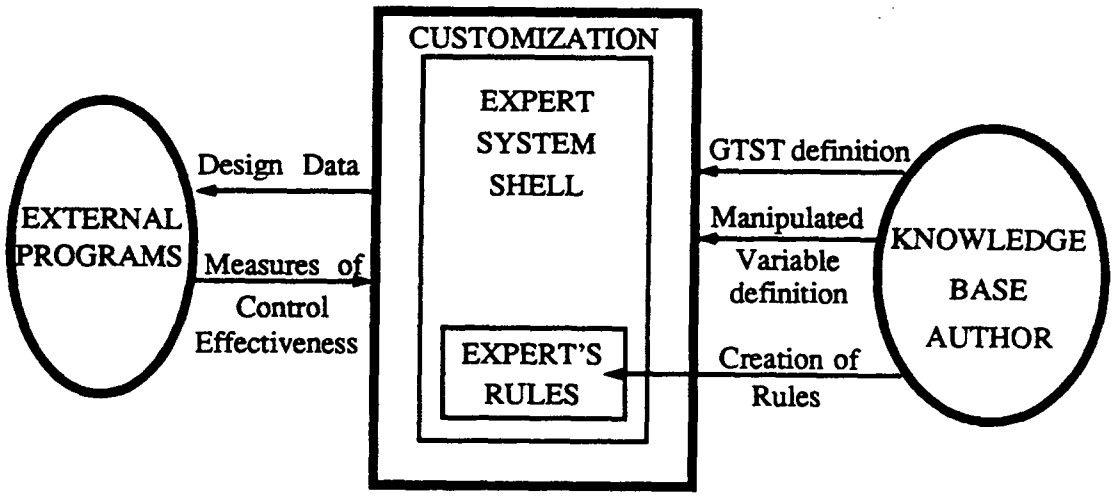


Figure 6.1. Functional representation of customization to an expert system shell.

Figure 6.1 shows the functional representation of the expert system shell, customization, and the knowledge supplied by the knowledge base author. The function of the customization is to provide the mechanics of the problem solution

method for the knowledge base author, allowing him to concentrate on the rules specific to the expert. The knowledge base author must define the goal tree to be used for the problem solution, and enter the rules governing the creation of controllers following the rules of a domain expert.

6.1. Shell Requirements

The first section of this chapter discusses the features required of the shell in order to use the knowledge base building method described in Chapter 5. These features are needed to implement the customization in the form of a partial knowledge base discussed in the second part of this chapter. Other suggested features are also discussed as they relate to the success of the expert system as a design tool.

6.1.1. Integration of Heuristics and Procedures

All expert system shells allow the use of heuristic knowledge in the form of rules of some kind. However, the control design problem involves calculations which provide the designer with supportive information based on equipment design and some kind of process model. The first area of consideration is the ability of the shell to utilize these numerical procedures. The shell in which the expert system is to be implemented should allow the use of programs which

provide this information, without extensive modification of those programs.

Typically, shells which allow execution of other programs communicate with the external programs through the use of data files in a specific format. Some shells actually allow direct communication with standard data base management programs [Tuason and Bispo, 1987]. Others simply have a standard text file format which they use to communicate with other programs. If this is the case, then the external program must create a text file in the proper format for the shell to read [KES, 1988]. Some shells actually allow modification of the shell's behavior through definition of functions usually written in LISP [Raphals and Chassell, 1986].

The ability to run programs external to the shell is very important, particularly if programs already exist which use design information to simulate the process. Simulation results are often the final criterion used to configure the controls for a process, and as such they are vital information to the control design procedure. The simulation programs may be "canned" and not allow modification by the knowledge base author in order to communicate directly with the expert system. However, most simulation programs produce data files which can be manipulated by another program to translate the results into a form compatible with the expert system shell.

6.1.2. Frame Structure

The shell must allow the use of frames to represent the knowledge. Frames are used to represent the goals, success paths, equipment, and control idioms (or loops). The shell must allow creation of frames during execution since the number and identity of the equipment and control loops are not necessarily the same for each design. Frame inheritance is a feature which is generally useful, but not necessary for the control design problem. The use of inheritance may be useful depending on other capabilities of the shell.

One shell the author is familiar with allows construction of the frame representation from two perspectives. The first is the *inheritance model*, and the second is the *domain model* [Raphals and Chassell, 1986]. In the inheritance model, objects or frames which inherit properties from another object or frame are represented as *kinds of* the parent frame. In the domain model, the *has a* relationship is represented. Here, frames which are considered parts of another frame are represented as such. The *has a* relationship can be used to advantage in the knowledge base for an expert system for control design by representing subgoals as parts of the goal immediately above. During inferencing, the subgoals of a particular goal can then be referenced as a group. Other shell limitations can interfere with the utilization of this *has a* relationship in the manner described above. The section below concerning the customization of a general purpose shell discusses these limitations.

6.1.3. Chaining

The methodology for creating the knowledge base for an expert system for control design assumes that a primarily backward chaining inferencing strategy will be used by the shell. The fact that backward chaining is required is not a drawback, but rather an advantage. Even the least expensive shells provide backward chaining in order to obtain the results of a consultation. The goal tree formulation of the design problem conforms it for solution in the backward chaining manner. The only precaution that should be taken is that the shell must allow some kind of priority during backward chaining in order to resolve conflicts created by competing goals.

The simplest implementation of priority is the depth first search strategy for antecedent satisfaction. If a rule has multiple conditions in the antecedent, and the first is taken and pursued to exhaustion before the second is considered, then a priority based on antecedent order is implemented. This depth first antecedent order priority is the case for every rule considered, therefore the depth first strategy is recursive in nature, pursuing the first antecedent of each rule considered in back chaining. Even this primitive form of priority fulfills the requirements of the shell to be used for the expert system.

6.1.4. Access Oriented Programming

Use of the methodology outlined in Chapter 5 requires that the shell provide access oriented programming. In the general case, access oriented programming

is the provision for procedures as facets on the slots of frames. These facets specify actions to be taken, or procedures to be executed under the conditions defined by the type of facet. The most common of these facets is the *when changed* facet. The *when changed* facet fires or executes whenever the value in the slot to which it is attached changes. The procedure to be executed is usually flexible in nature, allowing modification of the current state of the problem solution.

Some shells provide the use of *demons*. Demons are more general than the *when changed* facet. Demons are simply executable procedures which become active under specified circumstances. Demons provide access oriented programming by monitoring the value of a slot in a frame, and executing when the slot takes a specific value. Each shell has its own specific implementation, but not all allow the use of access oriented programming. Since access oriented programming is necessary in order to use the methodology described in the previous chapter, this feature is one which must be considered with care. For specific information as to why access oriented programming is needed for the problem solution, see Chapters 4 and 5.

6.1.5. Justification of Results

One feature which is absolutely critical to success of an expert system is the ability to justify the results obtained. The user must be able to obtain

reasons for the results which he is presented. Justification or explanation of the results builds the confidence of the user. Gaining the users respect is the difference between a successful program and an unsuccessful one. If the user respects the program's ability, then he can trust the expert system to the extent that he can determine the reasons for the results. Justification of the results is practically necessary during debugging as well as for the purposes of gaining the user's confidence.

Another feature which helps to accomplish the same purpose of gaining the user's confidence is explanation. The shell should supply a feature by which the knowledge base author can supply explanations for the various user supplied attributes. That is, the author should be able to supply a text description or explanation which is displayed by the shell if the user responds to a question with *explain* or *why*. These explanations are provided to guide the user to a response which most closely represents the process, equipment design, and control objectives. Providing explanations of this kind is most easily accomplished if the shell provides a facility to perform the function.

6.2. Shell Extensions

The shell may be customized with a partial knowledge base which provides certain relationships between frames in order to facilitate use of the methodology described in Chapter 5 for building the knowledge base for an expert system for

control design. The next sections describe some useful extensions in the form of a partial knowledge base. The remainder of the chapter is devoted to outlining the partial knowledge base from a top down point of view.

6.2.1. Goals and Success Paths

The method for building the knowledge base involves defining a goal tree success tree model for the control system design. The skeleton for the goal tree success tree as represented in the knowledge base can be provided regardless of the specifics of the goal tree itself. That is, the goal tree and success path framework can be defined as frames with the appropriate relationships. It may be necessary to define several classes or parent frames, one for each level in the goal tree, or it may be possible to define a single class of goals and still accomplish the task of building the appropriate structure. The need for multiple classes of goals depends upon the rule capabilities of the shell used. A class of success paths should also be created which will satisfy some of the goals (the lowest level subgoals). The following paragraphs describe the specifics of these two classes or frames.

6.2.1.1. Goals as Frames

Depending on the shell, it may be necessary to create several classes of goals. The object is to create frames to represent the goals with necessary at-

tributes already defined. A goal requires only one attribute to indicate whether it has been satisfied or not. The value of that attribute is determined by the value of the same attribute of the subgoals which lie below the goal. If any one of the subgoals is not satisfied, then the goal is not satisfied. The *has a* relationship between a goal and its subgoals is useful, and can be implemented using attributes for the goals. The implementation of the *has a* relationship is discussed further below as it relates to the rules used to link the goals.

6.2.1.2. Success Paths as Frames

Frames representing the success paths must also be created. It is more convenient to create *Controllers* rather than success paths in the case of control system design. Controllers should have attributes which describe the controlled variable and the manipulated variable. When the the success path associated with a manipulated variable is satisfied, that is, it has been determined to be the best for a particular controlled variable, a controller is created, and the attributes representing the controlled variable and manipulated variable are given appropriate values. Thus, the success paths represented as manipulated variables satisfy the lowest level subgoals represented by controlled variables when a controller links a manipulated variable to a controlled variable.

Several classes of controllers should be created to represent the regulatory, supportive, and constraint controllers. It is necessary to distinguish the regulatory controllers from the rest of the controllers because during regulatory

control system design, extra constraints must be imposed as to which manipulated variable can be used for control. For instance, for regulatory control, each manipulated variable can be assigned to only one controller. However, more than one constraint controller can manipulate the same manipulated variable, which in turn can have its associated regulatory controller.

6.2.2. Rules for GTST

Rules must be given to represent the links between the goals. In theory, only one rule is required to link the goals together, provided the goal tree has been built properly. This one rule relies on the ability to express the *has a* relationship between a goal and its subgoals. As discussed in Chapter 5, the rule could look something like:

> If a Goal2 of Goal1 is not satisfied, then
> Goal1 is not satisfied.

This rule states that if any one of the goals which Goal1 *has* is not satisfied, then Goal1 is not satisfied. Goal1 and Goal2 are variables which are bound to each of the goals in turn. This rule is very powerful, and as such can cause difficulties during inferencing. The problem arises because this rule must be used many times in a recursive manner in order to work downward into the goal tree. The shell in which DICODE was implemented was able to chain backward through this rule in a recursive manner and obtain a correct result,

however the shell produced a warning message each time the rule was used recursively. The warning indicated that an attempt was made to obtain the value of the attribute *satisfied* which was already in the process of receiving a value, and it could be the result of a cycle in the attribute hierarchy. The warning probably results because the rule must be used more than once for each goal, as well as being used recursively due to the inheritance implemented in the class definitions for *goals*. The logic is correct, but the shell somehow becomes confused while using the rule recursively.

Note that this rule requires the ability to describe the *has a* relationship. The *has a* relationship may be provided as a function by the shell, but if it is not, then the relationship can be defined through the use of attributes of the goals. For example, if Goal2 and Goal3 are subgoals of Goal1, then we can define two attributes called *uplink* and *downlink* for all goals and give the value of *uplink* of the goals Goal2 and Goal3 the same value as the attribute *downlink* of Goal1. Then, the rule linking goals could be of the form:

> If the downlink of Goal1 = the uplink of Goal2 and
> the satisfaction of Goal2 is false, then
> the satisfaction of Goal1 is false.

Note that this rule requires the ability to compare values of different attributes. The ability to compare values of two different attributes which are not numeric may not be possible in all shells. If this is the case, then the attributes could be defined as being integers, which any shell should be able to compare.

The use of attributes to describe the *has a* relationship does have the

advantage that the individual using the partial knowledge base must simply provide the correct values for attributes of the goals rather than specify the *has a* relationship in the definition of the goals. Therefore, the shell could simply read a file which contains the goal tree structure in the form of the instances of the goals with their appropriate attribute values which form the goal tree.

If the shell is unable to backward chain through multiple instances of the single rule given above, one class of goals can be defined for each level in the goal tree. Then, one rule is required to link any two levels of the goal tree. For example, if the two classes *goal* and *subgoal* are created as two adjacent levels in the goal tree, then the following rule would be used to link the two levels.

> If the downlink of any goal = the uplink of any subgoal and
> the satisfaction of the subgoal is false, then
> the satisfaction of the goal is false.

This rule is much easier to understand, and one would expect that a shell would have less difficulty chaining through the goals, since each of the rules would only be used once at any time, while the single rule described before must be used recursively. In fact, the KES^{TM} shell was able to chain backward using a separate rule to link each two adjacent levels of goals without problems.

Rules which establish the satisfaction of the lowest level subgoals must also be provided. The lowest level subgoals each represent a control objective in the form of a process variable to be controlled by a specific controller. It is therefore convenient to express the satisfaction of the lowest level subgoal as having obtained a controller whose controlled variable is the same as the

controlled variable with which the subgoal is associated. In other words, if the system has been able through the use of other rules to find a controller linking a manipulated variable to the controlled variable associated with the subgoal, then the subgoal has been satisfied. The rules to satisfy the lowest level subgoal would look something like the following.

> If the variable of a goal = the controlled variable
> of a controller then
> the satisfaction of the goal is true.

Note that the above rule deals with the satisfaction of a subgoal rather than the failure of satisfaction. Since the rules linking the goals with each other dealt with failure of satisfaction, some confusion could result depending upon the way in which the shell assigns default values to attributes. For instance, if the above rule fails, then the goal should be unsatisfied. If the rules fail which cause the goals to be unsatisfied based on the failure of a subgoal to be satisfied, then the goal should be satisfied. The difficulty arises due to the fact that under different rule failure, the attribute must take a different default value. In order to solve this problem, one of several methods should be considered. One approach is to change the rules to all deal with satisfaction of the goals rather than failure to satisfy the goals. The rules to be changed would then have to check all of the subgoals of a particular goal to determine satisfaction. The shell must allow checking all of the goals as an option in writing rules. Other options utilize facets or demons as described in the next section.

The final set of rules are those to be supplied by the knowledge base author

adding to the partial rule base. The rules to be supplied are those which outline the conditions under which a specific manipulated variable is to be used by a controller for a measured or controlled process variable. These rules set up the controllers linking the success paths or manipulated variables with the lowest level subgoals or controlled variables. The rules of this type have the following form for the regulatory control design.

> If (conditions under which manipulated variable M1 should
> be used to control controlled variable CV1) and
> availability of manipulated variable M1 is true
> then create regulatory controller C1 and
> controlled variable of C1 is CV1 and
> manipulated variable of C1 is M1.

Note that the antecedent of this rule checks the intended manipulated variable to ensure it is available for regulatory control pairing. It was mentioned above that each manipulated variable can be assigned to only one controlled variable for regulatory control. The check enforces this constraint. However, a demon or facet is needed in order to update the *availability* of the manipulated variables each time a rule assigns one to a regulatory controller. That demon is described later in the section below concerning facets.

6.2.3. Facets or Demons for GTST

As mentioned above, a problem exists in assigning default values to the attribute *satisfaction* of the goals depending upon which type of rule failed to

give it a value. One solution was to change the form of the rules which base the *satisfaction* of a goal on the satisfaction of its subgoals. The rules would have to check all subgoals for satisfaction rather than any one for failure. Most shells do not allow checking an attribute of all the members of a group for a specific value in conjunctive form without writing the members out specifically. Another possible solution to the problem is to create facets which assign default goal satisfaction values appropriately depending on whether the subgoal is one of the lowest level (only success paths below it) or not. In the framework described above, a demon could be executed any time the satisfaction of a goal becomes unknown. The demon could then check to see if the goal is a lowest level subgoal, in which case the goal should become unsatisfied. If the goal is not a lowest level subgoal, the goal should become satisfied. In the context of the rules described above, the lowest level subgoals can be distinguished from the others by the value of the attribute which identifies the controlled variable associated with it. Only those subgoals which are lowest in the goal tree will have a value of this attribute other than the default value. Therefore, the demon could look something like the following.

```
When the satisfaction of a goal = unknown
then
if the variable of the goal is not the default
     then the satisfaction of the goal is false,
else
     the satisfaction of the goal is true.
```

Another purpose of demons in the partial knowledge base is to resolve the

problem of the same manipulated variables (or success paths) having the ability to satisfy more than one control objective described by a lowest level subgoal. In the case of the regulatory control system, any single manipulated variable can be used to control only a single controlled variable. In order to ensure that each manipulated variable is used only once for regulatory control loop pairing, a demon can be used to change an attribute of another class of objects called *manipulated variables* when it is used by a controller. This attribute will be changed to indicate that the manipulated variable is no longer available. The rules which create a controller that uses a particular manipulated variable must then check in the antecedent to ensure that the manipulated variable has not already been used.

Use of the demon described above requires that another class of objects be defined in the partial knowledge base. The class *Manipulated Variables* is created and given two attributes. The first is an attribute indicating which variable it represents, and the second indicates the availability for use by a controller. Let the identifying attribute be *name* and the availability attribute be *availability*. The demon would look like the following.

When the manipulated of a Regulatory Controller =
the name of a Manipulated Variable
then the availability of the Manipulated Variable is false.

Note here that this demon will apply only to the regulatory controllers, and not to the other controllers such as the supportive, and constraint controllers. The supportive and constraint controllers are treated separately since they are

allowed to use the same manipulated variable for more than one controller, and indeed use the same manipulated variables already used for the regulatory controllers through the use of overrides.

6.2.4. External Program Use

Typically, calculation programs are used to provide information about the quality of control achievable by a certain control system. Examples of these measures are the relative gain array (RGA), relative disturbance gain (RDG), inverse nyquist array (INA), and other results based on simulation of the process. The results of calculation of these measures must be transferred to the expert system, and then used appropriately within the expert system to select control loop pairings.

One way to standardize the use of these *tools* is to create a class of objects called *Tools* with attributes which describe limits of acceptable ranges for adequacy of control. For example, DICODE makes use of the RGA of numerous composition control pairings in order to determine the most acceptable pairing. Since the RGA is a measure of control interaction based on a particular pairing of controlled and manipulated variables, the RGA of each possible pairing must be considered and compared to acceptable values. The frame representing the RGA would have attributes which describe acceptable values for the RGA of any pairing. It is also necessary in this case to create another class of objects

representing the various control loop pairings each with an attribute taking the value of the RGA for that pairing. Several of these *Tools* can be used which have the same basic function of comparison of control loop pairings. The RDG is another similar tool, as is integrated error resulting from a disturbance. Other measures may simply evaluate the adequacy of a single pairing of a controlled and manipulated variable. If the measures can be quantified as real numbers which are passed to the expert system, then the expert system can compare the measures of the pairings to acceptable values in order to determine the best control system.

The partial knowledge base should provide a frame definition for the pairings with an attribute to indicate its adequacy. The individual expanding the knowledge base must provide attributes for the pairings to take the values returned by the external program as measures of adequacy. He must provide as many attributes as there are tools to evaluate the pairings. The partial knowledge base will also provide the format for rules to evaluate the pairings based on "good" and "bad" values of the tools. The way in which these tools are used can be demonstrated by the following rule.

> If the RGA of a pairing is greater than the large value of RGA
> then the possibility of the pairing is false.

This rule indicates that if the value of the RGA for a particular pairing is larger than the value considered large, then the pairing should not be used. The attribute *possibility* is used as the attribute to indicate the pairings success

or failure of a test.

Two different philosophies can be adopted at this point. One is to use threshold values for the tools and allow perhaps several pairings or manipulated variables as possible for each controlled variable, and then arbitrarily select among them for the final result. This approach has the advantage of producing several alternative designs as adequate with the disadvantage of difficulty in presenting the possible designs.

Another approach is to begin the threshold values at an ideal value such that under most circumstances none of the pairings will survive the tests. Then, each time all possibilities are rejected, the threshold values are loosened by a predefined amount, the attribute *possibility* of all pairings is erased, and the search is initiated again. This process is continued until either at least one possible design survives the tests, or absolute constraints are reached on all of the tool thresholds. If absolute constraints are reached on all tools, then obviously no result can be obtained, and the user should be informed of this result. Otherwise at least one design has survived and can be presented as the result.

It should be noted that the values of the thresholds are loosened in a predefined way. The change in threshold value should be an amount reflecting the accuracy or reliability of the measure. For example, if a measure such as the RGA based on a simple steady state model is used, then values within 10% of one another are probably essentially the same. Therefore, the value considered

large for the RGA should be increased by 10% each time. It is then possible that multiple designs survive the test on the same relaxation cycle. If multiple designs survive, then one of them must be chosen as the result based on some perhaps less reliable information since by all reliable information, the surviving possible designs are equally adequate.

This second approach is the one used in the expert system DICODE and has been found successful. It does however require the use of some demons or facets in order to implement the relaxing of the threshold values, resetting the appropriate attributes, and reinitiating the chaining with the new values of the thresholds. In the case of DICODE, the RGA, RDG, and relative stream sizes are used to select among the possible composition control pairings. Each of these measures are relaxed on each cycle until at least one pairing survives. Since the values of these measures are all based on the same steady state model of the column, they are essentially equal in accuracy and reliability. Final selection of the control design is accomplished by using an estimate of the relative integrated error resulting from a disturbance to the process. The reliability of the integrated error as a measure of control adequacy is somewhat questionable because it can hide oscillatory behavior. Therefore, the integrated error is used only as a last resort to select between otherwise equivalent control designs.

Control system selection need not be based on the above method. The measures could as easily be applied to specific pairings of one controlled variable with one manipulated variable. As an example, consider a regulatory control

system which consists of three manipulated and three controlled variables. The RGA for this system could be used to select the control variable pairings on a one by one basis with the further constraint that each controlled variable must be paired with a different manipulated variable.

The result of the implications discussed in the last few paragraphs is that another rule is required to determine whether any of the pairings has survived and a demon must be used to relax the thresholds whenever the rules reject all possibilities. The rule should look something like the following.

```
If the possibility of any pairing is true
then the pairing process is complete.
```

A default value of *not complete* should be provided for the attribute *pairing process* when the above rule fails. Then the demon to relax the thresholds will be similar to the following.

```
When the pairing process is not complete
then
    for each tool
        the large value of the tool = (the large value
            of the tool)*(1.0 + the relaxation of the tool)
        the small value of the tool = (the small value
            of the tool)*(1.0 - the relaxation of the tool)
        if the large value of the tool > the large constraint
            of the tool
        then the large value of the tool = the
            large constraint of the tool
        if the small value of the tool < the small constraint
            of the tool
        then the small value of the tool = the
            small constraint of the tool
    for each pairing
```

the possibility of pairing is undetermined
pairing process is undetermined
obtain value of pairing process.

The above demon takes care of relaxing the threshold values until the constraints are met. It also resets the values of the attribute *possibility* of the pairings and the value of the attribute *pairing process* to undetermined. It then reinitiates the inference process to obtain the value of *pairing process*. Another demon must check to see if all constraints have been met. The demon for this purpose should be similar to the following.

When the pairing process is not complete and
the large value of all tools = the large constraint of the tool
and the small value of all tools = the small constraint of the tool
then display the failure message and stop.

The demon above traps the situation where none of the possible configurations survives the tests at the absolute limits of acceptable adequacy. In this case, the constraints have simply been set too tightly for any control design to survive the tests. The demon should display some kind of message indicating the circumstances and then discontinue the session, or reset to start a new session.

The last piece of the knowledge base is the rules which deduce the control loops based on the pairing selected as the best possibility. These rules must be provided by the knowledge base author since the creation of controllers to satisfy the lowest level subgoals is based on a loop's association with the pairing determined as best. That is, each pairing has a different set of manipulated and controlled variable pairings. These rules also must reference the attribute

pairing process in order to make the backward chaining process complete. The rules of this type can be exemplified by the following rule.

> If the pairing process is complete and
> the possibility of DV is true
> then create instance Distillate Composition Controller of
> frame Regulatory Controllers and
> manipulated of Distillate Composition Controller is distillate and
> controlled of Distillate Composition Controller is
> distillate composition.

6.3. Summary

Experience in creating expert systems for a particular purpose leads to some insight into the requirements of the software to be used. In the case of expert systems for control design, the requirements are clear when the problem is modeled as a goal tree success tree (GTST). Other shell provided facilities are found necessary in order to satisfy performance criteria of the final expert system. In summary, an expert system shell for control design must have the following characteristics when the GTST model is used for the problem solution.

1 Backward chaining inference strategy
2 Frame representation of knowledge
3 Rules allowing priority
4 Access oriented programming
5 Ability to run and communicate with external programs.

Other requirements often imposed by the intended use of the expert system include the following.

1 Ability to justify results (final and intermediate)
2 Explanation facilities which can be modified by the author
3 User interface which can be modified by the author.

When modified in the ways described above, the expert system shell can provide most of the mechanics needed for the control design problem to be solved in an expert system. The framework for the goal tree is provided in a way that the structure of the goal tree for a specific problem can be created simply by creating instances of the goal frames and providing appropriate values for attributes which describe the goals relationship to the others of the goal tree. The framework for the controllers is also provided along with a procedure to ensure consistency in manipulated variable selection. The last section discussing the use of external programs as information upon which to base the variable pairing is not necessarily the best way to accomplish the task. INITIALLY, the author used simpler rules in the DICODE knowledge base, considering each loop separately and selecting the best manipulated variable. However, the method using RGA, RDG, relative stream sizes (L/D and V/B), and estimated integrated error works well and has now been implemented in the DICODE knowledge base. Appendix D gives a listing of a partial knowledge base written in the KES^{TM} shell which implements the GTST model and provides the framework for regulatory control loop selection based on measures of control system adequacy.

Implementation of the configuration selection criterion discussed above can be quite involved, and actually create difficulties in the inference process. The

research has shown that most of the control design method can be cast into an organized and structured model with the possible exception of the regulatory control loop pairing. This part of the control system design is not specifically addressed by idiomatic control and is expert dependent. It is therefore difficult to draw any generalizations from the implementation of a specific expert's knowledge. The next chapter discusses the use of simulated neural networks to solve the regulatory design problem.

CHAPTER 7

NEURAL NET FOR CONTROL DESIGN

The purpose of the research reported in this thesis is to formalize a method for constructing the knowledge base for an expert system in the control design domain. The purpose of the method is to give structure to the knowledge base in order to reduce the effort required during expert system creation, extension and modification. Most of the problem solution has been cast into an organized structure successfully. The only part of the problem solution for which little structure can be defined is selection of the control loop pairings for regulatory control. The reason for the difficulty encountered in this area is that the solutions tend to be expert dependent. That is, the structure of the solution method varies from expert to expert. Some experts use simple heuristics based on column design parameters to decide upon control loop pairing. Others use information based on steady state models of the process. The information used to decide upon the regulatory control loop pairing can range in complexity up to results from detailed dynamic simulations.

In response to the difficulty encountered with respect to regulatory control loop pairing, a different technique to modeling the problem has been used. Rather than relying on the expert to relate his solution method to the knowledge engineer through solving several design problems, a simulated neural net can be used to solve the problem. A simulated neural net can be trained to produce the control loop pairing based on the same information the design engineer uses. Therefore, the net can be trained on several examples taken from actual columns in operation. Before delving into details of this application of simulated neural nets, some background information is necessary.

7.1. Neural Networks

Simulated neural nets attempt to reproduce output data from input data by dense connection of simple nonlinear computational elements. These computational elements use a weighted sum of the inputs, pass the sum through a nonlinear function, and produce a single output. Figure 7.1 shows the form of a single node or element of a neural net. Figure 7.2 shows the typical sigmoidal shape of the nonlinear function used in the processing element or node. The function is given as

$$f_j(a_j) = \frac{1}{1 + e^{-a_j}}$$

with

$$a_j(\mathbf{x}) = \mathbf{w}_j^T \cdot \mathbf{x}$$

where

$$\mathbf{x} = (x_1, x_2, \ldots, x_n, 1)^T$$

$$\mathbf{w}_j^T = (w_{1j}, w_{2j}, \ldots, w_{nj}, w_{n+1,j})$$

Dense interconnection of the outputs of nodes as the inputs of other nodes creates the topology of a neural net similar to the biological nervous system. Changing the net in order to produce the desired output from an input is achieved by manipulation of the weight on each connection. The most exciting characteristic of the net is its ability to change the weights on inputs to a node using information local to that node. The automatic changing of weights in order to produce the desired output is described as learning.

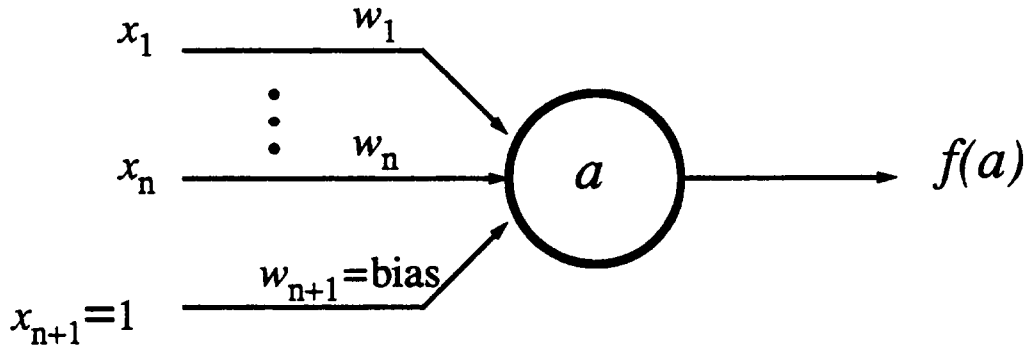


Figure 7.1. One processing node with weighted inputs and one output.

The conception of neural nets can be traced to attempts at modeling the neurons of the human brain in hopes of simulating the biological nervous system. Much of the earliest work in the area of neural nets has been attributed to the neurologists Jackson [Jackson, 1869,1958]and Luria [Luria, 1966]. Their

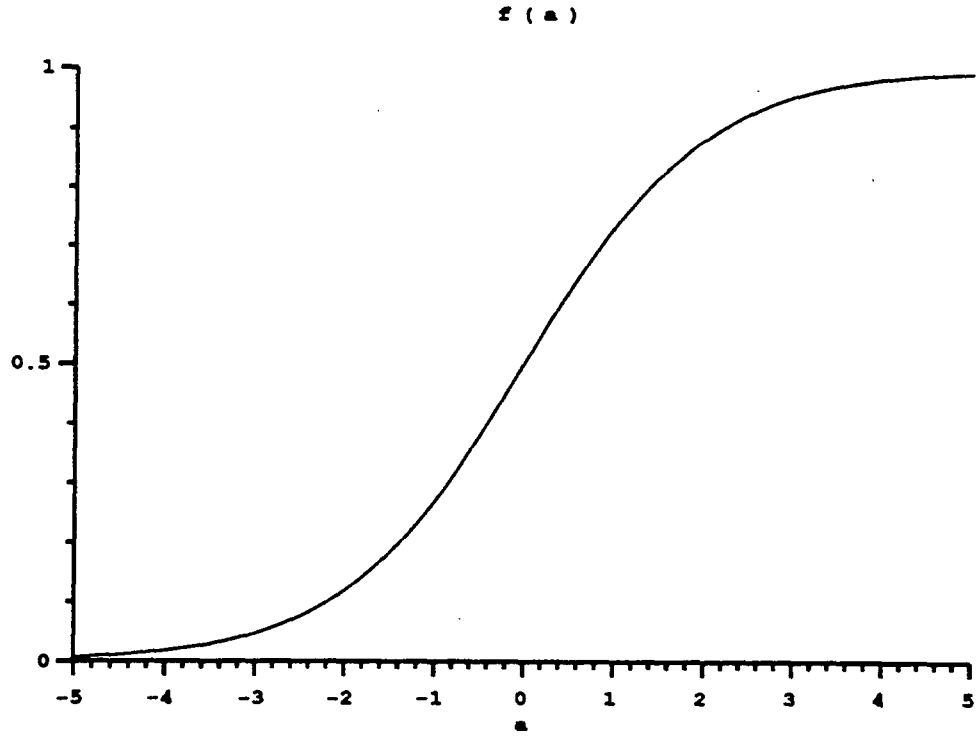


Figure 7.2. Output of processing node as a sigmoidal function activation.

support of the notion of the *dynamic functional system* in which the cognitive process resulted from the parallel functioning of a large number of components set the stage for further development of parallel distributed processing systems. Other earlier contributors to the area include Hebb [Hebb, 1949]and Lashley [Lashley, 1950]. Much of this early work lacked clarity and specificity. Later, others contributed significantly to the clarification of the distributed processing approach. Rosenblatt [Rosenblatt, 1959,1962]layed the foundations for distributed processing computation with the development of the perceptron.

The lack of development of the parallel processing idea in the late 1960's

and early 1970's can be attributed to the advent of the von Neumann computer and serial processing. However, some work continued in the area until the middle of the 1970's when the idea of parallel processing experienced a revival. Since the middle 1970's, extensive advancement has been achieved including hardware dedicated to parallel distributed processing. A summary of much of the current state of parallel distributed processing can be found in Rumelhart, McClelland, and the PDP Research Group [Rumelhart and McClelland, 1986].

Today, several neural net models with different node characteristics have been explored and applied to different problems. The full capabilities of a neural net can be obtained by a topology consisting of one input layer of nodes, one output layer, and one hidden layer located between the input and output layers. Figure 7.3 shows part of a three layer net. The input layer simply distributes the input values to the appropriate nodes in the hidden or second layer. The hidden layer nodes weight and sum the inputs and a bias signal (always 1), send them through the nonlinear function, and distribute the outputs to the output layer nodes. The output layer nodes weight and sum their inputs and a bias signal. The output nodes can use the same nonlinear function, or they can use a linear function. The choice as to which function to use in the output layer depends on the characteristics of the problem. The output of each of the output layer nodes is interpreted as one of the net outputs. Thus, an input vector produces an output vector. During learning, an input vector and output vector are used to vary the weights connecting the nodes in order to minimize

the difference between the net output and the desired output. One requirement seldom mentioned is that the input and output values must be scaled between zero and one.

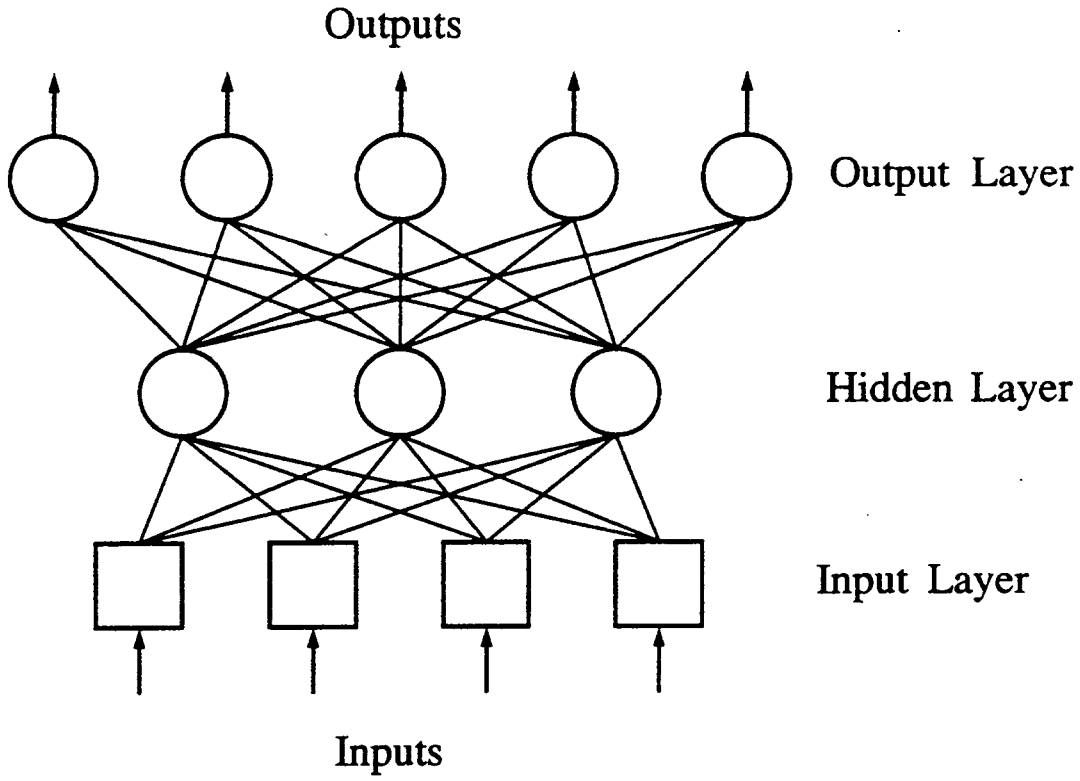


Figure 7.3. Part of a three layer neural net.

The most popular of the neural net architectures is the backpropagation network [Rumelhart and McClelland, 1986]. The backpropagation network is an example of a mapping network because it approximates a mathematical mapping function ϕ from vector $\mathbf{x}$ to $\mathbf{y}$, with $\mathbf{y} = \phi(\mathbf{x})$. Kolmogorov's Theorem [Hecht-Neilsen, 1987, Kolmogorov, 1957]in response to the 13^{th} problem of Hilbert [Lorentz, 1976]gaurantees the existance of a three layer neural network which will provide that functional mapping.

7.1.1. Learning in Neural Nets

Many learning methods have been used in neural net applications, each corresponding to a net with different characteristics. Learning by error propagation is the method used for the regulatory control problem presented later in this chapter, and therefore the discussion on learning will be limited to back propagation of error, or the generalized delta rule. In the generalized delta rule, an input vector is presented to the net. The net produces its output, and that output is compared to the *desired* output. The difference between the net output and the *desired* output is used to change the weights such that the error is reduced. The generalized delta rule (GDR) describes the method by which the weights on the inputs to the output layer of nodes are modified. The GDR defines the change in weight following presentation of input/output pair p as

$$\Delta_p w_{ji} = \eta(t_{pj} - o_{pj})o_{pj}(1 - o_{pj})o_{pi} = \eta\delta_{pj}o_{pi}$$

for the sigmoidal nonlinear node function. Here, t_{pj} is the target for the jth component of the output pattern p, o_{pj} is the jth element of the actual output pattern produced by the presentation of input pattern p, o_{pi} is the output of the ith node (an input to node j). Thus, $\delta_{pj} = (t_{pj} - o_{pj})o_{pj}(1 - o_{pj})$ for the weights on the inputs to the output layer nodes. For the hidden layer nodes, the error is propagated backward using the δ's of the output nodes connected to the hidden nodes. In this case,

$$\Delta_p w_{ji} = \eta\delta_{pj}o_{pi}$$

with δ_{pj} defined as

$$\delta_{pj} = o_{pj}(1 - o_{pj}) \sum_k \delta_{pk} w_{kj}.$$

Here the sum over k is over the nodes to which the output of node j goes as an input.

The GDR may be used for any nonlinear node function which is monotonically increasing and differentiable. For a more complete discussion of the GDR, the interested reader is referred to [Rumelhart and McClelland, 1986, p. 318]. The GDR learning procedure is an implementation of a gradient descent to the weights which minimize the sum of squared differences between the actual and desired output values summed over the output nodes and all pairs of input output vectors. The gradient descent minimization is strictly enforced only if the δ's are summed over the entire input/output set and the weights updated only once each sweep of the data. However, if implemented only after a complete sweep of the data, the GDR converges on the minimum at a very slow rate. By applying the GDR after each presentation as described above, learning is much more rapid, however a learning factor η which is less than one must be used to avoid instability problems.

7.1.2. Classic Problems

Many problems have been used to demonstrate the function and applicability of neural nets. One of the most often solved problems is the logical exclusive

or problem. The input consists of two binary digits, the output a single binary digit. Figure 7.4 shows the configuration of a three layer neural net used to solve the problem. Figure 7.4 shows a net with two input nodes (one for each input), two hidden layer nodes, and one output layer node. Note that the input nodes are not connected to the output node. Learning in this net is accomplished using backpropagation of error. This particular net topology can fail to learn the input output relationship due to the presence of a local minimum in the sum of errors squared with respect to the weights. Figure 7.5 shows a three layer configuration in which a single hidden layer node is present, but the input nodes are connected to the output nodes. This net topology does not fail to learn the input/output relationship, but does require approximately 500 sweeps through the set of four input/output vector pairs.

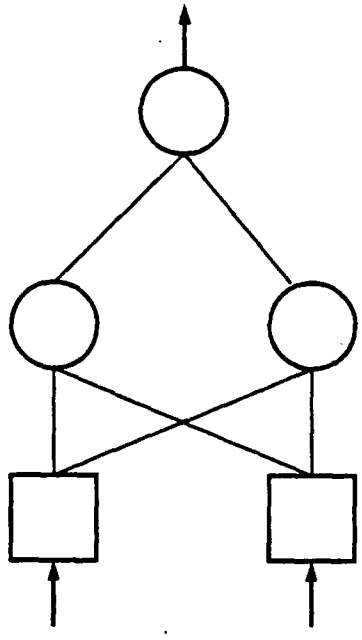


Figure 7.4. Net to solve XOR with two hidden layer nodes.

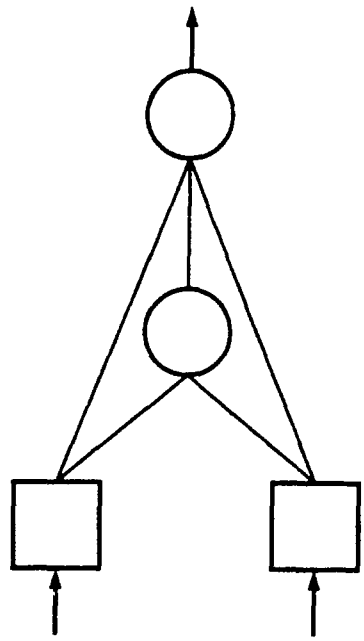


Figure 7.5. Net to solve XOR with one hidden layer node.

Other typical problems used to demonstrate the back propagation neural net include the parity problem, the encoding problem, addition, and other problems in which the input and output vectors represent binary numbers. In fact, few problems are presented in which the inputs and outputs can take real continuous values. Some evidence that neural nets can handle real continuous input values has been supplied by studies using neural nets to classify sonar targets based on the sampling of the amplitude of a sonar signal at different frequencies [Gorman and Sejnowski, 1988]. The application addressed by the remainder of this chapter uses two real continuous inputs along with twelve other binary inputs to produce ten binary outputs.

7.2. The Regulatory Control Design Neural Net

The expert system DICODE designs the control system for a binary distillation column based on user provided input. Some of the input consists of attribute values which can take one of a set number of possibilities. One example of this kind of input is the type of condenser used in the equipment design. Other examples of input include real numbers whose values can range from zero to infinity, or zero to one. Examples of these inputs include reflux ratio, compositions, and feed quality. One of the results of DICODE is the regulatory control system. Since the regulatory control design problem uses rules which are expert dependent and therefore a generalization of the problem solution structure is difficult to make, it was decided that perhaps one possible solution is to design a neural network to learn regulatory control system design by example.

7.2.1. The Problem to Be Solved

In order to determine if a neural net can be used to design the regulatory controls for a process, a data set of several input/output pairs must be created and used to train a neural net. The expert system DICODE uses rules to accomplish the task of designing the regulatory control system for binary distillation columns based on the knowledge of a leading domain expert. DICODE was used to produce the input/output data pairs to be used for the training of

a neural net. Therefore, the task of the neural net was to reproduce the results which DICODE yields, based on the same input data.

An examination of the DICODE knowledge base indicated that a minimum of eleven attribute values must be considered, each taking anywhere from two to seven different values, for a total of over 100,000 different possible combinations of input. Those 100,000 combinations supplied input to only consider relative stream sizes to to determine the level and composition control loops and considered pairing each controlled variable with a manipulated flow (that is no ratios were considered). An estimate of the number of sweeps of the data set required for learning was 500, indicating that a total of approximately 50,000,000 presentations of an input/output data pair to the net would be required in order to learn the input/output relationship. The problem was obviously too large to obtain results in a reasonable time which would indicate whether the application was valid.

Further examination of the problem indicated that twelve rules could be used to determine the pressure control loop. Once the method of pressure control has been established, the number of combinations of input required is reduced to approximately 6000. Some of the input combinations are not valid, and eight rules can be used to correlate the inputs resulting in a total of about 800 valid combinations of input. An important result which was recognized at this point was that a total of twenty rules could be used to reduce the size of a problem from 100,000 input combinations to 800, a factor of 125 decrease in size.

The significant reduction in size due to twenty rules demonstrates the power of an expert system for problems in which a combinatoric nature is present. A target of 100 input/output data pairs was set in order to make the problem of manageable size. One attribute was selected to be omitted (its value fixed at one of those possible), and another was reduced in its possible values in order to reduce the number of input combinations. Also, one input used (shown as the second entry in Table 7.1) is actually a derived quantity rather than a value used as input by DICODE. This quantity is the ratio of vapor boilup to bottom flow in the column. DICODE calculates the vapor boilup to bottom flow ratio from the reflux ratio, and feed and product purities by using a material balance on the column. This value was known to be an important one in the design problem, and helped to reach the 100 I/O pair target.

The 100 input combinations were generated using a FORTRAN program which wrote 100 data files in a format compatible with the KES expert system shell. The DICODE knowledge base was then modified to read the input files one at a time, infer the regulatory control system, and generate an output file containing both the input and the results. Another FORTRAN program was written to convert the file created by DICODE into a direct access file containing numeric values for the inputs and outputs to be used by the neural net.

7.2.2. The Network Used

Backpropagation of error was chosen as the network architecture, and a simulation was developed in FORTRAN on a microVax. The network consisted of either 14 or 15 input nodes, 3 to 25 hidden layer nodes, and 10 output nodes. The interpretation of the inputs is given in Table 7.1, and the interpretation of the outputs is given in Table 7.2. The program used to simulate the neural net presented the data for learning randomly. For each presentation of a data point, a random integer ranging from one to 100 was generated and the corresponding input/output pair was presented for learning. In all cases, a total of 50,000 total presentations (equivalent of 500 sweeps through 100 data pairs) was sufficient to produce essentially constant weights.

7.2.3. Results

The first problem encountered was how to scale the two continuous real valued inputs between zero and one. The reflux ratio and vapor boilup to bottoms flow ratio can theoretically range from zero to infinity. The first attempt at scaling was one which the author thought to be appropriate. The ratios were scaled as $\frac{R}{R+1}$ resulting in compression of the zero to infinity range into zero to one, with greater resolution for ratios close to one, and less resolution for values much smaller and much larger than one. This scaling was found to be inadequate, and the network with 25 hidden nodes was unable to learn all 100 input/output pairs. Examination of the cases when the net failed to find the

Table 7.1: Neural Net Input Interpretation

Input Node	Value	Interpretation
1.	0.0-1.0	Scaled reflux ratio
2.	0.0-1.0	Scaled vapor boilup to bottom flow ratio
3.	1.0	Pressure controlled by reflux flow (flooded condenser)
4.	1.0	Pressure controlled by distillate flow (flooded condenser)
5.	1.0	Pressure controlled by vapor boilup (heat input)
6.	1.0	Pressure controlled by vapor bypass around condenser
7.	1.0	Pressure controlled by coolant flow to condenser
8.	1.0	Condenser is used flooded
9.	1.0	Condenser is not flooded
11.	1.0	Equipment configuration includes vapor bypass around condenser
10.	1.0	Equipment does not include an accumulator
12.	1.0	Equipment includes an accumulator
13.	1.0	Distillate is a vapor
14.	1.0	Distillate is a liquid
15.	1.0	Vapor ratio divided by reflux ratio and scaled

correct result showed the problem to arise from the inability to compare the reflux ratio and vapor boilup to bottom flow ratio as DICODE does with rules.

Table 7.2: Neural Net Output Interpretation

Output Node	Value	Interpretation
1.	0.5-1.0	Reflux drum level controlled by reflux flow
2.	0.5-1.0	Reflux drum level controlled by distillate flow
3.	0.5-1.0	Reflux drum level controlled by coolant
4.	0.5-1.0	Column base level controlled by bottom flow
5.	0.5-1.0	Column base level controlled by vapor boilup
6.	0.5-1.0	Distillate composition controlled by reflux flow
7.	0.5-1.0	Distillate composition controlled by distillate flow
8.	0.5-1.0	Distillate composition controlled by vapor boilup
9.	0.5-1.0	Bottom composition controlled by vapor boilup
10.	0.5-1.0	Bottom composition controlled by bottom flow

Next, a 15^{th} input was created as the ratio of reflux ratio and vapor boilup to bottom flow ratio (before scaling) and then scaled in the same way as the reflux ratio and vapor boilup to bottom flow ratio. Again, the net with 25 hidden nodes was unable to learn all of the 100 input/output pairs.

After a bit of consideration, the apparent cause of the problem was discovered. DICODE compares the reflux ratio and vapor boilup to bottom flow ratios linearly using rules. Furthermore, comparisons are made relative to 5.0 for reflux ratio and 10.0 for vapor boilup to bottom flow ratio. Therefore, the neural net was unable to reproduce a linear comparison after the values were

scaled nonlinearly. Next, the reflux ratio was divided by 5 and the vapor boilup to bottom flow ratio was divided by 10, scaled as $\frac{R}{R+1}$ and the net was run again. This time, the neural net was able to successfully learn all 100 input/output pairs with 15 hidden nodes. Unfortunately, the neural net required that the inputs be scaled based on a knowledge of the rules to be reproduced. Since knowledge of the rules used to design a control system would not be available for the intended use (learning the design from examples), the above scaling method is unacceptable.

After further consideration, one final scaling was attempted. The reflux ratio for most columns would be in the range of 2 to 8 with a very few greater than 10. Also, the vapor boilup to bottom flow ratio would be in the range of 2 to 15 with a very few greater than 20. For very large reflux ratios, say greater than 20, the reflux ratio is for all practical purposes 20 (very large). Similarly, for vapor boilup to bottom flow ratios very large, say greater than 50, the ratio is for all practical purposes 50 (very large). Therefore, the reflux and vapor ratios were scaled linearly with a cutoff at 20 and 50 respectively. Any reflux ratio greater than 20 was considered 20, and any vapor ratio greater than 50 was considered 50. Also, the ratio of the two was taken (before cutoff) and scaled linearly from zero to 2. Thus, a reflux ratio of 20 became an input of 1.0, vapor ratio of 50 or greater became 1.0, reflux ratio over vapor ratio of 2 or greater became 1.0.

The linear scaling with cutoffs at 20, 50 and 2 for reflux ratio, vapor ratio,

and ratio of the two respectively produced astounding results. A neural net with four hidden nodes was able to learn all 100 input/output pairs. In fact, originally, the criterion for obtaining the correct result was as follows. The output vector was divided into four groups. The first three outputs represented the manipulated variables considered for reflux drum level control, the next two were those considered for column base level, the next three were distillate composition control, and the last two were bottom composition control. Within all groups except the first, the manipulated variable associated with the largest output of the group was interpreted as the one to be paired with the controlled variable associated with that group. The first group had the further requirement that the largest value be greater than 0.5 since a correct answer could in fact be no controller for reflux drum level (internal condenser, or no accumulator). The criteria described above were found to be overly tolerant. With the linear scaling and four hidden nodes, the square of the euclidean distance between desired and actual net output was less than 0.1 in all cases.

Next, the same linear scaling was used, but the ratio of reflux ratio and vapor ratio was not provided as an input. In this case, the net required a total of five hidden layer nodes in order to learn all 100 input/output pairs. Table 7.3 shows a summary of the different net topologies investigated in the course of solving a portion of the regulatory control design problem.

The success of the net should be measured as its ability is to interpolate between the cases on which it was trained. In order to test this ability, several

Table 7.3: Neural Net Topologies Used

Net Number	# Input Nodes	# Hidden Nodes	# I/O pairs Correct of 100 in Learning Set
1.	14	15	68
2.	14	20	79
3.	14	25	83
4.	15	15	82
5.	15	20	85
6.	15	25	83
7.	15	15	100
8.	15	3	98
9.	15	4	100
10.	15	5	100
11.	15	10	100
12.	15	15	100
13.	14	4	99
14.	14	5	100

1 - 3: scale reflux ratio and vapor ratio as ratio / (ratio + 1)

4 - 6: scale reflux ratio, vapor ratio and (vapor ratio)/(reflux ratio) as ratio / (ratio + 1)

7: scale reflux ratio as (rr/5)/(rr/5+1), vapor ratio as (vr/10)/(vr/10+1), and vr/rr as (0.5*vr/rr)/(0.5*vr/rr+1)

8 -12: scale reflux ratio 0-20 to 0-1, vapor ratio 0-50 to 0-1, vr/rr 0-2 to 0-1 with cutoffs above 20, 50, and 2 respectively

13-14: scale reflux ratio and vapor ratio as in 8-12, don't use vr/rr

cases were run with combinations of reflux ratio and vapor ratio in the range over which the linear scaling was used. In these tests, values of reflux ratio ranged from 1.0 to 15.0 in increments of 1.0, and vapor boilup to bottom flow ratio ranged from 2.0 to 30.0 in increments of 2.0. The other inputs were fixed at values representing a binary distillation column with a condenser used "normally" (not flooded and not internal), the pressure is controlled by coolant flow to the condenser, the distillate is in the liquid form, and the feed flow is manipulated by another process.

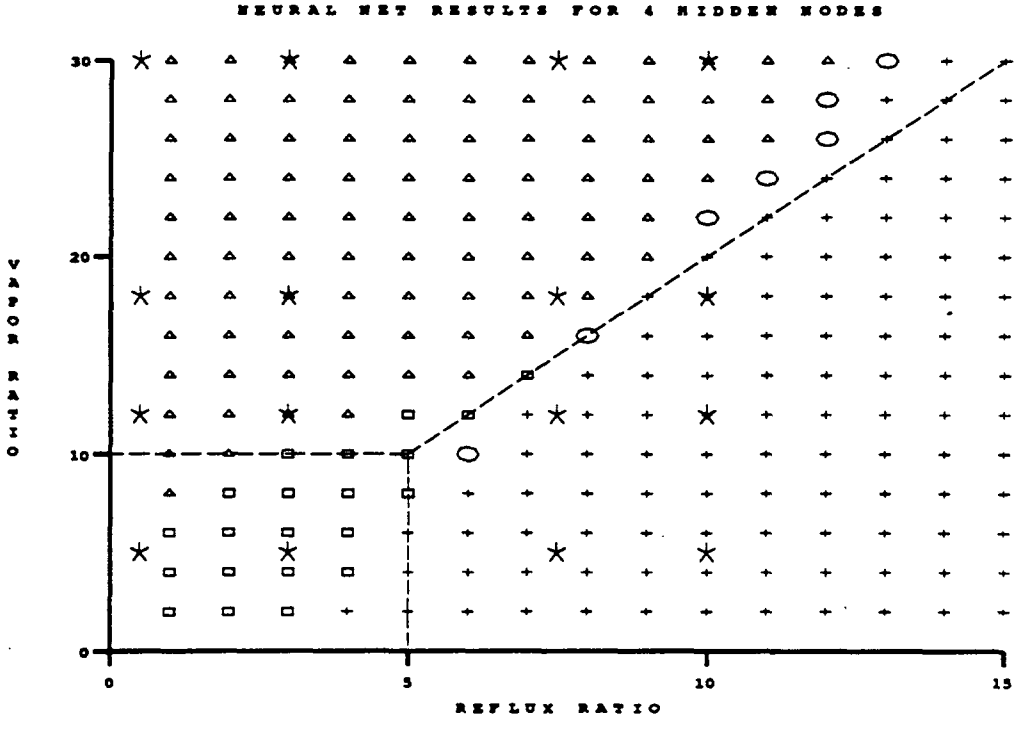


Figure 7.6. Neural net results for 4 hidden nodes. Hollow boxes indicate the DB level control loop pairing, crosses indicate LB, triangles indicate DV, and circles indicate a pairing was not obtained.

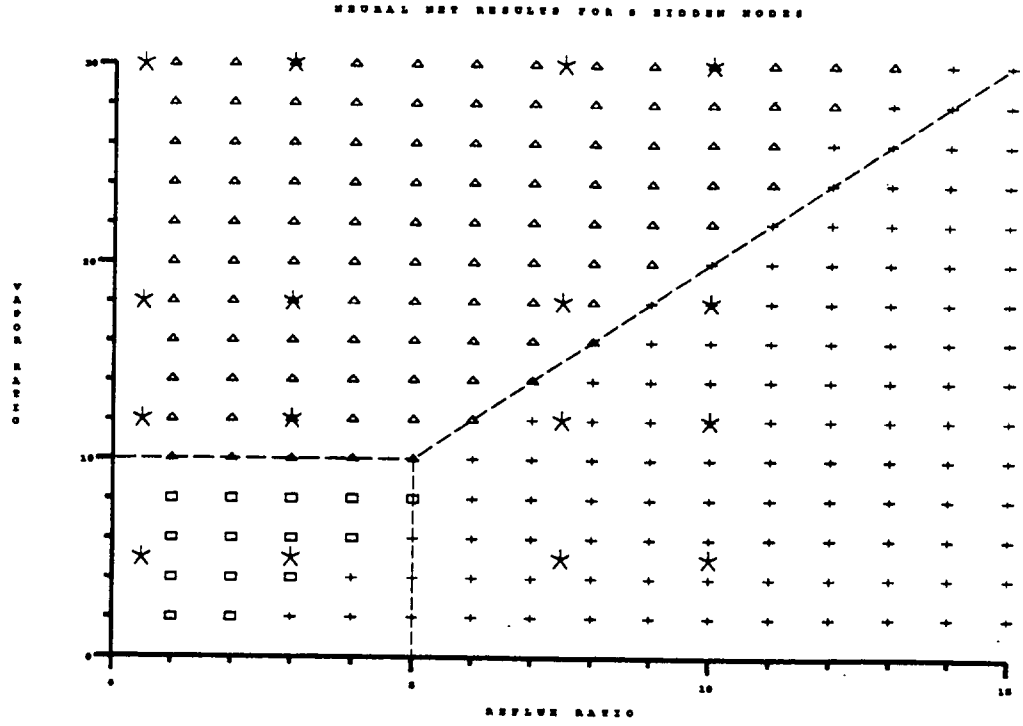


Figure 7.7. Neural net results for 5 hidden nodes.

Figures 7.6 through 7.8 show the results of three tests. The dashed lines divide the test region into three partitions indicating the three level control variable pairings considered by DICODE. The divisions are provided by the rules in DICODE and represented in Table 2.5 which is repeated here as Table 7.4. The area closest to the origin represents the region of reflux ratio and vapor boilup to bottom flow ratio where DICODE pairs the reflux drum level control with the distillate flow and the column base level with bottom flow (call it DB). The region of reflux ratio greater than 5.0 and below the diagonal dashed line is the region in which DICODE pairs the reflux drum level with reflux flow and

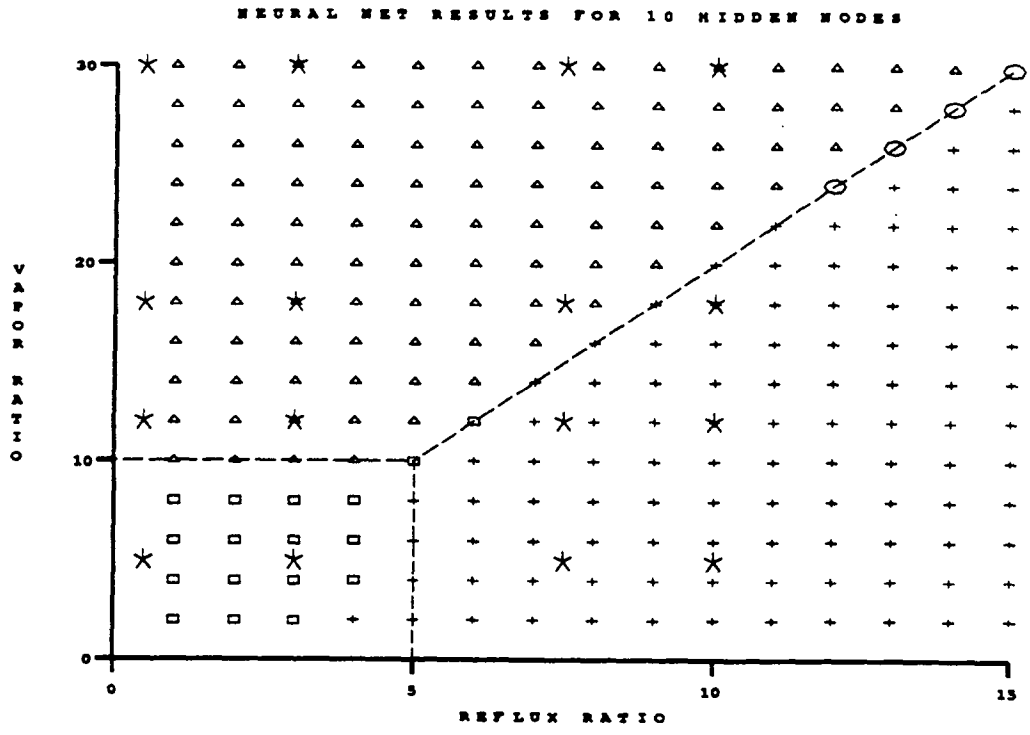


Figure 7.8. Neural net results for 10 hidden nodes.

column base level with bottom flow (call this one LB). The third region of vapor ratio greater than 10.0 and above the diagonal dashed line indicates the area in which DICODE pairs the reflux drum level with distillate flow and column base level with vapor boilup (DV).

In Figures 7.6 through 7.8, the empty boxes indicate instances where the neural net obtained a pairing of DB, the crosses indicate a pairing of LB, and the triangles indicate a neural net result of DV. Note also the presence of circles in Figures 7.6 and 7.8. The circles indicate instances where the reflux drum level was not paired with a manipulated variable by the neural net as defined by a

Table 7.4 Material balance control variable pairings.

V/B \ L/D	<5	5 ≤L/D<10	≥10
<10	DB	LB	LB
10≤V/B<20	DV	DV \ LB	LB
≥20	DV	DV	$DV^{t} \setminus LB^{tt}$

The diagonals with two entries indicate that a comparison of L/D and (V/B)/2 must be made. The pairing on the left is recommened when (V/B)/2 > L/D, and the pairing on the right is recommended for the opposite case.

[t]Overrides must be provided for accumulator level.
[tt]Overrides must be provided for column base level.

value of 0.5 or greater for one of the three manipulated variables considered. The stars represent the values of reflux ratio and vapor ratio for input/output pairs used to train the net. In Figure 7.7, no circles appear, indicating that the net with 5 hidden nodes never failed to find a manipulated variable for the reflux drum level control in the cases tested. However, note also that the net with 5 hidden nodes performed more poorly than the other two in matching the boundaries set by DICODE. The failure to match the boundaries should not be

considered a failure of the net to interpolate, but rather results from the fact that the training set was not concentrated around the boundaries but rather distributed somewhat equally throughout the region considered.

In examining the results presented in Figures 7.6 through 7.8, it must be recognized that in order to more clearly determine the boundaries set by the net, a finer grid of reflux ratio and vapor ratio must be used. However, in training the nets, no attempt was made to concentrate the selected points of reflux ratio and vapor ratio near the boundaries. If the training set did concentrate more closely around the boundaries, better performance as measured by reproduction of the boundaries could be expected. In the case where the net is trained on actual existing designs, it would not be possible to concentrate the training set data in areas where boundaries exist. Furthermore the position and even existence of the boundaries would not necessarily be known.

7.3. Summary of Results

Neural nets have been heralded as the up and coming solution to many problems. As with most new techniques for problem solving, the proof is in applications. The application of neural nets to the problem outlined above was complicated by the need to scale inputs between zero and one. The scaling problem is one which can consume considerable time and effort, particularly if the significance of the inputs is not known. In the case presented above, the

rules which the neural net was to reproduce were known, but in the general case little may be known as to the significance of the values and how they should be scaled. With respect to the scaling problem, the application of neural nets is quite similar to analog computing.

It must be realized that neural nets are simply a way of modeling an input/output relationship in which the model is fixed in the form of densely interconnected processing elements with nonlinear behavior. The learning method is simply the way in which the weights are changed to make the model fit the data. In this respect, neural nets are nothing new, and in fact are deficient in one very important aspect from an engineering point of view. The weights represent the parameters which are used to fit the model to data, yet in themselves, the parameters mean very little to the problem. It is extremely difficult to interpret the meaning of the weight values in order to gain some insight into the process being modeled. Furthermore, the neural net is not guaranteed to learn anything other than the examples with which it is presented during learning. Reasonably accurate interpolation can be expected, but accurate extrapolation should not be expected.

In the neural net results presented in the previous section, the ability of the net to interpolate between its training examples was found to be somewhat lacking. Even though the net learned input/output pairs exactly, the net was unable to provide a correct answer for cases close to the boundaries set by the rules in DICODE.

One final and unexpected result obtained through this application of neural nets is a demonstration of the power of an expert system to handle problems with a combinatorial nature. As mentioned in the section outlining the problem to be solved by the neural net, twenty rules reduced the total possible combinations of input from 100,000 to 800. In order to solve the problem which includes the knowledge of those twenty rules, a total of 27 input nodes would be required, and 23 output nodes. A total of 100,000 possible combinations of input would be required to cover the input space. The problem solved consisted of 100 different input/output pairs for which a total of 15 input nodes, 10 output nodes, and 4 hidden nodes were required.

CHAPTER 8

CONCLUSIONS

Although expert systems and AI programming techniques have been in existence for many years, application of those techniques to deep knowledge problems prevalent in the engineering fields has developed slowly. The reason for the slow development of deep knowledge applications is that rules often do not adequately represent the knowledge. Early expert systems solved the shallow knowledge problems for which rules could adequately represent the expert's knowledge. The advent of object oriented programming environments, frame based knowledge systems, and access oriented programming has provided the tools needed to solve deep knowledge problems with AI techniques. Unfortunately, providing the tools is not enough in order to support application of expert systems to deep knowledge problems. One of the primary objectives of the research described in this dissertation was to give some guidance for using the facilities provided by expert system shells for the purposes of solving deep knowledge type problems. Specifically, the design of control systems was chosen

Conclusions based on the research described in this dissertation can be drawn in a number of areas. Engineering problems are typically different than most other problems due to the deep nature of the knowledge to be represented in the expert system. That deep knowledge has structure which can be exploited during expert system construction if the structure is understood. The representation of that knowledge in a structured model is useful if not necessary in order to create an expert system.

8.1. Distillation Control Design With an Expert System

Chapter 1 describes the incentive for applying expert systems to the control design problem. However, the deep nature of the knowledge to be represented created some difficulties. Idiomatic control was found to be a structured method for control system design, and the goal tree success tree (GTST) model was found to represent that method as an intermediated before creation of the expert system. Idiomatic control analysis and synthesis is described in Chapter 2, and the GTST model is presented in Chapter 3. The result of using idiomatic control, representing it in a GTST, and then entering that GTST structure directly into the knowledge base of an expert system has resulted in a specific methodology which can be used to create an expert system in which the knowledge is easier to understand, debug, modify, and extend. The methodology involves the following steps.

1) Use the idiomatic control analysis procedure to define the idioms needed to solve the problem.

2) Use the idiomatic control design procedure to define the design problem.

3) Represent the idiomatic control design procedure in a GTST.

4) Create the expert system by

   a) creating frames to represent the goals,

   b) creating frames to represent controllers,

   c) creating frames to represent controlled variables,

   d) creating frames to represent manipulated variables,

   e) creating rules to link goals,

   f) creating rules to satisfy lowest goals with success paths,

   g) creating rules to represent the expert's knowledge of when to create controllers linking manipulated and controlled variables, and

   h) creating facets on slots of frames to implement a necessary constraint on the regulatory control design (see Chapter 5, section 5.3.4).

Implementation of the methodology to create an expert system for designing control systems for distillation columns is described in Chapter 4. The DICODE knowledge base and its functioning is described in detail. Creation of the DICODE knowledge base has resulted in some generalizations in the form of specific guidelines for rule and frame structure. The exact representation of the goal tree as frames with needed attributes and rules to link them is outlined in detail in Chapter 5. The generalizations are extended to create a customization

of the expert system shell which facilitates construction of expert systems for the purpose of designing controls for any process. The customization of a shell for the purposes of provided necessary facilities for the control design problem is described in Chapter 6.

8.2. Application of Neural Nets to the Control Design Problem

All of the knowledge for control design has been given structure by the methodology except the expert dependent knowledge. The only knowledge required which has not been cast into a specific organization is that knowledge used to determine under what specific conditions a controlled variable should be paired with a specific manipulated variable. Some conclusions can be drawn about the form of rules which are expert dependent, although specific structure such as the GTST is difficult to define. The poor definition of structure for expert dependent knowledge lead to the application of neural nets to represent part of the control design problem.

Conclusions concerning the ability of neural nets to learn the regulatory control design process include comments the scaling of input values and about covering the input space. Proper scaling of the input values to the neural net was critical. In Chapter 7, the problem discussed concerned a net with 15 input nodes and 10 output nodes. If the inputs were not scaled properly, 25 hidden nodes resulted in the net learning only 83 of the 100 input/output data pairs

in the learning set. With proper scaling, a net with 4 hidden layer nodes was able to learn all 100 of the input/output data pairs in the learning set. Correct scaling was found to be the simplest possible, but perhaps not the most logical.

The successful net was found to be somewhat lacking in ability to interpolate between points on which it was trained. The net roughly approximated the regional partitioning implemented by the rules of DICODE. Failure of the net to conform exactly to the partitioning created by the rules of DICODE can be attributed to poor coverage of the range of continuous inputs in order to limit the total number of input/output pairs in the learning set.

8.3. The Combinatorial Nature of Design

One of the claims of supporters of expert systems is their ability to handle problems with a combinatorial nature. One of the unexpected results of the research was a demonstration of that ability. The expert system DICODE handles input resulting in a possible 100,000 combinations or more just for the regulatory control design problem. The number of rules associated with the logic of the basic regulatory design is about 70. This represents the rules associated with the GTST and the expert dependent knowledge.

When neural nets were applied to the regulatory control design problem, It was found that a net which could learn those 100,000 combinations of input would be far too large and take far too long to learn the input/output data for

the purpose intended. In fact, a net which could handle only 100 combinations of input represented by 14 or 15 input nodes, with 25 hidden nodes and 10 output nodes consumed over 8 hours of cpu time on a microVax. The successful net with only 4 hidden layer nodes required approximately 20 minutes of cpu time resulting in 100 weights to be modified for learning. This net was unable to produce acceptable results when interpolating between input data in the learning set.

8.4. Recommendations for Future Work

The results of expert systems for control design presented in this dissertation represent a first step toward applications to design problems in other engineering disciplines or even engineering problems in general. Providing expert system shells is equivalent to providing any other programming language such as FORTRAN. Yet no guidelines for programming in expert system shells have been provided. There is no equivalent to standard FORTRAN or modular programming methods for using expert system shells. More work in the area of providing guidelines and generalizations for specific applications is needed.

The use of neural nets to solve some problems can be successful. However, the best area for application of neural nets is to model problems or processes about which almost nothing is known except the inputs and outputs in the form of data. Any process which must be considered a "black box" represents a good

possibility for application of neural nets. The neural net is essentially a model with a fixed structure and many parameters which are used to fit the model to data. The problem with a neural net model of a process is that it is difficult to obtain any insight into the process based on the values of the parameters which are used to fit the model to the process.

For most engineering problems, some kind of model can be acquired based on first principles. A model based on first principles is far superior to the "black box" approach unless it is unable to approximate the input/output relationship. Unlike neural nets, the parameters used to fit the model to the process have meaning due to the first principles approach to model development. Furthermore, the model is usually better able to extrapolate outside the region used to fit the model to the process. More research into the application of neural nets and parallel distributed processing to chemical engineering problems is necessary to define the areas in which they can be applied with reasonable success.

APPENDIX A

GRAPHICS, OBJECTS, AND IDIOMS

During the research presented in this dissertation, some interesting results were obtained which are not directly related to the goals of the project. One of these results was the relationship between the graphics used to present the control design, the objects or frames used to represent the graphic entities within the expert system, and the definition of control idioms within idiomatic control analysis and synthesis. The following discussion describes the analogous use of graphic images, frames, and idioms in the graphic interface for the expert system, the expert system itself, and idiomatic control respectively.

A.1. GRAPHICS

One of the most commonly occurring difficulties encountered when designing graphics for applications such as DICODE, is the allowable graphics space. Many times, organization of the graphical objects is critical in representing the

entire picture. The space availability problem is one which was encountered early in the implementation of graphics to accompany the DICODE expert system package. Another consideration is the number of different graphical pictures necessary to represent all possible combinations of equipment and control systems. Certainly, every combination cannot be considered separately in defining graphical representations for the equipment and control systems.

One solution to both of the aforementioned problems is to divide the graphics space into areas within which specific graphical objects will reside. For instance, for the overhead system, a specific area is defined in which all condensers will be drawn, regardless of type. Sensors are likewise given specific areas within which they will reside. In order to effectively use this modular form of picture building, all connection points, that is streams and signal paths, must have the same location at the border of the allotted space. The modular method solves the problem of effectively using the available graphics space while solving the combinatorial problem as well. The result is a graphics system which can provide every combination of equipment and control system with a minimum of different pictures while ensuring that space will be available for each piece of the picture.

Defining the control loops (connecting sensors and actuators) is easily accomplished once the connection points are established for sensors and actuators. The only remaining problem is that the signal paths may sometimes have to cross other signal paths and streams. Those streams and signal paths may or

may not be present depending on the equipment and control system design. Part of the crossing paths problem may be solved by allotting space for control signal paths between areas designated for equipment and sensors. The signal paths will have stream crossings at predefined places. The crossing control signals remains a problem.

Three distinct types of control loops exist in the control system design. The three types are primary, secondary and constraint or override controls. The primary control loops are those for material balance, composition, pressure, and condensate temperature control. The discussion in the previous paragraph refers to the primary control loops. The secondary control loops are those which act as inner loops of cascade arrangements. The secondary control loops include flow controllers and BTU controllers which receive set points from the primary control loops. The secondary control loops are simpler to implement graphically than the primary control loops because they are confined to a small area close to a valve which manipulates the flow of a stream. Space considerations amount to ensuring space is available for a flow or BTU controller at all appropriate valve locations. Since secondary control loops are not absolutely necessary, they should be modular in nature, similar to the equipment. If they are not present, the primary control loops should connect with the actuator directly.

The constraint controls should also be modular in the same way as the secondary control loops. In the case of the absence of either a secondary or constraint controller, an analogous picture which connects the actuator (valve)

or other control element (in the constraint control absence) to the defined connecting point must be substituted.

The result of the modular design of the graphics support is a flexible system such that a new piece of equipment may be provided to fit into the defined area. The number of actual pictures which must be predefined is reduced dramatically, and the area available is used effectively. The only existing problem is the crossing signal paths. The solution would be to check whether a line exists where the one being drawn will go, and automatically leave a space on either side of the existing signal path. This solution requires graphics capabilities which were not available for the DICODE implementation.

A.2. OBJECTS

In the graphics section above, areas were designated for specific pieces of equipment. The pieces of equipment or sensors which can occupy the allotted spaces are referred to as graphical objects. A natural result of the modular representation of the graphical objects is the representation of the equipment, sensors and control loops as objects or frames within the expert system. One advantage of this representation is that attributes can be attached to the objects. Examples are condenser type, control loop input and output, and descriptions of equipment and control loops. In this manner, a number of different objects of the same class may exist, and different instances of the same class can be of

a different type with a different description. Furthermore, the objects may be created or destroyed when needed. The creation and destruction of objects during expert system execution solves the problem of flexible designs where some equipment may not be necessary.

The graphics software (PLOT10 GKS) allows drawing of any picture by one executable from information in data files (metafiles), and multiple pictures may be drawn on the same screen. The graphics executable therefore requires the names of the files to be drawn on the screen. File names as instances of a class of objects within the expert system provides the required facilities. The reason for the representation of the files as objects is that the KES shell communicates with external routines through the operating system. Information must be written to a communication file which is read by the external. If necessary, the external writes another communication file to be read by the expert system. In the KES shell, the members of a class (or the instances of an object) may be addressed as a group to be written in the communications file.

A.3. IDIOMS

Control idioms are so called mini-inventions which can be inserted as modules during structured control system design (idiomatic control design). The flow and BTU controllers are typical control idioms which are used extensively to improve control characteristics of primary controls. The representation of

control loops as objects within the expert system and particularly the flow and BTU controllers as removable modular objects is identical to the idiomatic formulation of control systems. The representation as objects allows for definition of input and output to differentiate the instances of the generic idiom. A textual description can also be attached to describe the purpose of the idiom.

The representation of the control loops as objects is identical to the idiomatic formulation of controls. The modular nature of the graphics objects and control loops as objects within the expert system and the analogy to control idioms provides an intriguing common ground for control system design by expert systems.

APPENDIX B

DICODE: A HISTORY

The expert system DICODE was developed as a demonstration of the method proposed for constructing the knowledge base of an expert system in control design. Development of the method was the primary goal of the research described in the body of this dissertation. The expert system was developed along the classic lines. That is, the knowledge base author or knowledge engineer was familiar with knowledge representation techniques and expert system shells. The knowledge engineer had an overall familiarity with the distillation process and a background in chemical process control, but no experience in control system design.

The problem domain was chosen as a joint venture between the Department of Chemical and Nuclear Engineering at the University of Maryland, and E. I. du Pont de Nemours, Luviers Building, Newark Delaware. The knowledge engineer was a graduate student, and the experts were provided by DuPont. The project evolved through several iterations of knowledge acquisition, prototyping and validation by the expert. The remainder of this appendix outlines the specific

steps through which the project developed.

B.1. Preliminary Work

Research at the Department of Chemical Engineering at the University of Maryland in the area of expert systems applied to chemical process control began in the spring of 1986. Preliminary work in the area of representing knowledge for expert systems in control design showed promise of producing useful results. The preliminary work involved representing the idiomatic control design procedure in a goal tree success tree (GTST). For a discussion of idiomatic control and the GTST model, see Chapters 2 and 3. The GTST model has been used to model deep knowledge for applications of AI techniques primarily for fault diagnosis in the nuclear power industry.

Using the GTST model to represent a problem in the design domain was a new application. The primary motive for using the GTST was to organize the knowledge in a structure usable in an expert system yet representative of the design process. The definition of the GTST model provides some degree of flexibility such that goals can take priorities and the goal tree may even change based on changes in data during execution of the expert system. Overall, the GTST model was found to be ideally suited to representing the idiomatic control design method due to the hierarchical nature of the objectives defined during idiomatic control design.

B.2. Industrial Applications

After the preliminary work had been accomplished, a particular application was sought which would validate the method proposed for knowledge base construction. It was very important that the project be industrially oriented in order to provide validity from an industrial point of view, and be of value to the chemical industry. At this point, interaction with DuPont uncovered a common goal. One of DuPont's experts in control design was to retire at the end of 1987, and another engineer was given the task of capturing some of his knowledge in an expert system before the expert retired. By combining our resources, the University and DuPont could both attain our goals.

B.3. Project Definition

The first interviews were spent defining the project. The purpose was to capture some of the expert's knowledge in control design, with the realization that only a small portion could be considered. It was decided that the scope of the problem would be limited to the controls for the overhead portion of a binary distillation column. These controls included column pressure, reflux drum level, distillate composition, and condensate temperature control. The first prototype would be used as an indication of whether the project could succeed.

The purpose of the expert system was to guide the user through the design of the control system based on equipment design and control objectives. Extensive use of warnings and recommendations were to be used, along with explanations of questions and justifications of results.

The primary users of the expert system would be control design engineers. Their use of the system would be to obtain a first cut at the control system based on simple heuristics. Further validation of the design before implementation would be the responsibility of the user. Secondary audiences would include chemical engineering students and possibly plant operators. The educational use of the expert system was taken as a second priority.

Of particular importance to the audience was the form of the results. Most engineers think in pictures or diagrams. In particular, the control design engineer solves the control design problem with a diagram of the process and controls. The primary result of a session with the expert system was to be a graphical representation of equipment and control design in the form of a P/I diagram. Text justification of the equipment and controls were to accompany the diagram. A printout of both the graphics and the text were to be available for the user.

B.4. Interviews With the Expert

The interviews following project definition were for the purpose of obtaining

the expert's knowledge. Knowledge acquisition has always been a bottleneck in the creation of expert systems. The difficulties center around the way the expert uses his knowledge. In expert systems, the knowledge is typically represented at least in part by rules. Unfortunately, experts seldom use rules explicitly, rather they use what is referred to as compiled knowledge. For example, when someone is learning to drive a car with a manual transmission, he is told to watch the tachometer to determine when to shift. Later, when the individual has more experience, he may be able to tell when to shift based on the sound of the engine. Once the person has been driving for many years, he knows when to shift "instinctively". The "instinctive" behavior described here is an example of compiled knowledge. This kind of knowledge cannot be used for an expert system, therefore it is necessary to coerce the expert into providing his knowledge as though he were trying to teach someone how to solve the problem.

In the first few interviews with the expert, he was asked to verbalize what he knew about distillation control design. The knowledge he then provided was somewhat disjoint and incomplete. Although the information he provided in response to this question was not directly useful, it did provide enough information to determine that much of what he would relate was provided in a text of which he was a coauthor. The first prototype was then developed based on information in that text.

The first prototype was given the name DOCSYD for Distillation Overhead Control SYstem Design. This prototype provided no graphics result, giving

only a list of the recommended controllers and their manipulated variables. The primary purpose of this first prototype was to represent the knowledge obtained from the text and present the expert with the form of interaction with the system. During development of this prototype, several holes were found in the knowledge provided in the text. The fact that information was missing was not unexpected since the text was a reference rather than a step by step guide to designing controls for distillation. The expert was asked to fill in the missing information for the next iteration. The expert recognized the importance of the graphics at this point, and therefore the graphics interface was set as one of the next tasks.

Acquiring the missing information from the expert was found to be difficult for the reasons mentioned before. In order to obtain the information needed to fill the holes in the knowledge base, the expert was presented with some example problems. Several distillation columns were presented to the expert for which he was to design the controls. The examples were taken from literature and from DuPont processes. The expert was familiar with most of the columns with which he was presented. Typically, he asked for the location of the column in order to determine which one it was, since he had probably designed the control system for it some time in the past. Even if the example was not one he had seen before, he often remembered a column which was very similar, and again did not apply the design procedure, but used a design he had previously developed. After finally presenting some examples which were somewhat unique, his design

procedure became more clear.

Although the examples were often not realistic, they did force the expert to apply his design method. His design method could be described in the following way. The reflux drum level control and distillate composition control loops could be obtained by using the reflux ratio. From DuPont's experience, the reflux drum level control is of higher priority than composition control. Therefore, the reflux ratio is used to determine which of the two flows reflux or distillate will be used for level control. If the reflux ratio is greater than 5, then the reflux flow is used for level control. If the reflux ratio is less than 5, then the distillate flow is used. The distillate composition control loop then manipulates the other flow. Using these rules ensures that the flow used for reflux drum level will be at least 20% of the total flow out of the drum. 20% of the total flow out of the drum should be adequate for controlling the level under most conditions, resulting in less frequent overrides of the composition control. The method of pressure and condensate temperature control was clearly presented in the text, and therefore little input from the expert was necessary.

Further iterations were necessary to provide adequate and appropriate graphics and text justifications for the system. Also, recommendations and warnings were necessary in the opinion of the expert. The graphics was provided by PLOT10 GKS from Tektronix Inc. as a FORTRAN executable. For a description of the graphics development, see Appendix A. It should be noted that up to this point in the development of DOCSYD, the shell provided no

facilities for forward chaining or access oriented programming. As a result, the goal tree was not directly represented in the knowledge base, and the procedural part of the knowledge base was complicated by the need for patch fixes to implement many of the required functions such as recommendations and warnings.

The prototype DOCSYD was essentially complete after only a few iterations through the process of knowledge acquisition, knowledge base modification, and verification by the expert. The next question was where to go from this point.

B.5. Extension of the Prototype

Extensions of the prototype expert system included enlarging the scope to include design of the controls for the entire column based on the same expert's knowledge. Further extensions included using other measures for control system design such as the RGA, RDG, and an estimate of the integrated error due to a disturbance.

B.5.1. Controls for Entire Column

The logical next step was to extend the DOCSYD knowledge base to include the control design for an entire distillation column. Rather than modifying

the existing knowledge base, two more expert systems were developed. One created the design for the column base controls and the other tied the overhead and column base expert systems together. The expert system tying the others together provided the level and composition control loops without specifics of the equipment design. It then allowed use of the overhead and column base expert systems to create a more detailed design. Results were provided in the form of a graphics representation and text description of the design for each of the three parts.

Using three expert systems to provide the design was found to have two major flaws. From a functional perspective, the three part expert system was found to be impractical because of the interaction between the overhead and column base design procedures. It was not possible to perform each piece of the design independently. Conflicts arose which could not be resolved without performing the design for the entire column. From a users' perspective, the three part expert system was unacceptable due to the time required to load the knowledge base for the overhead or column base part of the design. The three parts were combined into one expert system solving both of these problems.

The rules for creating the control system for the overhead part of the column remained much the same, although the interaction between the overhead and column base designs had to be considered. The rules for the column base controls were very similar to those used for the overhead section. The vapor boilup to bottom flow ratio was used much like the reflux ratio. If the vapor

boilup to bottom flow ratio was greater than 10, the vapor boilup was used for column base level control. If the vapor boilup to bottom flow ratio was less than 10, then the bottom flow was used for column base level control. Interaction occurs when the reflux ratio is greater than 5, and the vapor boilup to bottom flow ratio is greater than 10. In this case, the rules would suggest using reflux flow for reflux drum level control, and vapor boilup for column base level control. This combination of using both internal flows for material balance control violates the overall material balance of the column. To solve this problem, the reflux ratio and vapor boilup to bottom flow ratio were compared to determine which should take priority and use an internal flow. Thus, the vapor boilup to bottom flow ratio was divided by two, and then the larger of reflux ratio and scaled vapor boilup to bottom flow ratio was determined as the one to use the internal flow. The other material balance controller was forced to use a manipulated variable which was less than ideal (from a stream size point of view) for level control.

Once the expert system was extended to include design of the controls for the entire distillation column, the name was changed to DICODE for DIstillation COntrol Design Expert. The three graphic options remain, as well as two save text options. Another modification came as a result of a new release of the expert system shell which provided demons. The use of demons provided access oriented programming facilities which allowed the goal tree to be represented within the knowledge base and removed many of the patch fixes needed in the

actions section of the knowledge base. The expert system finally fulfilled the expectations of those involved in its creation, including the expert.

At this point, DICODE was a functioning expert system based on the knowledge of a leading expert in distillation control design. The expert system was distributed to the members of the Industrial Consortium for the purposes of evaluation. It was hoped that the evaluations would provide areas for improvement of the system from different perspectives. In the mean time, extension of the expert system continued.

B.5.2. Other Measures

At this point, the development of DICODE diverges from the use of the DuPont expert's knowledge, to using knowledge published by others in the field of distillation control design. The purpose of this modification was to determine whether the same framework could be used to effectively represent the knowledge of another expert and furthermore represent the knowledge of more than one expert in the same expert system. In order to include another expert's knowledge, it was necessary to include the use of relative gain (RGA), relative disturbance gain (RDG), and an estimate of the relative integrated error resulting from a disturbance to determine the control system design. Each of these is a measure of control effectiveness for a particular pairing of composition control loops. As such, the basic premise of material balance control priority had

to be modified. The material balance priority was transformed to composition control priority such that the same results were obtained if the user specified that the relative stream sizes were to be used as the basis for material balance control. The user was also given the option of considering the RGA, RDG, and integrated error (IE) based on composition control priority. In fact, the reflux ratio and vapor boilup to bottom flow ratio were considered measures identical in function to the RGA, RDG, and IE although they were the same for all composition control pairings.

The reflux ratio, vapor boilup to bottom flow ratio, RGA, and RDG are all used as *tools* in a similar way to determine the best control system. For each of the *tools*, rules compare the value of a tool associated with a particular composition control pairing to values considered to be acceptable. If the value is not within a specified acceptable range, then the pairing is removed from those considered valid. Each tool is used in this way, removing unacceptable pairings from those available. If no possible pairing survives these tests, then the range of values considered acceptable is increased. This process continues until at least one pairing survives the tests. More than one pairing may survive, so the pairing to be presented as the best control design is then selected as the one surviving with the smallest value of IE. The initial range of values considered acceptable and the rate at which that range is expanded can be modified by the user, although two default settings are provided. One provides settings to consider only reflux ratio and vapor boilup to bottom flow ratio, the other

provides the settings to consider all of the measures. A third option allows the user to provide the parameters to use the measures in any way he would like by providing tuning parameters in a data file to be read by the expert system. Modifying the use of the measures by providing this data file is discussed in Appendix C.

In order to obtain the values of RGA, RDG, and IE, a steady state model of the process must be fit to data provided by the user. A model using Eduljee's equation, [Eduljee, 1975] is used to relate the tower operation to the properties of the material being separated. The Fenske equation [Fenske, 1932] is used to calculate the minimum number of trays required for the separation. Both of these equations assume a constant relative volatility. Furthermore, constant molal overflow is assumed. The model is fit to design information provided by the user through a one dimensional search for the relative volatility which provides the desired separation. This model is then differentiated analytically to obtain the RGA, RDG, and IE for the various configurations. For a further discussion of the model used and its differentiation, the reader is referred to [McAvoy, 1983].

B.5.3. Differences in Philosophy

The two default tuning sets provided by DICODE represent two different control design philosophies. Consideration of only the relative stream sizes (re-

flux ratio, vapor boilup to bottom flow ratio) to determine the level control loops represents a philosophy promoted by P.S. Buckley and others [Buckley, et .al. 1984]. Consideration of primarily RGA to determine the composition control loops represents a different philosophy promoted by F.G. Shinskey [Shinskey, 1986]. These two philosophies in fact provide a different answer to the same problem.

The reason for considering primarily relative stream sizes is to ensure that the reflux drum level and column base level seldom require the use of override controls. That is, if the stream provided as the manipulated variable for level control is at least 10% of the total flow out of the vessel, then the override controls will be used less frequently. The priority here is to remain in a normal operating condition more of the time by providing a larger operating envelope.

A disadvantage of this design philosophy is that it may provide a control design which demonstrates severe interaction between the composition control loops resulting in poor quality control of the products. However, if only one of the two products is important, then the composition control loop for the other product can be detuned or replaced with a slow acting optimizing controller. Detuning or providing the optimizing controller effectively eliminates the interaction between the composition control loops by making one much slower than the other. If most of the columns with which the authors of [Buckley, et. al., 1984] are familiar have only one important product, this could explain the reason for using the relative stream sizes. Another possibility is that they sim-

ply decide to provide a decoupling controller for the compositions if significant interaction is present.

The reason for considering RGA as the primary tool for deciding upon the control system is to avoid possible interaction between the composition control loops. A control system which minimizes interaction will allow tighter control of the product compositions, but can result in a normal operating envelope considerably smaller than another control system which demonstrates more interaction. The smaller operating window can result in frequent excursions into the constrained operating region, where composition control must be sacrificed in order to maintain levels in the reflux drum and column base. However, if the column is run continuously, very close to design conditions, a design based on RGA can provide much better control than one based on relative stream sizes.

Each design philosophy has its advantages and disadvantages. Therefore, it is desirable to consider all possible measures of control adequacy to determine the best possible control system. The problem with considering all of the measures is in determining how to use each measure appropriately, weighing the advantages and disadvantages of a control system based on the measures. Since each expert has his own ideas about the relative importance of these measures, DICODE provides the user with the ability to change the way it uses the measures. DICODE also provides two default tunings for the user. For a discussion of how to modify DICODE's use of these measures, see Appendix C.

B.6. Present State of DICODE

Presently, DICODE is a functioning expert system which can provide a control system design for binary distillation columns based on a variety of control objectives and control effectiveness measures. DICODE currently uses a simple steady state model to evaluate the measures but has set the precedent for integration of other possibly more accurate and computationally intensive calculations. The control configurations include all of those considered in [Shinskey, 1984] and those considered in [Buckley, 1983] as well as others.

DICODE also accepts a wide variety of equipment designs including seven different condenser types and five different reboiler types. Results of a design session include three different graphics representations and two text descriptions of the equipment and controls.

The logic of DICODE does contain some bugs. A number have been corrected through extensive testing, but it is virtually impossible to test each of the combinations of input since the possible combinations number in the hundreds of thousands. Further testing will continue in order to correct as many of these bugs as possible.

APPENDIX C

USING DICODE

DICODE requires the KES^{TM} PS runtime expert system shell in order to execute. Modification of the knowledge base requires the development version of the shell. Hardware requirements are a DEC VAX machine (MicroVax included) with the VMS operating system, and a VT240 or equivalent graphics terminal. A personal computer with software to emulate the VT240 can also be used but may require modification of one of the FORTRAN programs provided.

C.1. Overall Operation

DICODE exists as a parsed knowledge base created by using the development program **KESP.EXE** and the text file DICODE.KB. DICODE.KB is a text file referred to as an unparsed knowledge base. DICODE execution is begun by running the **KESR.EXE** program with DICODE as the argument. KESR.EXE reads the parsed knowledge base (DICODE.PKB) and provides the

inferencing process applied to the knowledge contained in the knowledge base. When properly installed, DICODE can be started by typing "KESR DICODE" from any directory in which the user has read and write privileges.

DICODE requires approximately one to two minutes to load into memory. A title is displayed, and the user can then begin to interact with DICODE. DICODE is completely menu driven, and the user interacts with DICODE by responding to questions with one of the options provided unless a string or number is requested. A design session is initiated when the user responds to a question indicating he would like to start a new design. DICODE then asks the user to provide information which describes the equipment, control objectives and the process material. Throughout this interaction, DICODE may provide warnings and recommendations based on the user's response to prompts for information. If an undesirable combination of input is provided by the user, DICODE will provide a warning, and then allow the user to change the inputs involved if he would like. At any time during the question and answer phase, the user can request an explanation of the question and the possible responses by responding to the question with 'e'. He can also request an explanation of the significance of the question by responding with 'w'.

During this question and answer period, DICODE may need to run an external program. In order to run external programs, DICODE writes communication data files in the user's default (current) directory. If an error message is displayed which indicates that a file could not be written, then the problem is

probably related to write privileges. The user should run DICODE from within a directory in which he has read and write privileges.

After responding to the questions which describe the column for which the controls are to be designed, the user is presented with a menu which allows him to obtain a number of forms of the result. Results include a graphical representation of the column, level controls, and composition controls which is not equipment specific. A list of the controllers with their controlled and manipulated variables may be obtained as well. The user can save the graphics or the text if he would like. Another option allows the user to continue to a more detailed description of the design. If this option is chosen, a graphic representation of the overhead section is available which is specific to the equipment design, along with a text description of the equipment and control designs. Also, a graphic representation of the column base equipment and controls is available. Any of the graphics or text may also be saved at this point.

The graphics and text are chosen through a menu system. The last item on any of these menus is to return to the previous menu, until the user again arrives at the main menu from which he can start a new design, view previously saved results, or quit.

Any time a graphics result is selected for viewing, DICODE writes a file containing the names of some data files. Each of these data files contains information to draw a small portion of the entire graphics image. DICODE then runs a FORTRAN program called DECODE which reads the names of the data

files, then reads the data files themselves and produces the graphics results. When the user strikes the return key, DECODE completes execution and control returns to DICODE. Any time the user requests text results, DICODE builds a long text string from appropriate small ones, and then displays the text string with breaks so that the text does not scroll off the screen. If the user requests to save graphics results, DICODE writes the file containing names of the necessary data files under the name provided by the user. Similarly, if the user requests to save the text, DICODE writes the text string to a file whose name is provided by the user.

C.2. Getting Started

The DICODE software must first be properly installed. The software is provided on a TK50 tape cartridge in the form of a backup save set. Installation instructions are provided along with the tape. If the instructions are followed, DICODE should run properly. The user must be using a DEC graphics terminal or a PC emulating one. After logging in, the user needs only to type "KESR DICODE", and DICODE will be loaded and begin execution.

If a PC is used to emulate a graphics terminal, it must switch from text mode to Tektronix 4010/4014 graphics mode to display graphics. It must then switch back to text mode after the user is done examining the graphical results. The switching is implemented using escape sequences which are sent from the

computer to the terminal or PC. If the emulation software requires a specific escape sequence, the ones currently provided may not be correct. If the emulation is unable to correctly switch back and forth, the FORTRAN source code DECODE.FOR may need to be changed to send the proper escape sequences. The DECODE.FOR code is documented in order to instruct the user how to modify it.

C.3. Input Attributes and Their Use

Throughout interaction with DICODE the user will be prompted for responses to questions, including menus. Most of the questions require the user to respond with one of the provided possible responses. By responding, the user is providing values for attributes which DICODE has determined to be necessary but is not able to infer from other information. Some of the questions require the user to respond with a number, the reflux for example ratio. Also, some of the questions require the user to respond with a text string, the name of a file in which to save results for example.

Any time the user is presented with possible responses from which to choose, he is forced to use one of them. In this case, the user may not know which response is appropriate for the current problem, or he may not understand the question. In this case, the user can respond with 'e' in order to get an explanation. The explanations displayed by DICODE are created as part

of the knowledge base by the knowledge base author. They attempt to explain the question and the possible responses so that the user can select the response which most closely represents the current problem. However, since these explanations are provided by the knowledge base author, they may not adequately describe the question and possible answers for all users.

The user may also obtain an explanation which indicates the significance of the response to the question. This explanation may be displayed by responding with 'w' to the question. The "why" explanation is also provided by the knowledge base author and gives an indication of how the attribute is used to obtain the design results. In DICODE, the "why" feature is used as an educational tool. By asking for these explanations, the user can determine the way in which the attributes are used to obtain the design results. If a "why" explanation is not provided by the knowledge base author, the KES shell will display a "why" explanation which indicates the rule, demon, or action which caused the system to prompt the user for input. If the information is necessary due to a long inference chain, KES will trace the inference back as far as possible, displaying the rules involved in the chain. The shell provided "why" explanation is very useful during debugging of the knowledge base.

C.4. The Menu System

The execution of DICODE is governed by a menu system which is imple-

mented in the *actions* section of the knowledge base. The menus are actually attributes whose values determine the actions to be executed. Each menu is defined in the *attributes* section with values to represent the options allowed by the menu. In the *actions* section, a menu is used by implementing a *while* loop. The while loop is executed as long as the attached condition is true. The last option in any menu is to exit to the previous menu, so the condition used for the while loop is *while* the attribute is not the value to exit the menu. Within the *while* loop, each of the menu options is implemented by an *if-then* construct. The *if* checks the attribute value for one of the possible ones, and the *then* part executes the actions appropriate for that value of the menu attribute.

Once the user is down several layers into the menu system, he must respond with the last option of each menu to extricate himself from the menus. The only exception to this extrication procedure is in the "DETAILED DESIGN SUBMENU". The last option returns the user to the "CONTROL DESIGN MENU" which is the menu immediately above the submenu, but the second to last option allows the user to return directly to the main menu from which he may quit or start a new design. If the user selects this option, the KES^{TM} shell displays some warning messages which the user should ignore..

C.5. Getting Results

The most important results which DICODE can produce are the graphical

representations of the equipment and control designs. Appendix A outlines the way in which DICODE obtains the proper pieces of the picture to be displayed. The important points to remember in obtaining the graphics results include proper installation of the system, proper use of DICODE, and hardware and software needs.

If the DICODE software or the KES shell are not properly installed, DICODE may not know where to look for the files needed for the graphics functions. If this is suspected as the problem, section C.7 may be helpful in determining exactly where the problem lies. The user must also run DICODE from a directory in which he has read and write privileges. DICODE communicates with external programs using communications data files which it writes in the user's default (current) directory. If the user does not have privileges to write in the default directory, then DICODE will be unable to function properly.

As mentioned in the section on getting started, DICODE uses an external FORTRAN program to generate the graphics result. One way to test the graphics without having to load DICODE repeatedly is to run DICODE once, and save the graphics result. Then, to display the saved results, type "RUN SHOGRAPH" and respond to the prompt for file name with the name of the file in which the results were saved. The program SHOGRAPH is identical to DECODE, except that the user must provide the name of the file in which the pieces of the picture are written. If changes need to be made to DECODE in order to obtain the graphics result, the changes can be made in SHOGRAPH until

the proper code is developed, and then the changes can be made to DECODE as well.

If DICODE is properly installed and used, the only problems which could exist concern compatibility of software and hardware. The program which produces the graphics image classifies the terminal as either a DEC graphics terminal, or unknown. If the terminal type is unknown, the program assumes it is a PC emulating a DEC VT240 or equivalent which switches from text to Tektronix 4010/4014 mode. The program then sends any necessary control or escape sequences to the terminal to perform the switches. Therefore, if graphics results cannot be obtained, then a compatibility problem exists between the software and the terminal. The most likely cause is an emulation program which does not behave exactly as a VT240. The emulation software documentation should provide the escape or control sequences necessary to perform the switches. The source code of DECODE.FOR and SHOGRAPH.FOR is documented to indicate where changes should be made to provide the correct switching.

C.6. External Program Use

The KES^{TM} shell provides the ability to pass commands to the operating system. DICODE makes use of this facility for a number of purposes including running external programs, displaying text files, and deleting communications

files created to run external programs. The most important use of this feature is to run the program which fits a model to the column data, and to run the graphics program.

The graphics program has been mentioned before as it relates to getting results from DICODE. More importantly, the program DECODE calls into a shareable image created from the Tektronix PLOT10 object code. A shareable image is like an executable subroutine, or a collection of executable subroutines without a main program. The DECODE object code must be linked with this shareable image in order to produce the executable. The shareable image has only one entry point specific to the calling procedure contained in DECODE and is therefore useless for any other purpose.

When the user specifies that he would like to see a graphical representation of the equipment and control designs, DICODE writes a communication file containing the names of the members of several classes of objects. Each of the names of these objects corresponds to a data file called a *meta file* which contains information to draw a small piece of the graphics result. DICODE then runs the external program DECODE which parses the communications file to obtain the names of the meta files as the elements of a string array. DECODE then places the terminal in graphics mode, and passes the string array to the shareable image which contains the executable to reproduce the graphics results. Within the shareable image, each of the meta files is opened separately, and the meta file is read and interpreted as GKS commands to draw

on the screen. The GKS commands are executed using data also provided in the meta file. The shareable image then awaits input from the user to indicate that he is finished looking at the picture. When a carriage return is entered, control then returns to DECODE which sends the necessary escape or control characters to the terminal to switch it back to text mode, and ends execution. Control then returns to DICODE, and the user is returned to the menu from which he requested to see a graphics result.

Each meta file was created by a FORTRAN executable which calls into another shareable image not provided with the DICODE software. The FORTRAN code makes many calls to graphics subroutines contained in the shareable image in order to draw a small graphics figure on a workstation. One possible workstation is a metafile which must be opened in a FORTRAN statement. As the picture is drawn, all information needed to reproduce the picture is entered into the meta file. The meta files are then used as described above to generate the graphical results of DICODE.

Another use of external programs is the steady state model of the distillation column which provides the RGA, RDG, and an estimate of the integrated error due to a disturbance (IE). During execution, DICODE may determine that the RGA, RDG, or IE is necessary to solve the design problem. Since the external program named DICODETOOL is defined in the DICODE knowledge base as the only source for the RGA, RDG and IE, the external program is executed. However, also in the definition of the program DICODETOOL,

it is specified that a communications file must first be written containing information which DICODETOOL needs. Therefore, the needed information is obtained from the user and written to a file before the program executes. The information which DICODETOOL needs includes the reflux ratio, mole fractions of light and heavy keys in the feed, distillate, and bottom flows, the type of disturbance expected to enter the system, the number of ideal trays in the column, and the feed quality.

The model is a simple one which requires only a one dimensional search for the relative volatility which will obtain the product purities based on the number of trays and reflux ratio. The relative volatility is determined to within $10^{-6}\%$ relative error. Once the relative volatility has been found, it is used to calculate the RGA, RDG, and IE from equations which were obtained through analytical differentiation of the model equations. DICODETOOL then writes the values in another communications file and exits. When control returns to DICODE, the values of RGA, RDG, and IE are read from the communications file and inferencing continues.

C.7. DICODE in the Computer

Once DICODE has been installed properly, all software except for KESR.EXE should be contained in subdirectories under the directory named DICODE.DIR. The DICODE.DIR directory can reside anywhere on disk pro-

vided that the users have read privileges. A number of directories exist under DICODE.DIR. COMMAND.DIR contains DCL command files used to compile, link, and run the source code to create the meta (data) files for graphics. More importantly, it contains the command file DICODELOGIN.COM which defines pathnames to the files DICODE.PKB, SHOGRAPH.EXE, and the graphics shareable image GRAPHIX.OLB. It also contains the command file which moves the files to the correct subdirectories under DICODE.DIR during installation.

The subdirectory DATA.DIR contains all of the meta files used by the graphics programs SHOGRAPH, and DECODE as well as the data files containing the tuning parameters for DICODE to use with the RGA, RDG, reflux ratio and vapor boilup to bottom flow ratio. The tuning data files are explained in detail in Section C.8. The subdirectory DOC.DIR contains documentation about DICODE including a list of all of the graphics meta files with a description of the picture drawn by each, and the installation procedures for DICODE. The directory EXEC.DIR contains all executable files used by DICODE. The KESR.EXE file can be installed in this directory if necessary. The graphics shareable image GRAPHIX.EXE is located in EXEC.DIR as well as the object library or entry point definition file GRAPHIX.OLB. The KBS.DIR directory contains the parsed and unparsed knowledge bases DICODE.PKB, and DICODE.KB respectively. The OBJECT.DIR may contain any of the object code resulting from compiling the source code in the SOURCE.DIR directory. The

SOURCE.DIR directory contains all of the source code used to generate the meta files for graphics as well as the source code for SHOGRAPH, DECODE, and DICODETOOL.

Logical pathnames are defined in the user's environment so that he may run DICODE from any directory in which he has read and write privileges without needing to know where the DICODE software is located. The logical pathnames also allow DICODE to find the files it needs in order to run external programs to produce graphics and to use the steady state model. The pathnames are defined by two lines which can be added to either the system login file, or each user's login file. The first line defines a logical to point to the DICODE.DIR directory, and the second executes a command procedure in the subdirectory COMMAND.DIR under the DICODE.DIR directory. These lines and their placement in the login files are described in the installation procedures.

C.8. Tuning DICODE

DICODE uses an external program named DICODETOOL to calculate values for the RGA, RDG, and IE for several possible composition control loop pairings. These values are used to evaluate the effectiveness of using the associated pairing for control of the column. Each of these measures has an ideal value, but a range of values is considered acceptable. The reflux ratio and vapor boilup to bottom flow ratio may also be used in a similar way since it is desir-

able to control levels with the larger flows. DICODE compares the values of RGA and RDG of each pairing to a defined acceptable range. If the values are outside the acceptable range, then the pairing is removed from those considered for control purposes. The reflux ratio is similarly used as a measure to remove unfavorable composition control pairings based on the desire to use the larger flows for level control. If all possible pairings fail these tests, then DICODE increases the range of values considered acceptable by a defined amount. The process of increasing the acceptable ranges continues until one of two situations occur. The first situation is when DICODE fails to find an acceptable pairing after increasing the ranges to their limits. The second is when DICODE finds one or more acceptable pairings.

Limits on the upper and lower bounds of the ranges are defined. If the ranges of all the measures increase until the upper and lower bounds hit the constraints and no pairing survives the tests, then DICODE informs the user that no acceptable pairing could be obtained based on the constraints imposed on the ranges of acceptable values for the RGA, RDG, reflux ratio and vapor ratio. The user must then start the design over again. DICODE should never fail to find an acceptable pairing based on the tuning parameters provided as defaults. However, if the user provides DICODE with other tuning parameters, it is possible that no acceptable pairing will be found.

Under usual conditions, DICODE will find at least one pairing which will survive before hitting the constraints. Often, more than one pairing will survive

the tests in a single cycle of the increase in ranges. DICODE then uses the IE to select the most promising of those that survive.

The RGA, RDG, reflux ratio (RR), and vapor boilup to bottom flow ratio (VR), are defined as members of the class *Tools*. Each *Tool* has attributes *large, large increment, large hilimit, large lolimit, small, small increment, small lolimit,* and *small hilimit.* Each pairing is defined as a member of the class *Pairings* with attributes *RGA, RDG,* and *IE.* The values of *RGA* and *RDG* of the *Pairings* are compared with the values of *small* and *large* of the *Tools RGA* and *RDG* respectively to determine if they are within the acceptable ranges. If not, the pairing is considered not acceptable.

For each *Tool*, the values of *large* and *small* begin at the ideal value. Each time all pairings fail to pass the tests, the value of *large* is modified by multiplying it by the value of *large increment*, and the value of *small* is modified by multiplying it by the value of *small increment.* The changing of the values of *small* and *large* continues only until the value hits its respective constraint. Once the value of *large* reaches *large hilimit* or *large lolimit* it is no longer modified, and once the value of *small* reaches *small lolimit* or *small hilimit* it is no longer modified.

The following is a listing of the file RGARDG.DAT which is used to consider primarily the RGA and RDG to determine the best control loop pairings.

```
assertclass Tools = RGA, RDG, RR, VR.
Tools:RGA>Large = 1.0.
Tools:RGA>Small = 1.0.
Tools:RGA>Small increment = 0.98.
Tools:RGA>Large increment = 1.2.
```

```
Tools:RGA>Large hilimit = 1000.0.
Tools:RGA>Large lolimit = 1.0.
Tools:RGA>Small hilimit = 1.0.
Tools:RGA>Small lolimit = 0.5.
Tools:RDG>Large = 0.1.
Tools:RDG>Small = 0.1.
Tools:RDG>Small increment = 1.05.
Tools:RDG>Large increment = 1.2.
Tools:RDG>Large hilimit = 1000.0.
Tools:RDG>Large lolimit = 0.0.
Tools:RDG>Small hilimit = 1.0.
Tools:RDG>Small lolimit = 0.0.
Tools:RR>Large = 1.0.
Tools:RR>Small = 1.0.
Tools:RR>Small increment = 0.9.
Tools:RR>Large increment = 1.1.
Tools:RR>Large hilimit = 100.0.
Tools:RR>Large lolimit = 1.0.
Tools:RR>Small hilimit = 1.0.
Tools:RR>Small lolimit = 0.01.
Tools:VR>Large = 1.0.
Tools:VR>Small = 1.0.
Tools:VR>Small increment = 0.9.
Tools:VR>Large increment = 1.1.
Tools:VR>Large hilimit = 100.0.
Tools:VR>Large lolimit = 1.0.
Tools:VR>Small hilimit = 1.0.
Tools:VR>Small lolimit = 0.01.
Consider Ratios = yes <1.00>.
%
```

Notice here that the attributes associated with RGA create the following range characteristics. The ideal value of the RGA is 1.0, so the values of *small* and *large* begin at 1.0. Each time all pairings fail to survive, the value of *small* is decreased by multiplying it by 0.98, the value of *small increment.* The minimum value of the RGA for a pairing is 0.5 so the value of *small lolimit* is 0.5. The value of *small hilimit* is given as 1.0 but it is actually irrelevant. The value of *large* for RGA is increased by multiplying it by 1.2, the value of *large increment.* The maximum value of RGA for a pairing is set at 1000.0 which simply represents a very large value. The value of *large lolimit* is set at 1.0, but is not actually needed for DICODE.

The attributes associated with the RDG are similarly defined. The ideal

value is actually 0.0, but a small value of 0.1 is given for the initial value of *small* and *large*. The value of large is increased by multiplying it by 1.2, the value of *large increment*, until it reaches the constraint of 1000.0 given by *large hilimit*. The value of *small* is actually increased in size rather than decreased by multiplying it by 1.05, the value of *small increment*, until it reaches a maximum of 1.0 set by *small hilimit*. The value of *large lolimit* is given as 0.1, and the value of *small lolimit* is given as 0.0 even though these values are not needed. The reason that the value of *small* is increased rather than decreased is because of the way it is used in a rule which combines the RGA and RDG. That rule is shown here.

```
Large RGA large RDG not good:
p:Pairing
if
     p>RGA ge Tools:RGA>large and
     p>RDG ge Tools:RDG>small
then
     p>Possible = false
endif.
```

In this case, we would like to allow a pairing whose RGA is large, but whose RDG is small to survive the test. Therefore, we use small here in a different sense for the RDG. It is also a measure of increasing size not being good, but the size increase is at a different rate than the size increase for *large*. Therefore, a pairing with a somewhat large RGA but a small RDG is not removed from those considered possible.

For the *Tools* RR and VR, the procedure is simple. The desired effect is

simply to ensure that the stream used for level control is at least 1% of the total flow from the vessel. However, it is desired to be somewhat discriminating in the case that two pairings have similar RGA and RDG values, but one is preferable to the other because it uses a larger flow for level control. Therefore, the values of *large* and *small* begin at 1.0, and are incremented and decremented respectively by 10% on each cycle. The maximum value is 100, and the minimum is 0.01.

The following rule demonstrates how the reflux ratio is used to eliminate unfavorable composition control loop pairings.

```
Large reflux ratio should not choose L:
if
     Reflux Ratio ge Tools:RR>large
then
     Pairing:LV>Possible = false.
     Pairing:LB>Possible = false.
     Pairing:LB_L>Possible = false.
     Pairing:LV_B>Possible = false.
endif.
```

This rule indicates that if the reflux ratio is larger than the value *large*, then all pairings which use reflux flow for distillate composition control are removed from those considered acceptable. The removal of composition control pairings using reflux flow is equivalent to removing those that use distillate flow for reflux drum level. The pairing LV indicates that reflux flow L is used for distillate composition control, and vapor boilup V is used for bottom composition control. The pairing LB_L indicates that reflux flow is used for distillate composition

control and B/L (bottom flow over reflux flow) is used for bottom composition control. Each pairing removed by this rule uses reflux flow for distillate composition control (and therefore distillate flow for reflux drum level). It is removed because the reflux drum level is controlled by a flow smaller than is acceptable.

The following is a listing of the file STREAMS.DAT which uses only stream sizes to determine the best pairing for composition control (and thus level control).

```
assertclass Tools = RGA, RDG, RR, VR.
Tools:RGA>Large = 1000.0.
Tools:RGA>Small = -1000.0.
Tools:RGA>Small increment = 1.00.
Tools:RGA>Large increment = 1.2.
Tools:RGA>Large hilimit = 100000.0.
Tools:RGA>Large lolimit = 1.0.
Tools:RGA>Small hilimit = 1.0.
Tools:RGA>Small lolimit = -100000.0.
Tools:RDG>Large = 1000.0.
Tools:RDG>Small = 1000.0.
Tools:RDG>Small increment = 1.02.
Tools:RDG>Large increment = 1.2.
Tools:RDG>Large hilimit = 100000.0.
Tools:RDG>Large lolimit = 0.1.
Tools:RDG>Small hilimit = 100000.0.
Tools:RDG>Small lolimit = 0.0.
Tools:RR>Large = 10.0.
Tools:RR>Small = 5.0.
Tools:RR>Small increment = 0.8.
Tools:RR>Large increment = 1.2.
Tools:RR>Large hilimit = 100000.0.
Tools:RR>Large lolimit = 5.0.
Tools:RR>Small hilimit = 10.0.
Tools:RR>Small lolimit = 0.00001.
Tools:VR>Large = 20.0.
Tools:VR>Small = 10.0.
Tools:VR>Small increment = 0.8.
Tools:VR>Large increment = 1.2.
Tools:VR>Large hilimit = 100000.0.
Tools:VR>Large lolimit = 10.0.
Tools:VR>Small hilimit = 5.0.
Tools:VR>Small lolimit = 0.00001.
Consider Ratios = no <1.00>.
%
```

Notice here that for RGA and RDG, the values for *small lolimit* and *large lolimit* are given very small values, and *small hilimit* and *large hilimit* are given

very large values. Also, the values of *large* are given very large values, and the values of *small* are given as very small. Therefore, the ranges considered acceptable are very large, resulting in the RGA and RDG not playing a role in determining the control loop pairing. However, notice the values used for RR and VR. The break points (*small* and *large*) are defined as 5.0 and 10.0 for the reflux ratio and 10.0 and 20.0 for vapor boilup to bottom flow ratio. The small values are decremented by 20% and the large values are incremented by 20% at each cycle. The limiting values on *large* and *small* are given as very small and very large respectively. For RR and VR, a composition control pairing's acceptability depends upon which manipulated variable will end up being used for level control.

For the STREAMS.DAT tuning, the control pairing selection is implemented in the following way. A pairing is considered unacceptable if it uses the reflux flow for reflux drum level control and the reflux ratio is less than the value of *Tools:RR>small.* A pairing is also considered unacceptable if it uses distillate flow for reflux drum level control and the reflux ratio is greater than *Tools:RR>large.* Similarly, a pairing is considered unacceptable if it uses vapor boilup to control column base level and the vapor ratio is less than the value of *Tools:VR>small*, or if it uses tails flow for column base level control and the vapor ratio is greater than the value of *Tools:VR>large.* If pairings survive these tests, then they are further tested based on a comparison of the reflux ratio and the vapor ratio divided by two. The comparison of RR and

scaled VR provides the selection between the pairings which use one internal and one external flow for material balance when the vapor ratio and reflux ratio are large.

The file named YOUROWN.DAT contains a copy of the file RGARDG.DAT. If the user would like to create his own tuning parameters, the file YOUROWN.DAT must be modified to use the RGA, RDG, RR, and VR as he would like. If he does not want one of the measures RGA or RDG to be considered, then values of *large* and *small* which create a window inside of which the measure will always fall should be provided. He may also change the breakpoints for RR and VR to match his own experience for control design. If he would like to ignore the relative stream sizes in considering control loop pairings, then a very large range should be provided initially. A very large value for *large*, and a very small value for *small* should be used. The increments may be given as 1.0 so that the values never change.

APPENDIX D

CUSTOMIZED SHELL

The following is a listing of a partial knowledge base written for the KES^{TM} expert system shell. The purpose of the knowledge base is to provide the facilities for building an expert system to design a single input single output control system for a process. A complete discussion of the customized shell appears in Chapter 6 of the thesis. The knowledge base is documented throughout to indicate where changes and additions should be made in order to extend it for an expert system to design controls for a specific process.

The partial knowledge base does not include any warnings or recommendations to be presented to the user. These must be provided as part of the extension for a specific purpose based on the experience of the expert whose knowledge is being represented. The knowledge base presented here is actually complete as a small example. The only user provided input is the values of the measures of adequacy called *tool1*, and *tool2* of the single possible pairing *M1M2*. The specific rules which create the controllers based on the experts knowledge must be provided, replacing those present for the example.

Documentation within the knowledge base also indicates that part of the knowledge base may be omitted if measures of control loop pairings are not used as a criterion for determining the control system design. If other information is needed in order to make this decision, input attributes must be provided in the *attributes* section of the knowledge base.

```
\**  The following is a sample knowledge base for use to cre-  **
\**  ate an expert system for control design.  The method uses **
\**  a goal tree to represent the problem, with controllers    **
\**  satisfying the lowest level subgoals.  The individual     **
\**  using this partial knowledge base should provide the goal **
\**  tree structure and the specific rules to create control-  **
\**  lers.  The knowledge base given here works but is simply  **
\**  a sample to be used as a template for knowledge base con- **
\**  struction.  The knowledge base is documented throughout   **
\**  to indicate where changes are to be made in order to cre- **
\**  ate your own expert system.                               **
\**  Note that explanation facilities are not provided, and no **
\**  demons are present to provide warnings or recommendations.**
\**  Furthermore, no user supplied attributes are defined other**
\**  than the values of the various measures for particular    **
\**  control configurations.                                   **
text:
\**  The following text is displayed when the system cannot    **
\**  find a pairing which satisfies the criterion specified as **
\**  constraints on the measures.                              **
  {failure:
  " ",
  "The expert system has failed to find an acceptable solution",
  "due to constraints which do not allow any possible solutions",
  "to succeed.",
  " "}
%
\**  The following section defines types. Types are attributes **
\**  types just like real, int, str, or sgl.  They are then    **
\**  used in the definition of attributes allowing attributes  **
\**  normally of type sgl to be compared.                      **
types:
  link: sgl
    (dummy, link1, link2, link3, link4, link5, link6, link7,
     link8, link9, link10).
  controlled variables: sgl
    (SG1, SG2, SG3, SG4, SG5, SG6, SG7, SG8, SG9, SG10,
     dummy).
  manipulated variables: sgl
```

```
    (M1, M2, M3, M4, M5, M6, M7, M8, M9, M10).
%
attributes:
  pairing process: sgl
    (complete, not complete)
    [default: not complete].
  temp: real.
  epsilon: real [default: 0.0001].
  itemp: int.
  itemp1: int.
  itemp2: int.
%
\**  The following section defines the goals, controllers,     **
\**  tools and pairings, and manipulated variables are defined **
\**  as classes or frames.  Instances of these are then created**
\**  to represent the problem.  See the actions section where  **
\**  the goal tree and manipulated variables are set up for    **
\**  the example.                                              **
classes:
  goals:
    attributes:
      uplink: link [default: dummy].
      downlink: link [default: dummy].
      satisfied: truth.
      variable: controlled variables [default: dummy].
    %
  endclass.
  subgoals: [default: dummy]
    attributes:
      uplink: link [default: dummy].
      downlink: link [default: dummy].
      satisfied: truth.
      variable: controlled variables [default: dummy].
      lowest: truth [default: false].
    %
  endclass.
  sub2goals: [default: dummy]
    attributes:
      uplink: link [default: dummy].
      downlink: link [default: dummy].
      satisfied: truth.
      variable: controlled variables [default: dummy].
      lowest: truth [default: false].
    %
  endclass.
  sub3goals: [default: dummy]
    attributes:
      uplink: link [default: dummy].
      downlink: link [default: dummy].
      satisfied: truth.
      variable: controlled variables [default: dummy].
      lowest: truth [default: false].
    %
```

```
endclass.

sub4goals: [default: dummy]
  attributes:
    uplink: link [default: dummy].
    downlink: link [default: dummy].
    satisfied: truth.
    variable: controlled variables [default: dummy].
    lowest: truth [default: false].
  %
endclass.

sub5goals: [default: dummy]
  attributes:
    uplink: link [default: dummy].
    downlink: link [default: dummy].
    satisfied: truth.
    variable: controlled variables [default: dummy].
    lowest: truth [default: false].
  %
endclass.

controllers: [default: dummy]
  attributes:
    controlled: controlled variables [default: dummy].
    manipulated: manipulated variables.
  %
endclass.

regulatory controllers: [inherits: controllers]
  [default: dummy]
endclass.

supportive controllers: [inherits: controllers]
  [default: dummy]
endclass.

Constraint controllers: [inherits: controllers]
  [default: dummy]
endclass.

other controllers: [inherits: controllers]
  [default: dummy]
endclass.

Manipulated Variables: [inherits: controllers]
  attributes:
    name: manipulated variables.
    availability: truth [default: true].
  %
endclass.

Tools: [default: tool1, tool2]
  attributes:
    large_value: real.
    small_value: real.
    large_constraint: real.
    small_constraint: real.
    relaxation: real.
  %
endclass.

pairing: [default: M1M2]
  attributes:
```

```
      Possible: truth [default: true].
      tool1: real.
      tool2: real.

\**  KB author add attributes for other measures of configura- **
\**  tion adequacy here.                                        **

    %
  endclass.

%

\**  The rules section contains the rules used for backward     **
\**  chaining.  The first 5 rules are used to link the layers   **
\**  of the goal tree together.                                 **

rules:

  Goal satisfaction rule:
    g1:goals,
    g2:subgoals
    if
      g2>uplink = g1>downlink and
      g2>satisfied = false
    then
      g1>satisfied = false.
    endif.

  Subgoal satisfaction rule:
    g1:subgoals,
    g2:sub2goals
    if
      g2>uplink = g1>downlink and
      g2>satisfied = false
    then
      g1>satisfied = false.
    endif.

  Sub2goal satisfaction rule:
    g1:sub2goals,
    g2:sub3goals
    if
      g2>uplink = g1>downlink and
      g2>satisfied = false
    then
      g1>satisfied = false.
    endif.

  Sub3goal satisfaction rule:
    g1:sub3goals,
    g2:sub4goals
    if
      g2>uplink = g1>downlink and
      g2>satisfied = false
    then
      g1>satisfied = false.
    endif.

  Sub4goal satisfaction rule:
    g1:sub4goals,
    g2:sub5goals
    if
      g2>uplink = g1>downlink and
      g2>satisfied = false
```

```
    then
      g1>satisfied = false.
    endif.

\**  The next 5 rules establish satisfaction of a subgoal that **
\**  is lowest in level.  That is, the defined goal has no     **
\**  goals below it, but it is satisfied by a controller.      **

  Lowest subgoal satisfaction by controller:
    g:subgoals,
    c:controllers
    if
      c>controlled = g>variable
    then
      g>satisfied = true.
    endif.

  Lowest sub2goal satisfaction by controller:
    g:sub2goals,
    c:controllers
    if
      c>controlled = g>variable
    then
      g>satisfied = true.
    endif.

  Lowest sub3goal satisfaction by controller:
    g:sub3goals,
    c:controllers
    if
      c>controlled = g>variable
    then
      g>satisfied = true.
    endif.

  Lowest sub4goal satisfaction by controller:
    g:sub4goals,
    c:controllers
    if
      c>controlled = g>variable
    then
      g>satisfied = true.
    endif.

  Lowest sub5goal satisfaction by controller:
    g:sub5goals,
    c:controllers
    if
      c>controlled = g>variable
    then
      g>satisfied = true.
    endif.

\**  The following rule connects the pairing selection with    **
\**  the rules which assign controllers.  If the pairing se-   **
\**  lection method is not used, then this rule may be deleted.**

  Pairing completed:
    p:pairing
    if
      p>Possible = true
    then
      pairing process = complete.
    endif.
```

```
\**  The following rules are examples of the form of the rules **
\**  provided by the knowledge base author indicating the con- **
\**  ditions for success of a particular pairing of manipulat- **
\**  ed and controlled variables for regulatory control.  The  **
\**  conditions consider the tools available. These four rules **
\**  should be replaced if necessary by the kb author.  If the **
\**  pairing selection method is not used, then these rules    **
\**  should be deleted.                                        **

  Tool1 creates failure for pairing based on large_value:
    p:pairing
    if
      p>tool1 gt Tools:tool1>large_value
    then
      p>Possible = false.
    endif.

  Tool1 creates failure for pairing based on small_value:
    p:pairing
    if
      p>tool1 lt Tools:tool1>small_value
    then
      p>Possible = false.
    endif.

  Tool2 creates failure for pairing based on large_value:
    p:pairing
    if
      p>tool2 gt Tools:tool2>large_value
    then
      p>Possible = false.
    endif.

  Tool2 creates failure for pairing based on small_value:
    p:pairing
    if
      p>tool2 lt Tools:tool2>small_value
    then
      p>Possible = false.
    endif.

\**  The following rules are examples of the form of the rules **
\**  provided by the knowledge base author indicating the con- **
\**  ditions for creating regulatory controllers. The creation **
\**  of regulatory controllers is based on the success of a    **
\**  pairing which passed the tests.  These rules should be    **
\**  replaced by the kb author.                                **
\**  If the pairing selection method is not used, then the     **
\**  first two lines of these rules should be replaced with    **
\**  appropriate conditions as used by the expert system to    **
\**  create controllers. The last condition should be retained **
\**  in the case of regulatory control. Also, the names of the **
\**  controllers may be changed to be descriptive as should    **
\**  the names of the controlled variables, manipulated        **
\**  variables, and goals.                                     **

  Goal12 satisfied:
```

```
    if
      pairing process = complete and
      pairing:M1M2>Possible = true and
      Manipulated Variables:M1>availability = true
    then
      assertclass regulatory controllers = C1.
      regulatory controllers:C1>controlled = SG1.
      regulatory controllers:C1>manipulated = M1.
    endif.

  Goal21 satisfied:
    if
      pairing process = complete and
      pairing:M1M2>Possible = true and
      Manipulated Variables:M2>availability = true
    then
      assertclass regulatory controllers = C2.
      regulatory controllers:C2>controlled = SG2.
      regulatory controllers:C2>manipulated = M2.
    endif.

  Goal22 satisfied by M1:
    if
      pairing process = complete and
      Manipulated Variables:M1>availability = true
    then
      assertclass regulatory controllers = C3.
      regulatory controllers:C3>controlled = SG3.
      regulatory controllers:C3>manipulated = M1.
    endif.

  Goal22 satisfied by M2:
    if
      pairing process = complete and
      Manipulated Variables:M2>availability = true
    then
      assertclass regulatory controllers = C3.
      regulatory controllers:C3>controlled = SG3.
      regulatory controllers:C3>manipulated = M2.
    endif.

  Goal22 satisfied by M3:
    if
      pairing process = complete and
      Manipulated Variables:M3>availability = true
    then
      assertclass regulatory controllers = C3.
      regulatory controllers:C3>controlled = SG3.
      regulatory controllers:C3>manipulated = M3.
    endif.

%

\**  The demons section below contains all demons necessary to **
\**  create the knowledge base for control design provided     **
\**  that the goal tree is no deeper than a total of 6 levels  **
\**  including the root goal and all lowest level subgoals.    **

demons:

  Make goal satisfied_false for_unknown:
    g:goals
```

```
  when
    status(g>satisfied) = unknown
  then
    if
      g>variable # dummy
    then
     erase g>satisfied.
      g>satisfied = false.
    endif.
    if
      g>variable = dummy
    then
      erase g>satisfied.
      g>satisfied = true.
    endif.
  endwhen.

Make subgoal satisfied_false for_unknown:
  g:subgoals
  when
    status(g>satisfied) = unknown
  then
    if
      g>variable # dummy
    then
      erase g>satisfied.
      g>satisfied = false.
    endif.
    if
      g>variable = dummy
    then
      erase g>satisfied.
      g>satisfied = true.
    endif.
  endwhen.

Determine sub2goal satisfied for_unknown:
  g:sub2goals
  when
    status(g>satisfied) = unknown
  then
    if
      g>variable # dummy
    then
      erase g>satisfied.
      g>satisfied = false.
    endif.
    if
      g>variable = dummy
    then
      erase g>satisfied.
      g>satisfied = true.
    endif.
  endwhen.

Determine sub3goal satisfied for_unknown:
  g:sub3goals
  when
    status(g>satisfied) = unknown
  then
    if
      g>variable # dummy
    then
      erase g>satisfied.
```

```
      g>satisfied = false.
    endif.
    if
      g>variable = dummy
    then
      erase g>satisfied.
      g>satisfied = true.
    endif.
  endwhen.

Determine sub4goal satisfied for_unknown:
  g:sub4goals
  when
    status(g>satisfied) = unknown
  then
    if
      g>variable # dummy
    then
      erase g>satisfied.
      g>satisfied = false.
    endif.
    if
      g>variable = dummy
    then
      erase g>satisfied.
      g>satisfied = true.
    endif.
  endwhen.

\**  The following demon should be repeated as many times as    **
\**  is necessary to incorporate enough levels in the goal      **
\**  tree to accommodate the problem solution.  Simply copy the**
\**  demon and change sub5goal to subNgoal in the first and     **
\**  second lines where N is replaced by the next integer in    **
\**  sequence as many times as is necessary.                    **

Determine sub5goal satisfied for_unknown:
  g:sub5goals
  when
    status(g>satisfied) = unknown
  then
    if
      g>variable # dummy
    then
      erase g>satisfied.
      g>satisfied = false.
    endif.
    if
      g>variable = dummy
    then
      erase g>satisfied.
      g>satisfied = true.
    endif.
  endwhen.

Remove manipulated variable from design:
  c:regulatory controllers,
  m:Manipulated Variables
  when
    c>manipulated = m>name
  then
    erase m>availability.
    m>availability = false.
```

```
    endwhen.
\**  If tools are not used to evaluate the control pairings    **
\**  then the following demon may be deleted.                  **
  Relax thresholds for pairing failure:
    when
      pairing process = not complete
    then
      erase itemp1.
      erase itemp2.
      itemp1 = 0.
      itemp2 = 0.
      forall t:Tools do
        erase itemp.
        itemp = itemp1.
        erase itemp1.
        itemp1 = itemp + 1.
        erase temp.
        temp = t>large_value.
        erase t>large_value.
        t>large_value = temp*(1.0 + t>relaxation).
        erase temp.
        temp = t>small_value.
        erase t>small_value.
        t>small_value = temp*(1.0 - t>relaxation).
        if
          t>large_value gt t>large_constraint
        then
          erase t>large_value.
          t>large_value = t>large_constraint.
          erase itemp.
          itemp = itemp2.
          erase itemp2.
          itemp2 = itemp + 1.
        endif.
        if
          t>small_value lt t>small_constraint
        then
          erase t>small_value.
          t>small_value = t>small_constraint.
          erase itemp.
          itemp = itemp2.
          erase itemp2.
          itemp2 = itemp + 1.
        endif.
      endforall.
      erase itemp.
      itemp = itemp1.
      erase itemp1.
      itemp1 = 2*itemp.
      if
        itemp1 = itemp2
      then
        display attach failure of kb.
        break.
        stop.
      endif.
      forall p:pairing do
        erase p>Possible.
      endforall.
```

```
      erase pairing process.
      obtain pairing process.
    endwhen.
%
actions:
\**  The following section of actions sets up the goal tree    **
\**  structure by creating members of the classes goals, sub-  **
\**  goals, sub2goals, etc., and giving the attributes uplink  **
\**  and downlink appropriate values.  This section should be  **
\**  replaced by the knowledge base author to reflect the form **
\**  of the goal tree for his application.  Alternatively, the **
\**  information here could be read from a data file.           **
  assertclass goals = Goal1.
  assertclass subgoals = Goal11, Goal12.
  assertclass sub2goals = Goal21, Goal22.
  goals:Goal1>downlink = link1.
  subgoals:Goal11>uplink = link1.
  subgoals:Goal11>downlink = link2.
  subgoals:Goal12>uplink = link1.
  subgoals:Goal12>variable = SG1.
  sub2goals:Goal21>uplink = link2.
  sub2goals:Goal21>variable = SG2.
  sub2goals:Goal22>uplink = link2.
  sub2goals:Goal22>variable = SG3.
\**  The following section of actions creates the criterion by **
\**  which pairings are determined to be acceptable or not.    **
\**  In the classes section above, the class Tools was created **
\**  and given default members of tool1 and tool2. The members **
\**  of the class tools should be changed by the knowledge     **
\**  base author to reflect the names of the measures used to  **
\**  decide upon the best regulatory control system.  Also,    **
\**  appropriate values for large_constraint, small_constraint,**
\**  and relaxation should be contained here, as well as       **
\**  initial values for large_value, and small_value.          **
\**  Alternatively, this in formation may be read from a data  **
\**  file.                                                     **
  Tools:tool1>large_value = 1.0.
  Tools:tool1>small_value = 1.0.
  Tools:tool1>large_constraint = 10.0.
  Tools:tool1>small_constraint = 0.0.
  Tools:tool1>relaxation = 0.1.
  Tools:tool2>large_value = 2.0.
  Tools:tool2>small_value = 2.0.
  Tools:tool2>large_constraint = 10.0.
  Tools:tool2>small_constraint = 1.0.
  Tools:tool2>relaxation = 0.1.
\**  The following lines of actions create the members of the  **
\**  manipulated variables used for control design.  These     **
\**  could be changed to descriptive values, however, then the **
\**  values given for the variable type "manipulated variables"**
\**  defined in the types section must be changed to reflect   **
```

```
\**  the new values to be used.                                        **
  assertclass Manipulated Variables = M1, M2, M3.
  Manipulated Variables:M1>name = M1.
  Manipulated Variables:M2>name = M2.
  Manipulated Variables:M3>name = M3.
\**  The next section of actions checks the constraints on the **
\**  tool values to make sure they are not zero.  A zero con-  **
\**  straint would never be reached due to the way the large   **
\**  and small are relaxed (in proportion to their size).      **
  forall t:Tools do
    if
      t>small_constraint lt epsilon
    then
      erase t>small_constraint.
      t>small_constraint = epsilon.
    endif.
    if
      t>large_constraint lt -1*epsilon
    then
      erase t>large_constraint.
      t>large_constraint = -1*epsilon.
    endif.
  endforall.
\**  The rest of the actions section consists of initiating    **
\**  the chaining process by specifying to obtain the value of **
\**  the attribute satisfied of the root goal (Goal1), and     **
\**  then dis playing the results.  Other actions should       **
\**  replace this display method in order to conform to the    **
\**  author's desires.                                         **
  obtain goals:Goal1>satisfied.
  message combine("Root goal satisfaction is ",
                  goals:Goal1>satisfied).
  break.
  forall c:regulatory controllers do
    message combine("Controller ",c," controls ",c>controlled,
                    " with ",c>manipulated).
  endforall.
  break.
  stop.
%
```

BIBLIOGRAPHY

Årzén, K.E., "Realization of Expert System Based Feedback Control." Doctoral Dissertation, Department of Automatic Control, Lund Institute of Technology [1987]

Åström, K.J., J.J. Anton, and K.E. Årzén, "Expert Control." Automatica, 22, pp 227-286 [1986]

Asbjornsen, O. A., and B. Budurka, Systems Engineering Principles, Lecture Notes, The University of Maryland, College Park [1987]

Birky, G. J., T. J. McAvoy, and M. Modarres. "An Expert System for Distillation Control Design." Computers and Chemical Engineering, (to appear) [1989]

Birky, G.J., T.J. McAvoy, and B.D. Tyreus, "Expert System for Design of Distillation Controls." Proceedings of the 1988 Instrument Society of America Conference, October 16-21, [1988]

Blanchard, B. S., and W. J. Fabrycky, Systems Engineering and Analysis. Prentice Hall, Englewood Cliffs, NJ [1981]

Bristol, E., "After DDC: Idiomatic (Structural) Control." presented at 88^{th} National A.I.Ch.E. Meeting, Philadelphia, PA, June [1980]

Buckley, P.S., "Override Controls for Distillation Columns." Instrumentation Technology [1968], Reprinted by Moore Products Co. - Tech Paper 6804

Buckley, P.S., W.L. Luyben, and J.P. Shunta, Design of Distillation Column Control Systems. Instrument Society of America, Research Triangle Park, NC [1985]

Charniak, E. and D. McDermott, Introduction to Artificial Intelligence., Addison-Wesley Publishing Company, Inc., Reading, MA [1985]

Chin, G., "Guide to distillation pressure control methods," Hydrocarbon Processing, October [1979]

Eduljee, H.E., "Equations Replace Gilliland Plot.", Hydrocarbon Processing 54, pp. 120-122, September [1975]

Eisner, H. Computer Aided Systems Engineering. Prentice Hall, Englewood Cliffs, NJ [1987]

Fenske, M.R., "Fractionation of Straight-Run Pennsylvania Gasoline." Ind. Eng. Chem. 24, p. 482 [1932]

Fink, P.K., "Control and Integration of Diverse Knowledge in a Diagnostic Expert System." Proceedings of 9^{th} IJCIA [1985]

Gorman, R.P. and T.J. Sejnowski, "Analysis of Hidden Units in a Layered Network Trained to Classify Sonar Targets." Neural Networks, Vol. 1, pp. 75-89 [1988]

Hebb, D.O., The organization of behavior. Wiley, New York [1949]

Hecht-Neilson, R., "Kolmogorov's Mapping Neural Network Existence Theorem." Proceedings of the IEEE International Conference on Neural Networks [1987].

Jackson, J.H., "On localization." In Selected Writings Vol. 2, Basic Books, New York [1958]

Kapoor, N., M. Modarres, and T.J. McAvoy, "MAKE: Maryland Automatic Knowledge Extractor." Proceedings of ACC, p. 1347, Minneapolis, MN, June [1987]

"KES^{TM} PS Knowledge Base Author's Manual," Software Architecture & Engineering, Inc., Update A [1988]

Kister, H., and I. Doig, "When Would Floating Pressure Strategy Save Energy?" CEP, Vol. 77, No. 9, pp. 55-57, September [1981]

Kolmogorov, A.N., "On the Representation of Continuous Functions of Many Variables by Superposition of Continuous Functions of One Variable and Addition." Dokl. Akad. Naud USSR, 114, 953-956 [1957]

Lashley, K.S., "In Search of the Engram." Society of Experimental Biology Symposium No. 4: Psychological mechanisms in animal behavior, pp. 478-505, Cambridge University Press, London [1950]

Lorentz, G.G., "The 13th Problem of Hilbert." Proceedings of Symposia in Pure Mathematics, 28, American Mathematical Society [1976]

Luria, A.R., Higher coritcal functions in man. Basic Books, New York [1966]

Marino-Galarraga, M., T.J. McAvoy, and T.E. Marlin, "Short-Cut Operability Analysis Part II - Estimation of f_i Detuning Parameter for Classical Control Systems," I&EC Research, 26, p. 511 [1987]

Marino-Galarraga, M., T.J. McAvoy, and T.E. Marlin, "Short-Cut Operability Analysis Part III - Short-Cut Methodology for the Assessment of Process Control Designs," I&EC Research, 26, p. 521 [1987]

McAvoy, T.J., Interaction Analysis. Instrument Society of America, Research Triangle Park, NC [1983]

Modarres, M. and T. Cadman, "A Method of Alarm System Analysis in Process Plants." Computers and Chemical Engineering, Vol. 10, No. 6 [1986]

Prassinos, A.G., "Analysis and Synthesis of Complex Control Systems Via Idiomatic Control." M.S. Thesis, University of Maryland, College Park, [1982]

Prassinos, A.G., T.J. McAvoy, and E. Bristol, "Analysis and Synthesis of Complex Control Systems." Proceedings of the 9^{th} IFAC, Vol. 3, Budapest, Hungary, July 2 - 6, [1984]

Radian Corp.,"Energy Conservation: A Route to Improved Distillation Profitability." Technology Transfer Manual (DOE/CS/4431-T1). Washington D.C.: Department of Energy, [1980].

Raphals, L.A., ed. and R.J. Chassell, "IKE^{TM} Users Guide." Preliminary, LMI, Cambridge, MA, 14 March [1986]

Rosenblatt, F., "Two theorems of statistical separability in the perceptron." In Mechanisation of thought processes: Proceedings of a symposium held at the National Physical Laboatory, November 1958. Vol. 1, pp. 421-456, HM Stationary Office, London [1959]

Rosenblatt, F., Principles of Neurodynamics. Spartan, New York [1962]

Rumelhart, D.E. and J.L. McClelland, Parallel Distrubuted Processing: Explorations in the Microstructure of Cognition, Vol. I, MIT Press, Cambridge, Massachusetts [1986]

Shinskey, F.G., "An Expert System for the Design of Distillation Controls." Proceedings of the Third International Conference on Chemical Process Control, Asilomar, CA, January 12-17 [1986]

Shinskey, F.G., Distillation Control For Productivity and Energy Conservation. 2^{nd} ed., McGraw-Hill Book Company, New York [1984]

Shinskey, F.G., "Disturbance-Rejection Capabilities of Distillation Control Systems," Proceedings of the 1985 ACC, p. 1072, Boston, MA [1985]

Shinskey, F.G., "Two Expert Systems for Batch Reactor Control." Proceedings of the BIRA Conference on Expert Systems, Antwerp, Belgium, October 14-15 [1986]

Stanley, G., M. Marino-Galarraga, and T.J. McAvoy, "Short-Cut Operability Analysis Part I - The Relative Disturbance Gain," I&EC Process Design & Development, 24 [1985]

Stephanopoulos, G., J. Johnston, T. Kriticos, R. Lakshaman, M. Mavrovouniotis, and C. Siletti, "Design Kit: An Object-Oriented Environment for Process Engineering." Computers & Chemical Engineering, Vol. 11, 6, p. 655 [1987]

Thurston, C.W., "Computer Aided Design of Distillation Column Controls Part1." Hydrocarbon Processing, 60, pp. 125-130, July [1981]

Thurston, C.W., "Computer Aided Design of Distillation Column Controls Part2." Hydrocarbon Processing, 60, pp. 135-140, August [1981]

Tuason, E.B., D. Bispo, "Nexpert Fundamentals," Bechtel AI Institute [1987]

Waterman, D.A., A Guide To Expert Systems. Addison-Wesley Publishing Company, Inc., Reading, MA [1986]