

THE ADVISE.1 META-EXPERT SYSTEM:
THE GENERAL DESIGN AND A TECHNICAL
DESCRIPTION

by

*R. S. Michalski
A. B. Baskin
C. T. Uhrik
T. D. Channic*

Reports of the Intelligent Systems Group, ISG 86-8, UIUCDCS-F-87-962, Department of
Computer Science, University of Illinois, Urbana-Champaign, January 1987.

Reports

of the

Intelligent
Systems
Group

File No. UIUCDCS-F-87-962

The ADVISE.1 Meta-Expert System:

The General Design and a Technical Description

by

R.S. Michalski, A.B. Baskin
C. Uhrik, T. Channic

January, 1987

ISG 86-8

CC
CC
CD
CD
CC

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN URBANA, ILLINOIS

File No. UIUCDCS-F-87-962

**THE ADVISE.1 META-EXPERT SYSTEM:
The General Design and a Technical Description**

by

R.S. Michalski, A.B. Baskin,
C. Uhrik, T. Channic
Department of Computer Science
University of Illinois

ISG Report 86-8

January, 1987

This research has been supported in part by the Office of Naval Research under grants No. N00014-83-G-0030 and No. N00014-82-K-0186, the Defense Advanced Research Project Agency under grant No. N00014-85-K-0878, and the National Science Foundation under grant NSF DCR 84-06801.

THE ADVISE.1 META-EXPERT SYSTEM: The General Design and a Technical Description

ABSTRACT

This report describes a general purpose inference system called ADVISE.1. The system provides a set of tools for building and testing advisory or expert systems; hence we call it a meta-expert system. Due to its novel architecture and many novel features, it represents a second generation expert system shell. Among the important features of the system are:

- representation of knowledge in three different forms: a rule base, a network base, and a relational data base,
- availability of several inference control strategies,
- inductive learning capabilities,
- several rule evaluation schemes, and
- separation of control and meta-knowledge from domain knowledge.

The system has been tested in four different domains: soybean disease diagnosis, crop damage estimates, infant monitoring in newborn intensive care units, and entomological pest identification. These implementations and their experimental results are discussed.

Table of Contents

Chapter	Page
Chapter 1 A General Overview	1
1.1. Goals and Major Features of ADVISE.1	2
1.2. Conceptual Organization of ADVISE.1	3
1.2.1. Control Block and User Interface	4
1.2.2. Knowledge Base	5
1.2.3. Query Blocks	7
1.2.4. Knowledge Acquisition Block	8
1.3. The Architecture of ADVISE.1	9
Chapter 2 Research Goals of ADVISE and a Brief Review of Other Systems	12
2.1. Knowledge Representation	13
2.2. Structure Within a Rule Base	13
2.3. Separation of Control Information	13
2.4. Database Operations	14
2.5. Summary of Knowledge Representations Used in Related Systems	14
Chapter 3 The User Interface	16
3.1. Introduction	16
3.2. Philosophy of the Interface	16
3.3. Historical Development of Interface Tools	18
3.3.1. The Display Module	19
3.3.2. Advisecore	19
3.3.2.1. The Screen Manager Process	21
3.3.2.2. The Window Filter Process	22
3.3.3. The SunWindow Interface	22
3.4. The Current User Interface	22
Chapter 4 The Network Editor	29
4.1. Introduction	29
4.2. The ADVISE Network Representation	29
4.3. Features of the Network Editor	30
4.3.1. Screen Representation of Networks	30
4.3.2. Local Editing Options	31
4.3.3. Global Editing Options	33
4.4. A Sample Interaction	34
Chapter 5 The QUIN Subsystem	43
5.1. Introduction	43
5.2. The Relational Model	44

5.2.1. Relational Tables	44
5.2.2. Keys	45
5.2.3. Normalization	46
5.2.4. Relational Operations	46
5.3. Table Creation	47
5.3.1. Define	47
5.3.2. Define Event	48
5.3.3. Add	49
5.4. Table Retrieval	50
5.4.1. Relational Table Expressions	51
5.4.2. VL Conditions	54
5.4.3. Ordering of Rows	54
5.5. Table Modification	54
5.5.1. Change	55
5.5.2. Delete	55
5.5.3. Save	55
5.6. Inferential Operators	57
5.6.1. Fetch and Results	57
5.6.2. Cluster	58
5.6.3. Diff	59
5.6.4. AQ11	63
5.6.5. Esel	63
5.6.6. Varsel	64
5.6.7. Varcon and Varcont	64
5.6.8. Other Operations	64
5.7. Macro Language	65
5.8. Program Description	65
Chapter 6 The Rule Parser	66
6.1. Introduction	66
6.2. Language for Rule Representation	66
6.2.1. Rules	66
6.2.2. Right Hand Sides (RHS)	66
6.2.3. Left Hand Sides (LHS)	67
6.2.4. Rule Groups	67
6.2.5. Variables	68
6.2.6. Functions	68
6.2.7. Properties	72
6.3. Parser Construction	72
6.4. Basic Features	72
6.4.1. The PARSER Production Cycle	72
6.4.2. Operation and Use of the Parser	73
6.5. Updating the PARSER	75
Chapter 7 The Rule Evaluator	76

7.1. Introduction	76
7.2. User Procedures	76
7.3. Configuring the Evaluator	78
Chapter 8 Knowledge Base Paraphrasing	80
8.1. Design of a General Paraphraser	80
8.2. The Existing Rule Paraphrase Module	82
Chapter 9 The Tuple Manager	83
9.1. Introduction	83
9.2. The Data Types	83
9.3. The User-Visible Procedures	84
9.4. An Important User-Invisible Procedure	87
Chapter 10 General Utilities — TESTER and the Special Functions Module	88
10.1. The Low Level Network Debugger	88
10.1.1. How to Use the Tester Program	88
10.1.2. Tester Commands	89
10.2. The Special Functions Module	91
Chapter 11 The PLANT/ds System: Rule Based Inference Control	93
11.1. Introduction	93
11.2. The Knowledge Base	93
11.3. Control Scheme	94
11.4. Using the PLANT Program	96
11.4.1. Data Structures	96
11.4.2. Control Scheme Tools Package	100
11.4.3. PLANT/ds Tools Package	101
Chapter 12 The PLANT/cd System: Rule Based Inference Control	105
12.1. Introduction	105
12.2. User Description	105
12.3. Control Scheme Details	105
12.4. Future Research Goals	108
Chapter 13 The BABY System: Context Driven Data Annotation	110
13.1. Introduction	110
13.2. Clinical Perspective	111
13.2.1. Role of BABY in Diagnosis	111
13.2.2. Clinical Patterns	112
13.3. Clinical Context and Patient State	114
13.4. Knowledge Engineering Environment	114
13.4.1. Clinical Data	114
13.4.2. Uncertainty in Medical Data	115
13.4.3. The Knowledge Base	115
13.5. Baby Implementation	115
13.5.1. User Interface	116
13.5.2. The Knowledge Representation	116
13.5.3. The Inference Engine	119

13.5.4. The Patient State	120
13.5.5. Variable Suggestion	120
13.5.6. Network Parser	121
13.6. Conclusion	121
Chapter 14 The ALFALFA System: Entomological Pest Identification	123
14.1. Introduction	123
14.2. Preconditions and Termination Conditions	123
14.3. Multigoals	125
14.4. Disjunction on the Right Hand Side of Rules	126
14.5. Data Acquisition	127
14.6. Additional Research Goals	127
14.6.1. Describing Variable-Value Relations with Semantic Nets	127
14.6.2. Dynamically Changing Value Sets for Variable	129
14.7. Future Research Extension	131
Chapter 15 Summary and Future Work	133
15.1. The Language of the Future	133
15.2. Extension of Present Efforts	134
Acknowledgements	135
References	136
Appendix A Rule Parser Details	141
A.1. Basic Lexical Notions	141
A.2. Error Information	141
A.3. Mark Type Definition	142
A.3.1. Within a VARS Block	142
A.3.2. Within a RULES Block	143
Appendix B GVL, Grammar	145
Appendix C Internal Representation of a BABY Network	153

Figures

1. A conceptual diagram of the ADVISE system.....	3
2. A technical diagram of the ADVISE Architecture.....	9
3. Multi-question frame from PLANT/ds.....	17
4. Single question frame from PLANT/ds.....	18
5. Screen Management in Adviscore.....	20
6. The textual representation of tuples for "arches".....	42
7. Operations available in QUIN.....	43
8. The knowledge refinement cycle.....	44
9. Clinical Laboratory Values.....	45
10. Results of Selection Operation	46
11. Results of Projection Operation.....	47
12. Format of File "vls"	49
13. Result of Retrieval	50
14. Relational Table Expression Operators	51
15. The "spec" Table.....	51
16. Join of "spec" and "tests"	52
17. Projection & Join.....	53
18. Ordinary Projection	53
19. Projection & Count.....	53
20. Projection with Redundancy.....	53
21. Use of "min".....	54
22. Original Table to be Changed.....	56
23. Table with Modified Hgb Column.....	56
24. Clustering Criteria	58
25. Parameters Table.....	60
26. Criterion Table	60
27. Events to be Clustered	60
28. Meanings of Values in Events Table	61
29. Results of Clustering	61
30. GEM Parameters Table.....	62
31. GEM Events Table.....	62
32. Domain Values or Days of the Week.....	62
33. A LHS.....	67
34. Relation of Compiler Components.....	74
35. A Sample Paraphrase Program.....	81
36. Flowchart depicting the top-level structure of PLANT/cd.....	106
37. Flowchart depicting the FINDOUT procedure.....	107
38. Flowchart depicting the INFER procedure.....	108
39. System Structure of BABY.....	112
40. A BABY Knowledge Base Pattern	118
41. Network Description of Figure 40.....	122
42. Goal Structure of the ALFALFA System.	124
43. Abstract Outline of Algorithm Converting Variable/Value Graph to Rules.....	128

44. The Case of a Common Subtree in Variable/Value Graphs.....	129
45. Typical Order Independent Rules Derived from Variable/Value Graph.....	130

CHAPTER 1

A General Overview

Expert systems are now being built for a great variety of practical problems. The problems span many domains, including medicine, geology, chemistry, agriculture, defense, computer systems design, accountancy, architecture and VLSI circuit design. The systems built today are often representative of technology developed in the past decade, and suffer from a number of limitations. They typically use only one form of knowledge representation (e.g. production rules or semantic network), have no or extremely limited learning abilities, use only one type of inference mechanism and one type of control strategy, offer poor explanation facilities, and do not provide adequate mechanisms for interaction with databases and/or simulation models.

The research described here represents our attempts to overcome some of these limitations. We have developed an architecture and design ideas for a general purpose inference system called ADVISE. The system consists of a set of modules for the development of and experimentation with expert and advisory systems for various application domains (hence, a "meta-expert system").

This report describes the design philosophy behind the ADVISE system, presents results on its first implementation called ADVISE.1, and describes its application to various test problems. Among the important and novel ideas in ADVISE.1 are the incorporation of three different types of knowledge representation, inductive learning capabilities and several inference control strategies. We feel that ADVISE.1 represents a substantial advance in expert system design and functionality and can be viewed as a "second generation" expert system building tool.

Let us examine the notion of an expert system. An expert system is typically defined as a computer system that exhibits high performance in a specific problem domain due to a large amount of formally encoded domain knowledge, is able to conduct formal reasoning on this knowledge, and provides facilities for explaining its reasoning to a user. An expert system is supposed to perform a host of tasks that a human expert would typically perform: diagnose, interpret, consult, classify, identify, search through a space of possible solutions, explain and analyze [Buchanan, 1983].

In a typical expert system, the domain knowledge is represented as a set of production rules. These rules take the form of:

IF <condition> THEN <action> : α

where:

<condition> is a logic expression which must be satisfied to a certain preset degree, and

<action> is a set of actions to be performed if the <condition> part of the rule is satisfied. The <condition> may include both the context within which the rule is applicable, and the specific conditions to be satisfied by the situation in order to apply to it the <action>.

< α > is a certainty or strength that, together with the strength of the <condition>, affects the strength of assertions made in <action>.

The set of production rules that characterize a given application area is termed the *knowledge base*. The knowledge base is one of the three major components of an expert system.

The second major component is the inference mechanism by which the knowledge base is used to perform given tasks. The inference mechanism is divided into two parts:

- a *rule evaluator/executer* that does inference *within* a rule, and
- a *control scheme* that does inference *among* rules.

In many expert systems, some form of *probable* or *plausible* reasoning is used to handle uncertainties. Data are often represented as *object/value/confidence* triples.

The third major component of an expert system is the memory needed to store intermediate results of rules when they are actuated or *fired*. Some architectures term this component the *blackboard*.

A favorite starting point for expert system architecture is to organize these three components as a production system. Such a production system performs a *recognize-act* cycle in which a control scheme decides which rules to evaluate and, if any fire, executes their right-hand sides. For more details on production systems see [Nilsson, 1980], [Davis, 1978] and [Michie, 1980]. Many expert systems are not pure production systems, however.

There is another expert system architecture in which the expert knowledge is in the form of a network. The control scheme becomes a network traverser/updater. This architecture is implemented in PROSPECTOR [Duda, 1978a]. See [Hayes-Roth, Waterman and Lenat, 1983], [Gevarter, 1982], [Michie, 1980], and [Stefik, 1982] for more detailed overviews of expert systems. The following section shows how the ADVISE system extends the expert system featured described above.

1.1. Goals and Major Features of ADVISE.1

The goals of research on the ADVISE system are to explore basic problems, and to test the feasibility of building an advanced knowledge engineering workbench that integrates multiple knowledge representations, multiple control and rule evaluation schemes, and inductive learning capabilities. Major features of the current version ADVISE.1 include:

- representation of knowledge in three different forms: a rule base, a network base, and a relational data base,
- use of a very flexible representation for inference rules,
- modular design,
- availability of several control strategies,
- inductive learning capabilities,
- common virtual memory representation for data,
- several rule evaluation schemes,
- separation of control and meta-knowledge from domain knowledge,
- independent module for user interaction, and
- implementation of the system in Pascal (a popular language widely available on many computer systems),

As this list shows, there is an emphasis on user and developer flexibility and convenience in ADVISE. The user interface is designed as an independent module and supports the idea of user-oriented interaction. This interaction can be developed almost independently of the rest of the system and supports multiple terminal types.

ADVISE is designed with the philosophy that a highly modular system can be arranged into different configurations, best fitting different applications. Each module can be independently modified, and new modules readily incorporated into the system. This modular design makes ADVISE flexible and elegant.

1.2. Conceptual Organization of ADVISE.1

Figure 1 shows a conceptual diagram of the system. The system consists of four major components:

- (1) Control Block and User Interface
- (2) Knowledge Base
- (3) Query Block

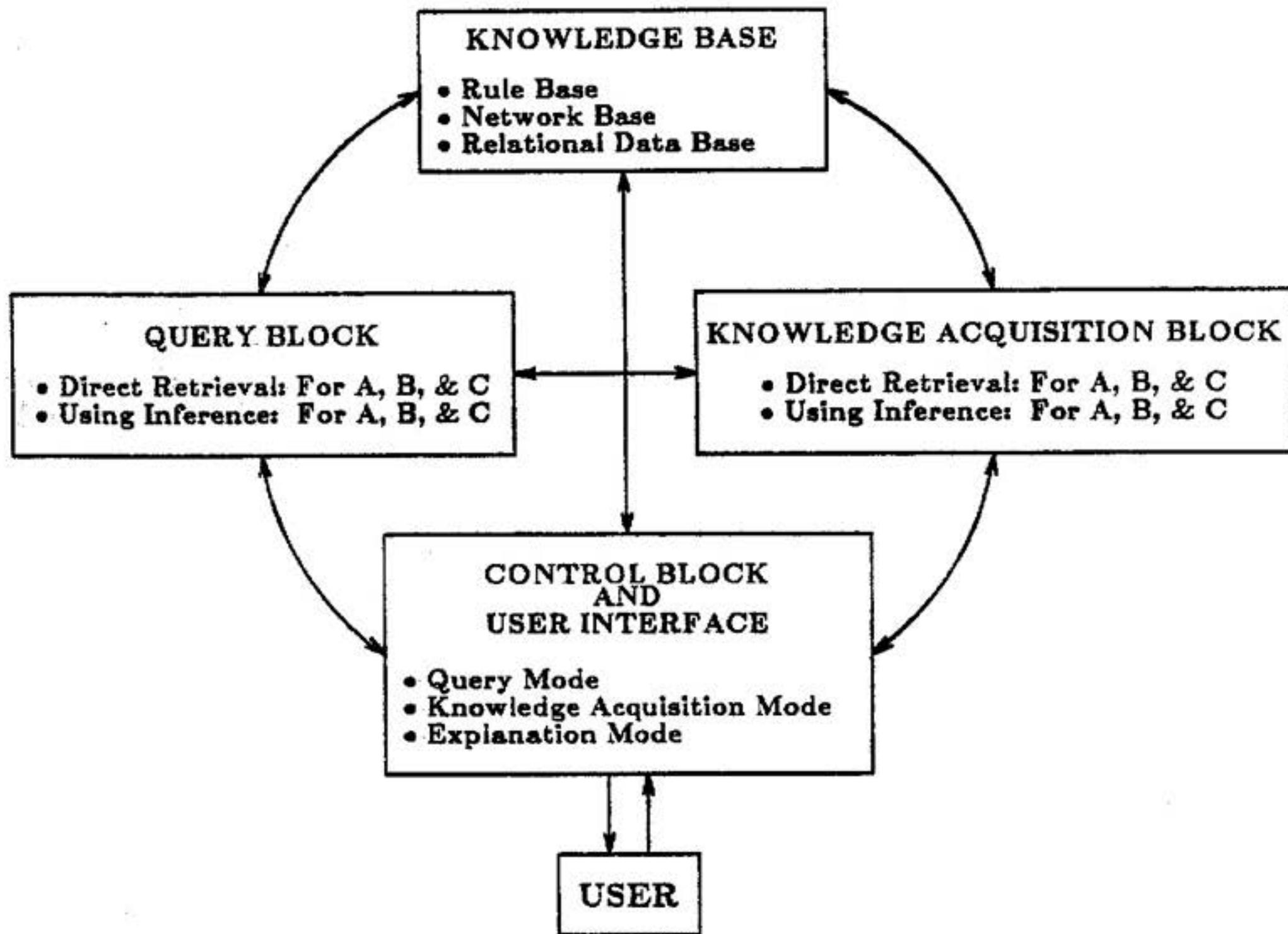


Figure 1: A conceptual diagram of the ADVISE system.

Each component supports operations on three types of knowledge representations: a rule base, a network structure, and a relational data base. The functions of these four components are described in the sections below.

1.2.1. Control Block and User Interface

The control block and user interface manages the user's interaction with other components of the system. It handles three distinct modes of operation of the system:

- Query mode
- Knowledge acquisition mode
- Explanation mode

The user interface portion of the system provides utility routines to manage the terminal screen and supports the explanation of internal data representations and operations in a textual and graphic form.

Query mode (Q-mode)

Query mode is used during consultation. In this mode, the system:

- selects questions to ask the user,
- accepts user answers and
- conducts an inference process involving the knowledge base and information provided by the user, in order to compute advice with an associated strength of supporting evidence.

In one respect, the consulting portion of the system is radical in design. In particular, there is no single problem solving strategy. Rather, local problem solving behavior is defined by the choice of an evaluation scheme, and global problem solving behavior is governed by a control scheme within and among rule groups. Specific provision has been made in the design for the inclusion of multiple control schemes and multiple evaluation schemes.

Knowledge acquisition mode (K-mode)

The knowledge acquisition mode coordinates both the encoding of expert derived rules into the knowledge base and the interactive invocation of the separate induction programs. This mode includes handling specific components for defining expert rules, manual refinement of rules, induction of rules from examples, and automated correction and improvement of the rules. The system also provides facilities for testing rules in interactive mode on individual cases, as well as in batch mode on a collection of cases.

Explanation mode (E-mode)

The explanation mode paraphrases decision rules in English, enables a user to understand the organization and functioning of the system in query and knowledge acquisition modes, allows simple interrogation of the contents of the knowledge base, and displays the steps in the process which led to a given piece of advice.

1.2.2. Knowledge Base

The network structure consists of three types of representation:

- a network base (e.g., conceptual network),
- a rule base, and
- a relational data base.

A unified "access protocol" is used for all three types of representation.

Network Base

The network base contains network structures representing general domain knowledge about interrelationships among various conceptual units. For example, it can include hierarchies of terms from the application domain indicating the level of generality of such terms (a "generalization tree"), a representation of the structure of the rule base, and precedence relations defined over questions for the user. Links between nodes in the network represent "static" relationships between concepts. The network organization is a form of the "Logic Net" described in [Baskin, 1980].

Rule Base

The rule base contains rules in the basic form:

CTX CONDITION ::> CONCLUSION : α, β

where:

CTX is an expression describing the context within which the rule is applicable;

CONDITION is a formal expression (in VL2 [Michalski, 1978]) which involves elementary conditional statements (called "selectors"), linked by various logic operators (including quantifiers);

CONCLUSION defines the decision or action which is executed when the **CONDITION** is satisfied by a given situation;

α is the strength of the evidence which supports the **CONCLUSION** when the **CONDITION** is completely satisfied ($0 \leq \alpha \leq 1$) and

β is the strength of evidence which supports the negation of **CONCLUSION** when the **CONDITION** is not satisfied ($0 \leq \beta \leq 1$).

The rule above is read: **CONDITION** implies **CONCLUSION** with forward strength " α " and backward strength " β ". Specifically, the rule states that: if the left hand side (LHS) of the decision assignment operator (::>) is satisfied, then the right hand side (RHS) is asserted with a degree of confidence α , and if the RHS is satisfied, then the LHS is asserted with a degree of confidence β . The decision assignment operator is equivalent to logical implication when $\alpha = 1$ and $\beta = 0$, and is equivalent to logical equivalence when $\alpha = 1$ and $\beta = 1$. The above rule is equivalent to the following rule group:

CONDITION	::>	CONCLUSION	: α
not CONDITION	::>	not CONCLUSION	: α
CONCLUSION	::>	CONDITION	: β
not CONCLUSION	::>	not CONDITION	: β

By providing both " α " and " β " for each rule, it is possible to use rules for inference in both forward and backward directions.

The use of parameters (α, β) above can be illustrated by an example taken from contract bridge. In bridge, each partner learns the same set of rules for bidding (called a bidding convention). Qualifiers such as "must," "should," "usually," "seldom" and "never" are used in the verbal description of the rules for the strength of implication. A bidder uses the rules in the forward direction to decide what bid to make. The partner of a bidder uses the rules in the reverse direction to decide about the strength of bidder's cards.

One well known rule in the Standard American bidding convention is an opening bid of one no trump. A textbook description of this rule can be paraphrased:

If you have a strong hand (16 to 18 high card points),
 a balanced hand (at worst one doubleton suit), and strength in
 all four suits (four stoppers) then you should *definitely* bid 1
 no-trump. (i)

This rule can be expressed in the formalism which we are developing as:

[high card points = 16..18] [number of doubletons ≤ 1]
 [number of stoppers = 4]
 ::>
 [bid suit = no_trump] [level = 1] : $\alpha=1, \beta=0.8$ (ii)

where $\alpha=1$ follows from the use of "definitely" in the rule. The value of " β " is the strength with which a bid of one no trump implies the hand described on the left hand side of the rule. The value of " β " is less than 1.0 because one no trump may be bid with other hands, as shown below.

The novice bridge player soon learns that like any other expert skill, bidding in bridge does not always follow strict rules, as described above. An experienced player will often use the following more general rule:

If you have a hand which meets the above requirements
 for a 1 no-trump bid, except that it has less than three stoppers,
 then you should *usually* bid 1 no-trump. (iii)

The rule above can be represented in our formal language as:

[high card points = 16..18] [number of doubletons ≤ 1]
 [number of stoppers ≥ 3]
 ::>
 [bid suit = no_trump] [level = 1] : $\alpha=.8, \beta=1$ (iv)

The parameter $\alpha=.8$ in the rule above indicates that if the left hand side of the rule is satisfied then the strength of confidence for a 1 no trump bid is somewhat less than it was in (ii). On the other hand, once this bid has been made, the partner can be certain ($\beta=1.0$) that the bidder's hand conforms at least to the requirements specified in the left hand side.

The rules (ii) and (iv) above can be combined into a single rule by using "weights" associated with individual values of the variables. The following rule is a formal representation for both (i) and (iii) above:

```
[high card points = 16..18] [number of doubletons ≤ 1] (v)
[number of stoppers = 3:0.7, 4:1.0]
::>
[bid suit = no_trump] [level = 1] : α=1, β=1
```

The coefficient 0.7 associated with "number of stoppers = 3" reflects the weakened strength of implication if there are three stoppers. Both the weights on the "number of stoppers" and the value of " α " are used to determine the strength of supporting evidence for the CONCLUSION. Since the weights have been associated with individual values, the global strength of implication, " α ", can be assumed to be 1.

Relational data base

The relational data base contains relational tables which represent any factual information, e.g., examples of experts' past decisions. A modified relational algebra has been developed using constructs from Variable-valued Logic to make the user access more natural and more concise [Michalski, 1974] [Spackman, 1982].

1.2.3. Query Blocks

The query block supports two types of queries: those which can be executed by direct retrieval from the knowledge base, and those requiring inference. Both types of queries are defined for all three types of representations in the knowledge base.

Query block using direct retrieval

Direct retrieval is used to display the contents of the knowledge base. The network base, the rule base, and the relational data base can each be retrieved and displayed using the query block. Direct retrieval is heavily used in the explanation mode of the system and also during knowledge acquisition.

Queries using direct retrieval on the relational data base include the traditional relational table operations such as as "project," "select" and "join" as well as various arithmetic or other transformations of the data items in the tables.

Query block using inference

One of the most important functions of the system is to compute the most plausible advice for the user in a specific situation. Queries involving inference (deductive and/or inductive) are supported for each form of knowledge stored in the knowledge base.

Queries using inference for the network involve "path following" within the network, e.g., "climbing the generalization tree." Inference over the network also occurs whenever the hierarchy is used to answer questions by searching the network for specified relationships between given concepts.

Queries using inference for the rule base are particularly important for providing expert advice. Rules are evaluated for a given situation using an "evaluation scheme" (a method of propagating uncertainties) in the order decided by a "control scheme." In this system, there is no single problem solving strategy. Rather, local problem solving behavior is defined by the choice of an evaluation scheme (of which several are planned), and global problem solving behavior is governed by a control scheme within and among groups of rules. We plan to implement several different evaluation schemes and a few control schemes in the system. In this way, a single knowledge base can be used for research into the performance

of differing problem solving strategies for consultation.

The QUIN relational data base (Chapter 8) has been enhanced to allow queries using inference. These additional operations allow the inference of rules from examples.

1.2.4. Knowledge Acquisition Block

The knowledge acquisition block supports knowledge acquisition by the direct representation of knowledge provided by human experts and also by inductive inference from facts provided to the system. The design of the proposed system includes knowledge acquisition for each type of knowledge stored in the knowledge base.

Knowledge acquisition by direct representation

The direct representation of networks, rules, and data bases of facts or examples constitutes "learning by being told." This is the major way that knowledge bases are being constructed today. The direct representation of expert knowledge is particularly important in problem areas where "rules of thumb" or other generalizations about the problem domain are known. Material which is supplied by the expert for direct representation is entered into the knowledge base as specified, but it may be modified by further knowledge acquisition using inference as described below. Special user interface ("debriefing") for interactive specification of rules and networks is provided. Batch submission of sets of examples is supported as well.

Knowledge acquisition using inference

The inclusion of machine based inference as a part of the knowledge acquisition process is intended to reduce the burden on human experts. By defining inference procedures over each component of the knowledge base, the system no longer relies solely on the human expert to organize and present a complete, concise, and error free knowledge base.

Knowledge acquisition using inference for the network involves both deductive extensions to the knowledge base and the inductive derivation of new or improved network structures. As new information is added to the network base, rules of inference are "executed" to enforce the logical consistency and completeness of the network. This corresponds to inference of new concepts and relationships from existing ones. Also, machine derived categorizations and hierarchies can be inferred from the knowledge base using conceptual clustering techniques implemented in the program CLUSTER [Stepp, 1980].

Knowledge acquisition using inference for rules includes the derivation of new or modified rules from existing rules or groups of rules using examples in the relational data base. This process will be implemented by adapting the already developed inductive learning programs AQVAL [Michalski, 1973], INDUCE [Larson, 1977] and ID3 [Quinlan, 1979]. Preliminary results by both principal investigators in this area suggest that the "refinement" of existing rule bases using induction is both possible and fruitful [Michalski and Chilausky, 1980] [Baskin, 1978]. The derivation of patterns of rules or rule groups from the isolated rules is also supported by the proposed system. Although results in this area are still tentative, the inference of patterns in groups of rules should allow the iterative improvement of the control scheme by which rules are selected for evaluation.

Knowledge acquisition using inference for the data base corresponds to a form of clustering. In our previous experiments, clustering algorithms have been used to partition the set of examples into groups which are "similar." This operation corresponds to having

a teacher label each example in the training set. The important property of the clustering algorithms is that they do not rely on a domain expert to categorize the examples. In addition, when the categories are not known, the clustering process can still be used. (The clustering operation corresponds to inductively deriving a relational operation which a human expert might have used to supply examples for the inductive derivation of rules.)

In addition to clustering, inference over a set of examples involves selection of the best "Representative" examples for use in the inference of rules. The program ESEL [Michalski, 1978] has been developed for choosing examples from a large set of examples which are "representative" of the set. The proper selection of examples becomes particularly important when rule inference is computationally expensive.

1.3. The Architecture of ADVISE.1

Figure 2 illustrates the architecture of the ADVISE system from a technical standpoint. Figure 2 shows the basic software modules in the ADVISE system and their interactions. A short description of each module is presented below:

- **USER INTERFACE**

The user interface provides a set of software tools for creating sophisticated user

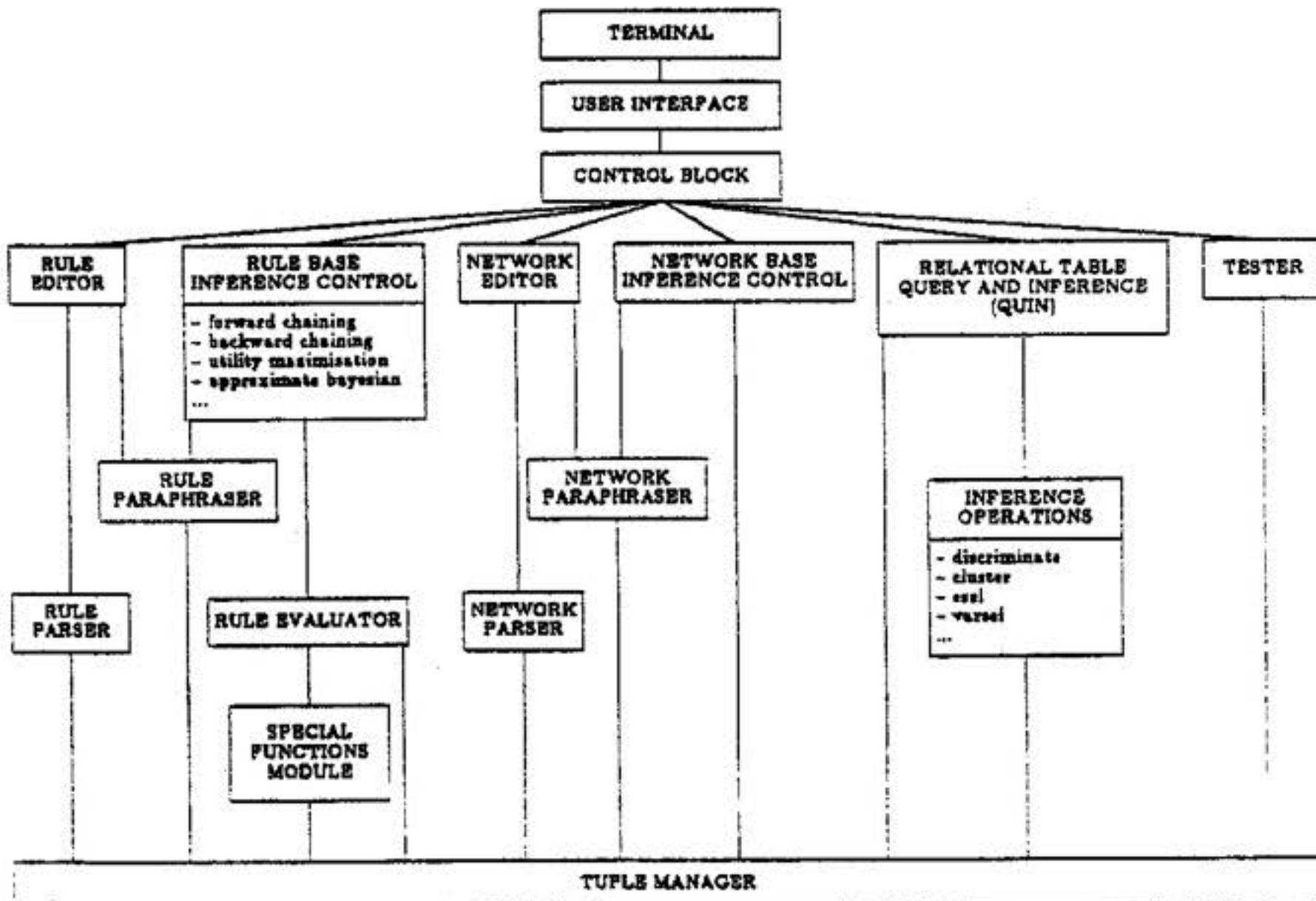


Figure 2: A technical diagram of the ADVISE Architecture.

interactions with the ADVISE program. It supports graphics, windows and menu driven user input. It is designed primarily to drive the Sun-2* Workstation, but is designed to be portable with minimum modification. The user interface is discussed in detail in Chapter 3.

- **CONTROL BLOCK**

This dispatches the major functions of ADVISE: parsing rules or networks using the parser modules, running a consultation using the rule base or network base modules, relational table query and inference using the QUIN module, and testing parts of ADVISE using the TESTER module. Currently the control block functions are implemented in a top-level window created using the user interface above. This top-level window is also discussed in Chapter 3.

- **RULE EDITOR**

This module is an interactive menu-driven editor for rules. It uses both the rule parser and paraphraser. This module has been designed but not implemented.

- **RULE BASE INFERENCE CONTROL**

The ADVISE system supports several types of knowledge base inference control (control schemes). They access knowledge in the tuple manager, request values for unknown variables and determine the truth value of pieces of knowledge via the rule evaluator. Control scheme tools are described in Chapter 11 (Section 11.3).

- **RULE PARSER**

This module parses an extended form of GVL₁ [Michalski and Chilausky, 1980]. It outputs a parse tree that is read by the TUPLE MANAGER and stored in the tuple format. See Chapter 6.

- **NETWORK EDITOR**

This module is an interactive menu-driven editor for networks. Its implementation is discussed in Chapter 4.

- **NETWORK PARSER**

An inference network is an alternate form of representing domain knowledge. This module parses the network representation into the tuple representation manipulated by the tuple manager. The network parser is discussed under the implementation of the BABY system in Chapter 13.

- **NETWORK BASE INFERENCE CONTROL**

This module, when implemented, will be used to carry on inference on networks.

- **QUIN**

This module is used to do queries and inference on relational tables. QUIN [Spackman, 1982] calls a host of data analysis and learning modules such as AQ11 [Michalski, 1978], ESEL [Michalski, 1978], CLUSTER [Stepp, 1980], PROMISE [Baim, 1982], and CONVART [Davis, 1981]. See Chapter 5.

- **TESTER**

This module is used to manipulate the tuple network directly. It also serves as a testing and debugging vehicle during the development of other modules in the system. See Chapter 10 for further details.

- **PARAPHRASERS**

These modules are responsible for explaining to the user how rule and network inference is

being used during a consultation. Another function of the PARAPHRASE module is used to "unparse" a rule from its parse tree form to the human readable form. This module is also used to explain the evaluation of particular rules to the user. Paraphrasing is discussed more in Chapter 8.

- **RULE EVALUATOR**

This module is responsible for evaluating the premise part of a rule and asserting its consequent if it fires. It evaluates and asserts rule parts under a variety of semantics for logical connectives in a multi-valued logic interpretation. See Chapter 7.

- **SPECIAL FUNCTION EVALUATION**

This module, also known as the TRAP module, is used to evaluate special functions. These functions can do such things as permit special displays, start up sensors, run models or simulations, etc. See Chapter 10 for a detailed discussion of the TRAP module.

- **TUPLE MANAGER**

The basic structure for storing information in ADVISE is the tuple. (Information is also stored in Pascal local variables, but this type of data is particular to the local environment and is not meant to be preserved.) An ADVISE tuple resembles the mathematical definition of a tuple (an ordered set) in which a set of tuples form a graph or network. Tuples, much like LISP property lists, are accessed by context. The tuple manager is based on the work in [Baskin, 1980] and is discussed in detail in Chapter 9.

The overall system architecture outlined in this section has served as a guide for the development of the ADVISE system. Before presenting a detailed chapter-by-chapter account of the current state of ADVISE, the next chapter reviews other expert systems technologies.

CHAPTER 2

Research Goals of ADVISE and a Brief Review of Other Systems

Recent literature ([Buchanan, 1985], [Harmon, 1986], [Spang, 1986] and [Riedesel, 1986]) documents in the neighborhood of 150 expert systems as being in use. These systems serve a very wide range of applications - such as advising nuclear power plant operators, configuring computer systems, detecting harmful drug interactions, identifying new herbicides and forecasting finance. One expects that many more lesser-known systems are in existence or are being developed. This is the context in which the role and plan for ADVISE research will be discussed.

At the time of its conception, the ADVISE system presented at least three revolutionary ideas in expert systems technology:

1. multiple knowledge representations (to combine a rule base with a relational database and a network base)
2. multiple control strategies
3. inductive learning tools for learning rules and automatically constructing classifications.

The design and implementation of ADVISE has emphasized the construction of a knowledge engineer's workbench. Interchangeable system components have been developed and made available to the knowledge engineer. The ADVISE effort is directed at creating and evaluating different expert system frameworks rather than providing a robust expert system shell for a specific approach. The ADVISE formalism provides multiple options for rule scheduling, uncertainty propagation, and knowledge representation. In this way, ADVISE can support a wide range of expert system methods.

In overall design and approach, ADVISE is most similar to AGE (distributed by SUMEX) and DUCK (Smart Systems Technology)*. ADVISE is in strong contrast to "empty" expert systems such as EXPERT (Rutgers University), OPS5 (Carnegie Mellon University or Franz, Inc.), and EMYCIN (SUMEX). These systems have more strictly defined structure within which the knowledge engineer must operate.

The AGE system consists of a modular set of tools that allow one to create and evaluate different expert system frameworks. Like ADVISE, the AGE system has predefined control frameworks which can be used to build specific expert systems. AGE is implemented in Interlisp.

DUCK provides support for rule-based knowledge systems using predicate calculus rules. It is not meant to be used as a stand-alone programming environment; rather, it is a package of utilities written in LISP that are useful for writing expert system application programs.

The remainder of this chapter discusses particular aspects of ADVISE and their relation to other systems.

*Information on many of the systems cited in this chapter was obtained by personal inquiry to the distributor and not through existing literature. For these systems, the source of information is provided instead of a reference.

2.1. Knowledge Representation

Although there has been work in ADVISE with network representations for knowledge (e.g. the system BABY — see Chapter 13), the primary knowledge representations in use consist of tables and rules. Both have been used in multiple expert system implementations. The relational operations in QUIN (see Chapter 5) are not directly incorporated into the rule evaluator and, thus, are not easily integrated into the operation of expert systems. QUIN has proven usable for interactive management of small databases for the purpose of rule generation and refinement.

As noted in Chapter 1, the rule representation used for ADVISE is GVL rules with positive and negative weights allowed on individual selectors and linear modules. Alpha and beta weights are similarly attached to the implication arrow in the rule. In general, GVL rules are larger than the types of rules used in other systems. They tend to contain more terms and there tend to be fewer of them to cover the same event space. The GVL rules are correspondingly more sophisticated to evaluate and are more likely to suspend execution because of missing data than the simpler rules in other systems.

ADVISE is different from many systems in its rich support of disjunction. On the left hand side of a rule, both internal and external disjunction are supported. In other systems, these disjunctions would lead to separate rules. The problem of disjunction on the right hand side of a rule can be handled by backtracking (see Chapter 14) or by using a truth maintenance system (TMS).

Current ADVISE does not support an explicit blackboard model. Global communication is accomplished by defining and using variables in a global variable block.

2.2. Structure Within a Rule Base

Two types of structure can be distinguished for a rule base: static vs. dynamic. Static structure can be defined before a consultation begins and used as part of the planning process and in rule scheduling. Dynamic structure is imposed at run time usually by the instantiation of such constructs as viewpoints or contexts. ADVISE does not provide general support for dynamic structure but does provide support for static structure through the use of rule groups and variable blocks. A data-directed context mechanism was developed for the BABY system (Chapter 14) but was not incorporated as a general tool.

A rule group is a functional block of rules that achieve a common goal. The rule group operates on either global or local variables which are imported into the rule group through a variable block declaration.

2.3. Separation of Control Information

The ADVISE system attempts to strongly differentiate high-level rule scheduling and control from low-level rule evaluation. In this way, the evaluation of rules has a minimum effect on the scheduling of rule activity. As much control information as possible is separated out of individual rules and contained in a "control scheme." In ADVISE, these control schemes are mutually exclusive for a single knowledge base but may be selected separately for separate knowledge bases. ADVISE has supported bayesian control, forward chaining, backward chaining, and a utility measure based control scheme. Only the utility measure scheme is unique to ADVISE. Most expert system tools today incorporate both forward and backward chaining and a few like ADVISE and KES II (Software Architecture and Engineering, Inc.) support multiple competing control schemes.

2.4. Database Operations

Relational database operations are supported in ADVISE as part of the QUIN system. The operations supported in QUIN are only available for interactive knowledge acquisition and are not available to the GVL rules. The induction operators provided for in QUIN are unique in expert system toolsets. The induction operators in RuleMaster (RADIAN Co.) come the closest to those supported in ADVISE.

A number of systems provide for using a relational database as part of expert system building — e.g. ART (Inference Corp.), Knowledge Craft (Carnegie Group), and Knowledge Workbench (Silogic Inc.). ADVISE provides "trap functions" which can be used to reach external databases during a consultation. Knowledge Workbench appears to provide a more convenient interface to external databases, but it is no more powerful than the one supported in ADVISE.

2.5. Summary of Knowledge Representations Used in Related Systems

This section gives a brief account of knowledge representations used in several well-known systems.

The AGE system is distinguished by a flexible knowledge representation which separates control, procedural, static, and solution knowledge. AGE has the ability to automatically pursue multiple goals.

The ART system incorporates a frame-based knowledge representation component with inheritance and a relational database for use in rule-based inference. A blackboard is used for communication between rules and object-oriented programming is accomplished via procedural attachment to slots in the frames. An assumption-based TMS is used to represent different assumption sets in the different viewpoints (contexts).

DUCK provides tools for expressing knowledge in predicate calculus rules and supports deductive database retrieval. Non-monotonic reasoning is supported via a justification-based TMS that supports viewpoints (contexts).

EXPERT is a FORTRAN-based system that provides decision rules to link findings to hypotheses. It differentiates between rules that link facts to other facts, facts to hypotheses, and hypotheses to hypotheses.

KEE (IntelliCorp) incorporates object-oriented programming, a frame system with taxonomic inheritance, rule-based reasoning with an assumption-based TMS. The system provides for a "rule class graph" which is similar to the concept of rule groups in ADVISE.

The KES II system is written in C and supports three different modes of reasoning:

- 1) production rules,
- 2) bayesian patterns, and
- 3) minimal set covers for a hypothesize and test subsystem.

Knowledge representation is structured in a simple semantic network formalism which is quite similar to the use of *tuples* in ADVISE. As in ADVISE, the three knowledge formats are different but have been partially integrated.

In the Knowledge Craft system, the representation language used is CRL, a descendent of SRL. The basic representations unit is a schema with a slot/filler structure. Relations between slots and inheritance of values are user-definable. Contexts are supported as is an integrated database management system with caching for storing schema.

OPS5 and its extended and faster cousin OPS83 (Production Systems Technology) support rule-based programming for expert system development. Data elements are

represented either as vectors or as objects with a list of attribute-value pairs. A blackboard is maintained for communication between rules. The blackboard also provides a natural representation for multiple goals.

CHAPTER 3

The User Interface

3.1. Introduction

This chapter describes the development and current state of the ADVISE user interface. The problem of human-computer interaction has received much attention [Human, 1985]. The study of computer user psychology, an outgrowth of earlier work in human-computer interaction, has emerged as a viable subfield in computer science [Moran, 1981]. A difficulty in working in this area is that so many factors are involved in designing interactions which typically cannot be pinned down with algorithms. The designer must rely on his own intuition, which can be remarkably deceptive. The aesthetic aspect of this problem (i.e., what is the most appealing way of presenting information) is most certainly a matter of personal taste. In view of these problems, our approach has been to refine our program interfaces by subjecting them to several cycles of group criticism. Some general guidelines that evolved from this effort are presented in the next section.

3.2. Philosophy of the Interface

Although computer scientists seem often to lose sight of this fact, the human user is a far more sophisticated "computing device" than any current computer program. With this in mind, we offer some general principles that have proved useful in the development of the user interface.

- **Feedback** : This can be as simple as verifying the user's input by prompting, or highlighting his response (in the case of option selection from a menu). If processing time between user responses is too long (as little as two seconds) it is important to indicate that the computer is doing something. This feedback should be placed somewhere near the current text insertion point. Particularly long response times are acceptable only if the user thinks that something particularly useful and important is being accomplished.
- **Help** : Help should always be made available to the user. With regard to expert systems, some indication of the status of the consultation should be provided. The program should distinguish between novice and experienced users. The novice should be provided with introductory material and perhaps more specific prompts. The user's name is requested prior to a session, and a history of interaction is maintained with the program, in order to determine his user class. The novice user learns a great deal about a program in the first encounter, and the program should respond to this within a single session. For example, explanatory prompts may be presented the first few times they are used and then become more brief as the consultation progresses.
- **Backup and Error Accommodation** : The user should be allowed to "undo" anything he/she has done. A related issue is that the user should be allowed to query the program about answers to previous questions. A useful mechanism with respect to user errors is to provide immediate feedback after a user response (a definitive statement of what has been requested).
- **Consistency** : The display should be logically laid out. Option menus should always be placed in the same area of the display. User typing should be minimized.

- **User Bandwidth :** We favor a block based interaction where a *block* is a single screenfull of information. Each block should provide multiple options and allow several questions to be answered before moving on to the next block.

These guidelines alone do not guarantee a successful interactive program. Each interaction should be analyzed to minimize false expectations in the user. This analysis should not be performed by the program designer, but by a variety of outside sources. To illustrate some of these concepts, some blocks from the PLANT/ds program are presented.

Figure 3 shows a multiple question block that is used to gather information concerning the leaves of a diseased plant to be diagnosed:

Premature Defoliation: () Absent () Present

Shotholing: () Absent () Present

Shredding: () Absent () Present

Withering and Wilting: () Absent () Present

Type "x" to register an entry.

Press SPACE bar to move forward to the next entry.

Press BACK SPACE to move back to the previous entry.

Press "?" to get help for the current question.

Press RETURN to terminate entries on this screen.

Figure 3 : Multi-question frame from PLANT/ds.

The user moves back and forth between questions, answering those appropriate to the situation. A single key press guides all of these functions. Help concerning these questions is always available.

Figure 4 shows a block that gets the value for a single variable. The user first enters the value (or any option), then he may or may not be asked to enter a confidence in that answer. After input is complete, the program prints the message that it is preparing the next question.

How would you describe the pattern of leaf spots growth?

- 1 - From edge of leaf inward.
- 2 - Scattered and plain.
- 3 - Scattered with concentric rings.
- 4 - Brown veinal necrosis.
- 5 - Does not apply.

b - go back one question

d - display a rule

e - explain how to answer the question

p - list pursued hypotheses	high	- >35%
-----------------------------	------	--------

r - list rejected hypotheses	low	- >35%
------------------------------	-----	--------

w - why is the question being asked	certain	- 100%
-------------------------------------	---------	--------

Type one number or letter indicating your choice

with confidence in the answer: c (certain) / n (high) / l (low)

Preparing next question

|-----.....

Figure 4 : Single question frame from PLANT/ds.

Although PLANT/ds manages to meet the guidelines stated above, its interface has become outdated by the advent of windows, mice, and workstations. The next section traces the development of the user interface from the beginning of the ADVISE effort, from the *blocks* of the *display module* to windows and pop-up menus.

3.3. Historical Development of Interface Tools

As long as ADVISE has existed, a user interface has been needed. Along with the evolution of the ADVISE computing environment, the user interface has developed considerably. The first attempt was called the *display module*. Many ideas emerged from the display module work to shape further development. The display module lacked a sophisticated degree of terminal independence which *Advisecore* [Channic, 1984], the second generation interface, sought to resolve. When development settled into the workstation environment, the third generation interface, the SunWindows* interface was born. Each of these interface generations is described briefly below.

*SunWindows is a trademark of Sun Microsystems, Inc.

3.3.1. The Display Module

The primary goal in the design of the Display Module was to provide tools for the construction of a user interface that is as independent as possible from the rest of the ADVISE system. By doing this, the interface designer is freed from concerns regarding the rest of the program and can concentrate on the user interface alone. It is important to be able to cycle through an interface design in order to arrive at "friendly" interaction.

With this in mind, the Display Module was developed as a separate, independent module. Some concepts which guided its development draw from the work of [Thursh, 1980]. In that work, the authors address the problem of developing a teaching tool for general and systemic medical pathology. The software that evolved, Blockr, views a man-machine dialog as consisting of a series of *blocks* of single screenfulls of text and diagrams. These *blocks* can point to each other or themselves, forming a very general network structure. *Blocks* are treated as data, consisting of text and instructions that, when interpreted, cause an organized page of information to be presented at the terminal. Since the system is data (*block*) driven, the interaction can be prepared independently of the program that interprets it. We have, in effect, an expert system for user interface design.

As mentioned earlier, the primary data structure manipulated by the Display Module is the *block*. Since *blocks* consist primarily of data, they can be created via a simple editing process. One consequence of the single screenfull of information organization is that it encourages the presentation of several questions on a single *block*. This is consistent with the goal of high bandwidth discussed in the previous section.

The Block Manager controls the interface between the *blocks* and the user program (in this case the Control Scheme). This interface was organized in co-routine fashion; the block manager would be invoked, cycle through a series of help *blocks* and return control to the Control Scheme for user input. All user input was passed to the Block Manager which would handle range checking and error processing. One component of the Block Manager, the Block Interpreter, was responsible for translating *blocks* into a form that can drive a specific output device. It had access to a set of low-level routines of the form consistent with the proposed CORE graphic standards [Foley, 1982].

3.3.2. Advisecore

As computer interfaces developed outside ADVISE, the display module became outdated since it had no facilities for overlapping windows or pop-up menus. The display module also lacked terminal independence. To utilize windows and menus and to facilitate terminal independence, Advisecore was developed. Advisecore was also based on the CORE graphic standard, primarily for (but not limited to) use with the SunCore graphics package on a Sun Workstation.* There were two major goals for Advisecore as described below.

Concurrent Processes with Status Reporting. Multiple active windows should be able to co-exist on a single screen if desired. Effective screen management must allow several processes to be active at any given time.

Processes Bound to Windows. In a window system, it is usually desirable to be able to choose the location and size of a window in which a program runs. Being able to do this without any modification to an existing program is also desired. To accomplish this, screen management must provide a means for running a program which runs on a standard terminal within a window of any reasonable size.

*SunCore and Sun Workstation are trademarks of Sun Microsystems, Inc.

In the Advisecore interface, screen management consisted of two aspects. The first is a screen manager process which handles window requests from all other modules. The second is a window filter process which allows a program to be bound to and run inside a window as if it were running on a standard terminal. Figure 5 shows how these two processes fit into the ADVISE system. The screen manager process handles all calls to display or receive

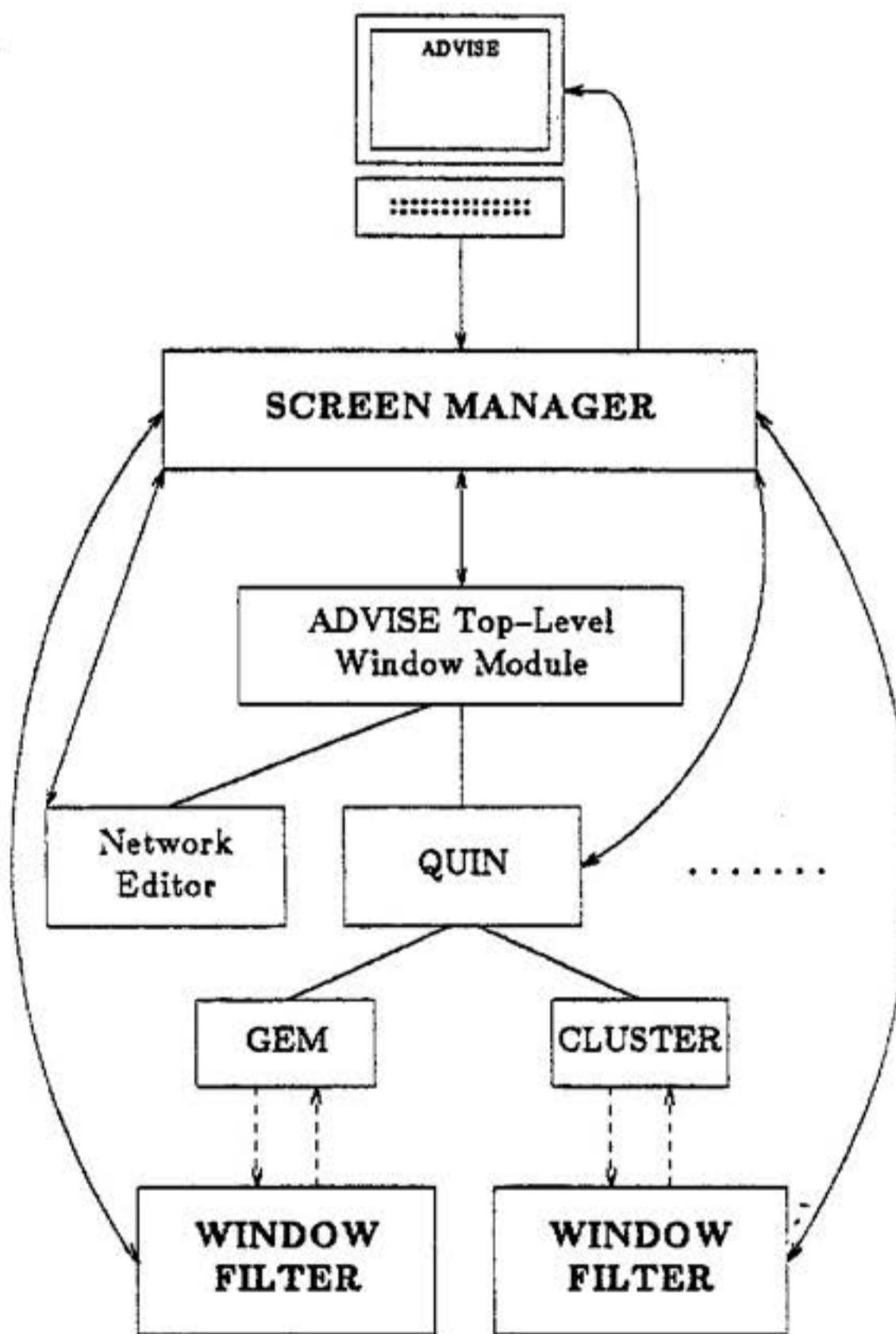


Figure 5 : Screen Management in Advisecore.

information to/from the user's terminal. If an ADVISE module does not use screen manager calls for such information, it must use an instantiation of the window filter process for communication with the screen manager process. Both the screen manager and window filter processes in their existing or proposed implementations are described in detail in the appropriate section below.

3.3.2.1. The Screen Manager Process

The main feature of the Adviscore screen manager is interprocess communication. As mentioned earlier, screen management loses a lot of functionality if multiple processes cannot be concurrently active within different windows. For this reason, some sort of interprocess communication is essential. The ADVISE screen manager uses sockets in the UNIX* operating system for interprocess communication. The sockets are represented in Figure 5 by bidirectional arrows. Sockets are a reliable means of two-way communication between processes. The ADVISE screen manager is designed to receive window requests from a process and also to deliver any input directed to that process from any window that process may have on the screen. This communication between a process and the screen manager is accomplished through a socket which is established during initialization of the Adviscore interface. Once the socket is established, further window routine calls merely send the request along with its parameters to the screen manager via the socket. The screen manager then executes the appropriate output action.**

Delivering input to a window is somewhat more involved. The screen manager must maintain a list of which windows belong to which processes and whether or not the process is awaiting input in those windows. At any given time, only one window is active with regard to input. The active input window is the window in which the mouse device is currently located. Any input which occurs is delivered to the appropriate process if that process is awaiting input. Otherwise, the input is ignored.

In order to handle input properly, a facility for polling the terminal input must be available so that the screen manager isn't stuck waiting for input when it could be processing other window requests. This facility is referred to as non-blocking input/output which means that a process does not get blocked if input is not available or if output is not currently possible. The screen manager must not be blocked when looking for input either from the terminal or from any socket for window routine requests. UNIX supplies this mechanism via the select system call. This call is essential to the ADVISE screen manager design.

The main algorithm of the ADVISE screen manager is briefly described below.

- (1) Poll (select) socket for new connections to processes calling the *coreinit* routine.
- (2) Poll (select) connections on socket for a request and process if present.
- (3) Poll (select) standard input.
- (4) If input is pending find out if the active window is expecting input. [If it is, echo the input (if indicated by request) and send the input over the appropriate connection. Otherwise ignore the input.]

*UNIX is a registered trademark of Bell Laboratories.

**Executing the appropriate output action, however, is much easier said than done. Problems of overlapping windows and redrawing damaged regions are non-trivial but are not dealt with here. These problems are solved by the screen packages used by the ADVISE screen manager and not by the ADVISE screen manager directly.

- (5) Go to (1).

3.3.2.2. The Window Filter Process

The window filter process allows processes which run on standard terminals to run within a window in the ADVISE interface. The window filter process currently exists only on the Sun version of the interface as part of the SunWindow screen manager package.

In the VAX version, the filter is designed as follows. The filter uses a socket to communicate with the screen manager and pipes to communicate with the process. Pipes are represented as dashed arrows in Figure 5. The window filter performs the following functions.

- (1) Requests that a window be opened on behalf of the process.
- (2) Receives output from the process via an output pipe and translates the output into window routine requests which are sent to the screen manager over the socket.
- (3) Receives input to the process via socket from the screen manager and passes that input to the process over the input pipe.
- (4) Requests that the window be closed when the process terminates.

3.3.3. The SunWindow Interface

As the ADVISE effort progressed, the workstation environment proved increasingly productive, until it clearly became the environment of choice. Development of the user interface focused largely on utilizing the existing window management features of the workstation to satisfy the interface goals. A Pascal interface to the SunWindow package was created to facilitate the use of existing window management programs, obviating the need for sockets and pipes. The SunWindow package was sufficiently flexible to be adapted to meet the screen management goals.

Two applications of the SunWindow interface have been developed. The first was a screen-oriented network editor which is described in the next chapter. The second is a top-level window which provides the necessary tools and "hooks" for existing ADVISE programs to run within a unified window format. The following section describes the top-level interface to ADVISE.

3.4. The Current User Interface

This section contains six screens that are representative of the initial entry into the ADVISE interface. The first screen is the top-level screen. It offers a variety of options via static menus.

Subsequent screens show entry into a consultation system (PLANT/ds) and into an interactive editor (the Network Editor). A knowledge base (backup file) is passed to knowledge base editors as an argument if a knowledge base has already been selected at the top level. Once an editing session begins, the options are sufficiently different as to make impractical a common editing language for rules, networks and tables.

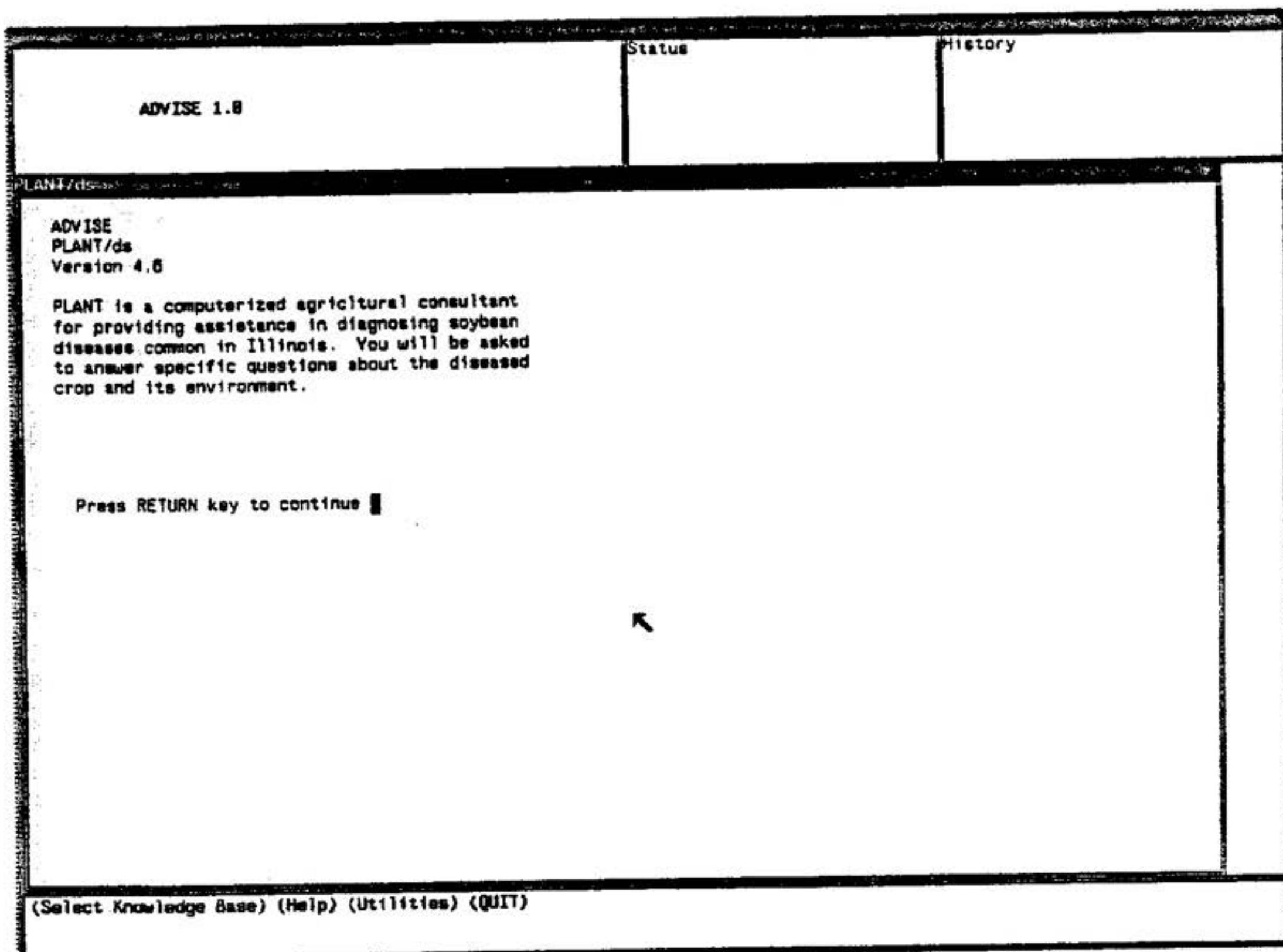
These screens represent an approximation of the final interface. Continued use of the interface will warrant regular refinements and modifications.

The first screen is the top-level screen for ADVISE. At the top of the screen are two windows — status and history. The purpose of the status window is to capture diagnostic output from other ADVISE modules (that may or may not run in the background) and allows the user to examine that output at will. The history window was designed so that the user can see where he has been in the system. The history window might indicate, for example, what windows if any are currently overlapped. At the bottom of the screen is a global menu that may change from time from time, depending on the interaction with the user. Options here will generally be extraneous, i.e. not directly related to knowledge engineering tasks. The main window on the left allows the user to select a way of looking at a knowledge base. The right window offers control scheme strategies, as well as manifestations of these control schemes — actual consultation systems.

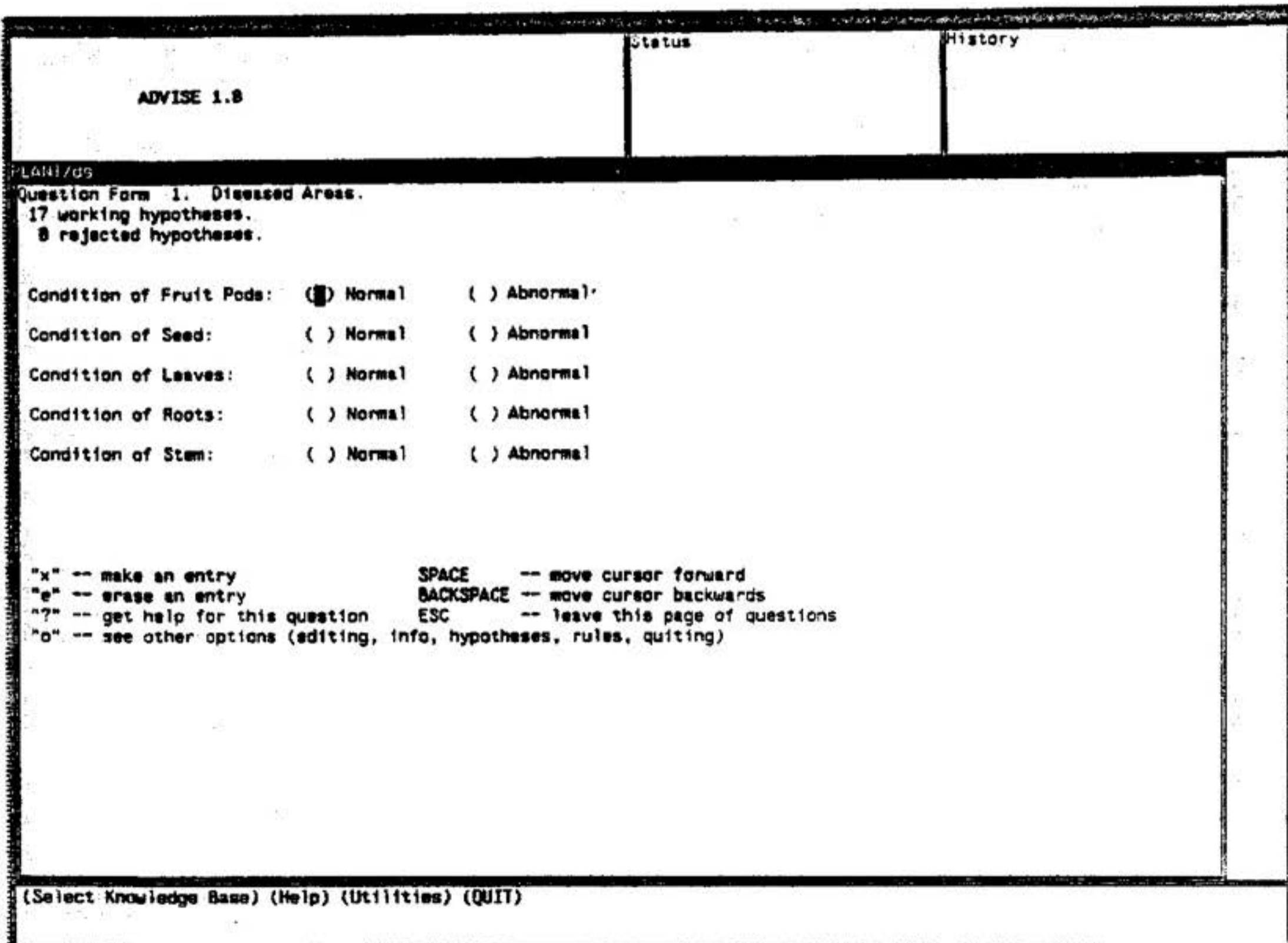
Here the user is selecting the PLANT/ds consultation system.

		Status	History
ADVISE 1.8			
Knowledge Acquisition and Maintenance		Consultation Systems	
(Rules) (Networks) (Examples)			(Backward Chaining Theorem Proving) (Information Theoretical Utility Measure)
			<input checked="" type="checkbox"/> (PLANT/ds) <input type="checkbox"/> (PLANT/cd) <input type="checkbox"/> (TURF) <input type="checkbox"/> (BABY)
(Select Knowledge Base) (Help) (Utilities) (QUIT)			

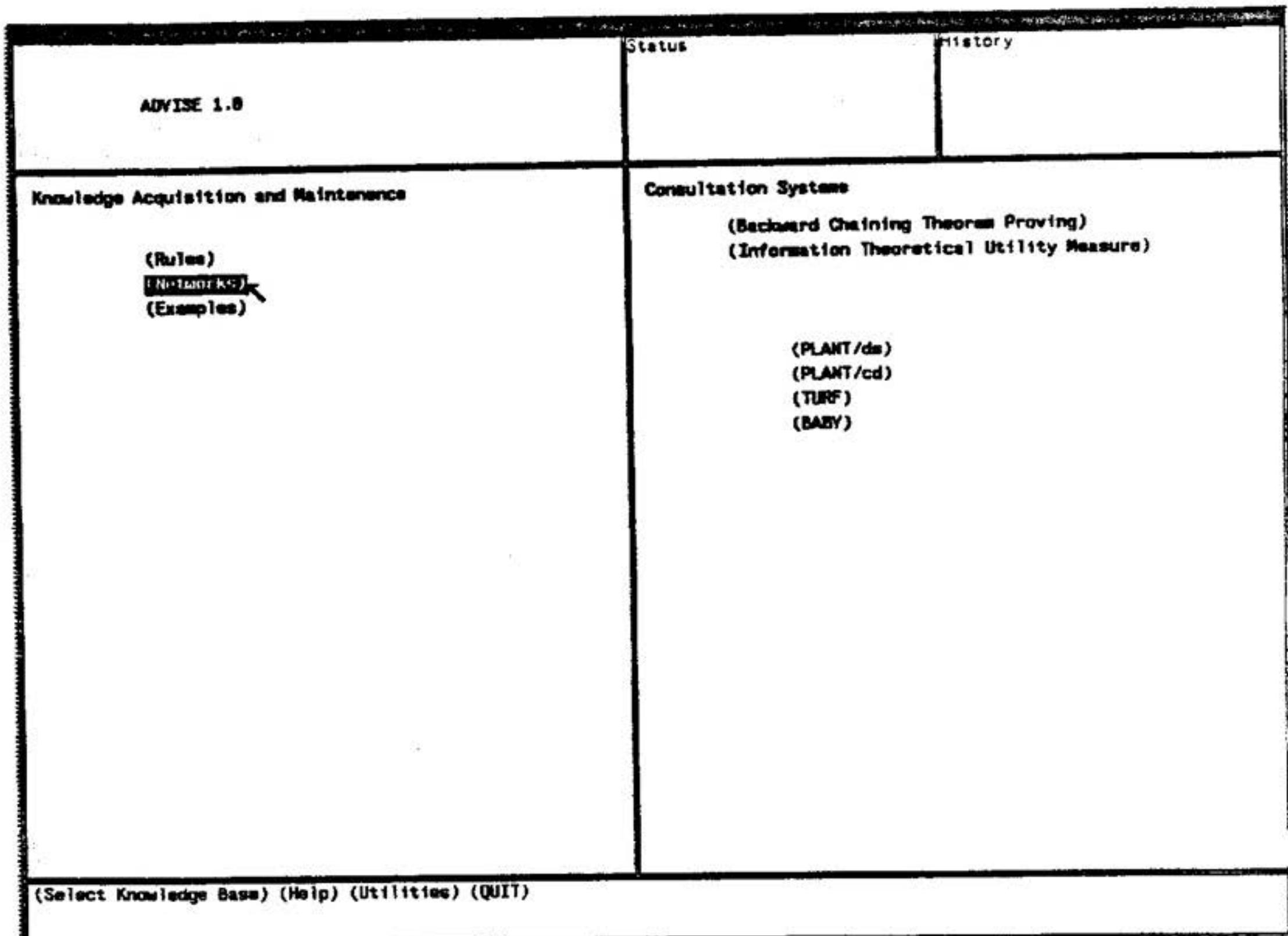
The second screen shows the initial entry into PLANT/ds. Note that the plant window obscures all the other options on the top-level window (except for the global menu at the bottom). PLANT is not open to modification by the naive user, hence he is not allowed to play with control schemes or knowledge bases at the time he runs PLANT.



The PLANT window allows PLANT/ds to run in a window as if it were on a terminal. This allows programs to run in the window system without any modification whatsoever. Of course, programs being developed now make explicit use of window management routines, such as the network editor. (See later screens.)

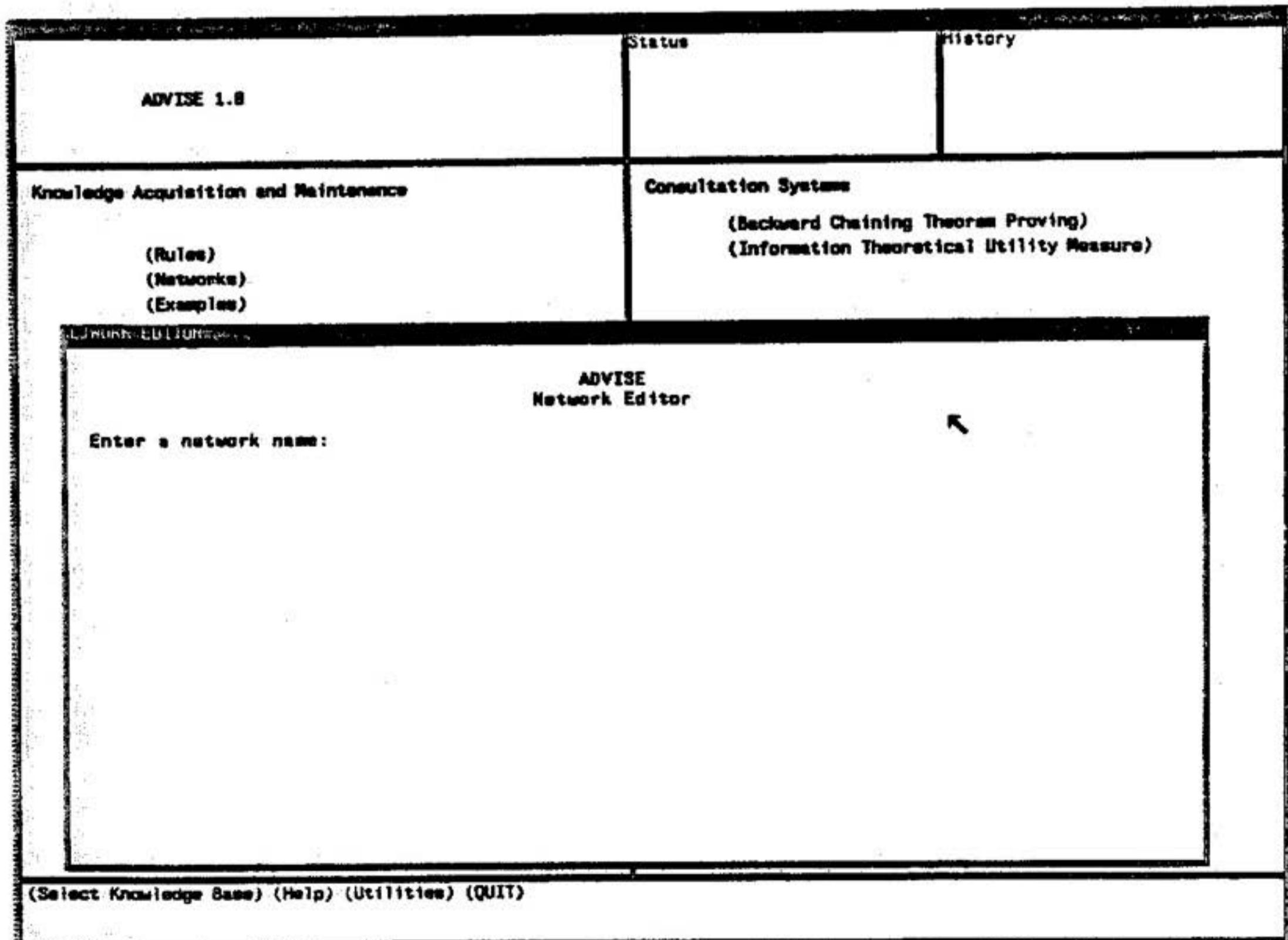


After exiting PLANT/ds, the user returns to the top-level window. Here the option of looking at a knowledge base as a network is being selected.



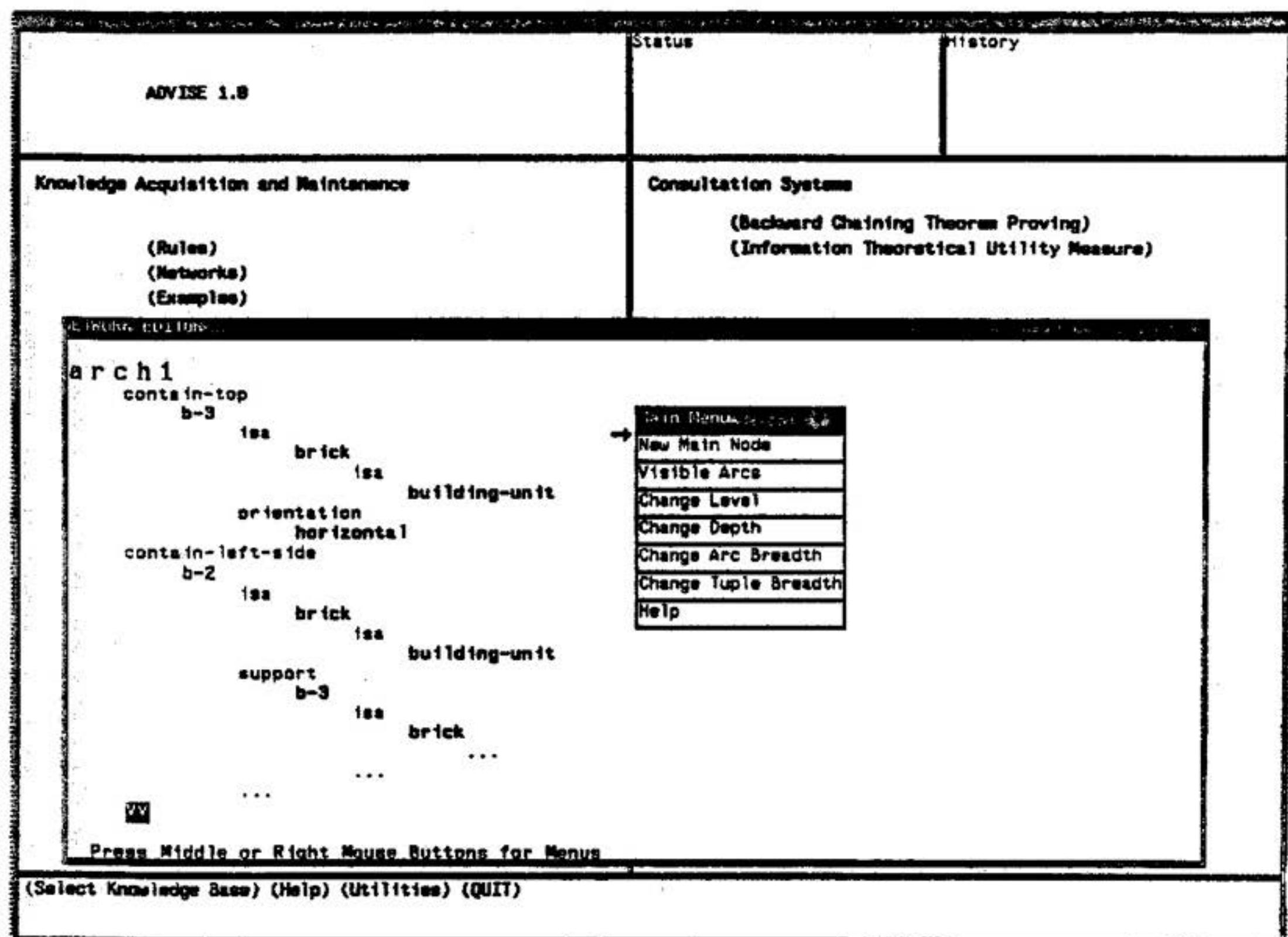
The screen below shows the initial entry into the network editor. If there were other facilities relating to networks the user would instead get a menu of available options from which to select. Since no knowledge base was selected prior to selecting networks, the network editor prompts the user for one. If a knowledge base had been selected, the network editor would have tried to open that knowledge base.

Unlike the PLANT window, the editor does not obscure all the options on the top-level window. Conceivably, a user may want to view the same knowledge base as rules, networks, and tables. In addition he may want to transfer data between the two. For example, a user could create a rule in the rule editor and add it as a node in a network by using the mouse to "pick up" the rule from the rule window and "carry" it to the network editor window.*



*Currently this transfer of data is not implemented.

This screen shows the display of the selected network along with the main menu of options. At this level, options are peculiar to the individual programs and the menus are the individual program's responsibility. The top-level menus provide all the options by which these lower levels may be entered, and are meant to be as flexible as possible without imposing any arbitrary language for starting up a variety of knowledge engineering tools.



CHAPTER 4

The Network Editor

4.1. Introduction

The Network Editor provides the facility for directly manipulating and editing ADVISE knowledge bases. Previously, when a knowledge base needed to be changed, either a text file (which is frequently not an accurate representation of an ADVISE knowledge base) had to be edited or the program that created the knowledge base must have code to manipulate the knowledge base directly. For example, the only way to alter the PLANT knowledge base was either to text-edit backup knowledge base files or run a rule parser on new input rules. QUIN, a program for editing knowledge bases represented as relational tables [Spackman, 1982], is available but has minimal benefit for the rules and network representations of currently implemented systems.

This chapter describes the Network Editor, which provides ADVISE with the capability for interactive manipulation of knowledge bases. A brief description of the ADVISE knowledge representation is provided below followed by the features of the editor and a presentation of a sample interaction.

4.2. The ADVISE Network Representation

The basic structure for representing knowledge in ADVISE is a *tuple*. A tuple is similar to a list in LISP. *Nodes* are like LISP atoms (nodes *cannot* be tuples), and a tuple is just a list of nodes. The second node of the the tuple usually has special meaning as a relation or arc between the head node and subsequent nodes in a tuple. A typical tuple looks like the one below.

```
(headnode arc subnode1 subnode2 subnode3 ...)
```

Of course, the same head node can have many arcs (relations) under it. These can be represented simply by additional tuples as follows.

```
(headnode arc1 subnode11 subnode12 ...)
(headnode arc2 subnode21)
(headnode arc3 subnode31 subnode32 subnode33 ...)
```

For efficiency reasons the above tuples would be represented internally as follows.

```
(headnode (
    (arc1 subnode11 subnode12 ...)
    (arc2 subnode21)
    (arc3 subnode31 subnode32 subnode33 ...))
))
```

In the actual implementation of this representation, nodes are memory addresses. Nodes have printnames associated with them as well as being associated with the tuples in which they appear as head node. The ADVISE tuple manager (see Chapter 9) handles all the manipulations of the knowledge base on the tuple level. The network editor simply makes the appropriate calls to the tuple manager based on its interaction with a user.

The tuple representation represents an important generalization over the basic concept of semantic networks (as described, for example, in [Winston, 1984]). Thinking about tuples in light of these networks, each tuple with the same head node can be considered a slot; each slot has a name (arc) and a value. Slot/value combinations are also known in ADVISE as *attributes*. The generalization over other representations is that slots or attributes can have many values associated with them. Thus, similar slots can be combined into a single slot

(house ((has-room living-room dining-room bedroom kitchen)))

or a single slot may have several values associated with it, for example, both a qualitative and quantitative value.

(block-1 ((orientation vertical 89.5)))

The ADVISE representation of knowledge via tuples is a general mechanism for representing not only networks but rules and relational tables as well. These representations, however, are beyond the scope of this chapter. Please refer to Chapters 5 (QUIN) and 6 (The Rule Parser) for information on these representations.

4.3. Features of the Network Editor

The network editor is a menu-driven interactive program with modest use of graphics, which runs on a Sun-2 Workstation. The interface is written on top of the SunWindow package developed by Sun Microsystems, Inc. (See Chapter 4). Naturally, being menu-driven, it is easy to figure out how it works just by pressing the mouse buttons a few times. Hands-on experimentation is the best way to learn to use the editor. For a detailed user's guide see [Channic, 1985].

The three subsections below describe the display of networks and the options (local and global) available for editing networks.

4.3.1. Screen Representation of Networks

In displaying a network, a non-graphic approach was taken to allow minimum modification for running the program on machines without graphic capabilities. Nevertheless, the network structure is readily apparent as the screens in Section 4.4 demonstrate. Nodes are represented in boldface. Arcs under nodes are not in boldface, and are set one line below and indented from the main node. Subnodes under arcs are placed similarly under the arc. Additional subnodes are placed on the same line immediately following the preceding subnode.

The only other thing a user need know in order to use the network editor is how it crops the network to fit on a display. There are three parameters which affect the display — namely, depth, arc breadth, and tuple breadth. Depth is the number of arcs down from the main node to display. Arc breadth is the number of arcs to traverse from each node. And tuple breadth is the number of subnodes to display under the head node. How these parameters affect the display will be seen in the interaction section.

If in spite of these parameters the network still cannot fit on the screen, the network editor leaves markers that indicate information has been cropped from display. At the top level of the network, these markers are arrows that point in the direction of the missing information. Menus are available at these markers to scroll the top level of the network in order to see the missing information. Beyond the top level of the display, missing information is indicated by a string of dots — "...". Missing information at this level can usually be viewed only by descending the network to make this level the new top level and, if necessary, scrolling or changing the appropriate parameters.

4.3.2. Local Editing Options

At each node or arc in the network, two sets of options are available. One set of options affects the node or arc itself, the other set affects the environment around the node or arc. An example of a *local* option is changing a printname of a node. An example of a *global* option is adding an attribute after a node. Local options are discussed in the remainder of this section. Section 4.3.3 describes the global options.

Options Available When Not At an Arc or Node

New Main Node	Allows you to enter the name of a node which will become the new top node in the display.
Visible Arcs	Allows you to select a subset of arcs to follow in displaying the network
Change *****	Allows you to enter a new value for ***** — depth, breadth, or tuple breadth — which will affect the display accordingly
Back Up	If present in menu, allows ascension of the network to the previous top node
Help	Prints these options

Options Available from Arcs

Change Arc	Allows a new arc to be inserted in place of the arc in the current attribute.
Change Printname	Allows the printname of the current arc to be changed EVERYWHERE it occurs in the network.
Delete Attribute	Removes this arc and all subnodes from underneath the head node; i.e., the entire tuple is removed.
Yank Arc	Places this arc into the arc buffer to be used in subsequent Put Arc operations
Yank Attribute	Places the attribute (this arc and all nodes underneath it) into the attribute buffer to be used in subsequent Put Attribute operations.
Make Invisible	Inhibits the display of this arc and all other occurrences of this arc as well as everything underneath them.
Enter Dictionary	If an arc does not appear in the dictionary, this option allows you to put it there.
Help	Displays these options

Options Available at Nodes

Change Node	Allows you to change a numeric node to a new number or to a new node.
Change Printname	Allows you to change the printname of this node EVERYWHERE it occurs in the network
Delete From Tuple	This node is removed from its current tuple position.
Yank	Allows you to yank this node into the node buffer for subsequent 'Put Node' operations
Make Focus	This node becomes the new top node in the display of the network.
Enter Dictionary	If the node is not in the dictionary, you may enter it there if this option is present. If and only if a node is in the dictionary, it can be made the main node via the 'New Main Node' option on the Main Menu
Help	Prints this message

4.3.3. Global Editing Options

Global editing options reflect options that affect the environment of the node or arc pointed to by the mouse.

Options Available When Not at an Arc or a Node

Edit/Create	Start a new session with a new network
Write (Backup)	The edited network will be written to the file given at startup
Write (Text)	A text representation of the edited network will be written to a specified file.
Quit	Graceful exit from a session, updating the network
Abort	Immediate exit from editor, no update.
Help	Displays these options

Options Available from Arcs

Add Node	Allows a node to be added immediately under the current arc in the attribute in which the current arc occurs. Nodes presently under the arc are shifted to the right.
Add Attribute	Allows an attribute (an arc followed by zero to 254 nodes) to be added under the main node below the attribute which contains the current arc.
Put Node	The contents of the node buffer will be put immediately under the current arc in the attribute in which the current arc occurs. Any nodes presently under the arc are shifted to the right.
Put Arc	The contents of the arc buffer will be put under the main node below the attribute which contains the current arc.
Put Attribute	The contents of the attribute buffer will be put under the main node below the attribute which contains the current arc.
Help	These options are displayed.

Options Available From Nodes

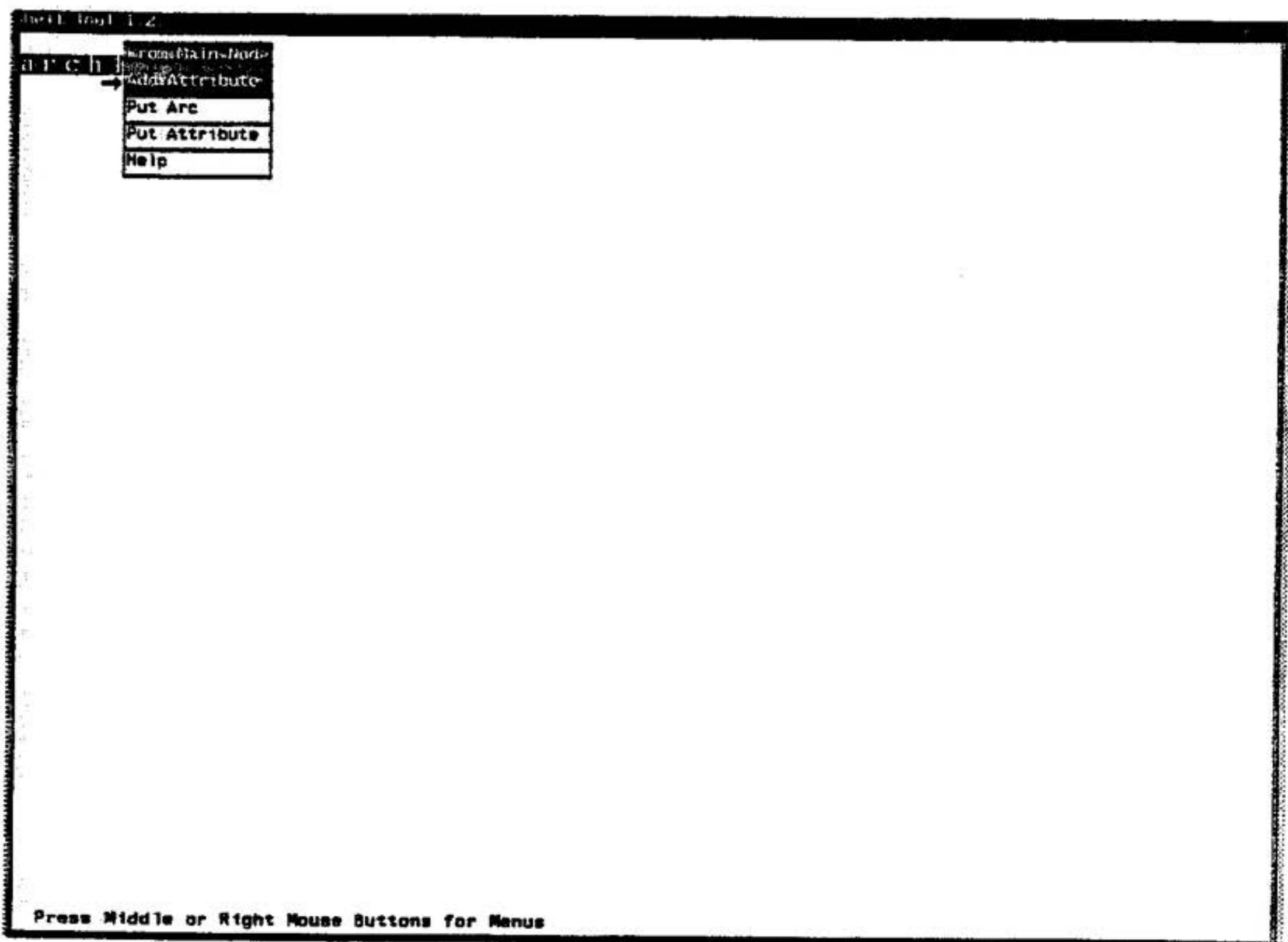
Add Attribute	Allows an attribute (an arc followed by zero to 254 nodes) to be added under the main node
Put Arc	The contents of the arc buffer will be put under the main node as the first attribute.
Put Attribute	The contents of the attribute buffer will be put under the main node as the first attribute.
Add Node	Allows a node to be added to the right of the current node in the attribute in which the current node occurs.
Put Node	The contents of the node buffer will be put to the right of the current node in the attribute in which the current node occurs.
Help	This message is displayed.

4.4. A Sample Interaction

In this section, screens are presented that show steps in the construction of a simple network. The example is chosen to illustrate the features of a network editor, and is not intended to have any semantics in the context of ADVISE. Therefore any resemblance of the network to ADVISE knowledge bases, living or dead, is purely coincidental. The network represents a arch made of building blocks.

In the first screen, the network editor has been invoked and a node, 'arch1', to be taken as the root node for the display has just been typed in. The editor must be supplied with a network name and a root node before it can begin a session. The network name can be passed as argument or typed in when the editor starts up. If "arch1" already existed in the network, the structure under this node would be displayed under the default parameters. The default parameters are set to not affect the display, i.e. the window size is the only limiting factor to the display.

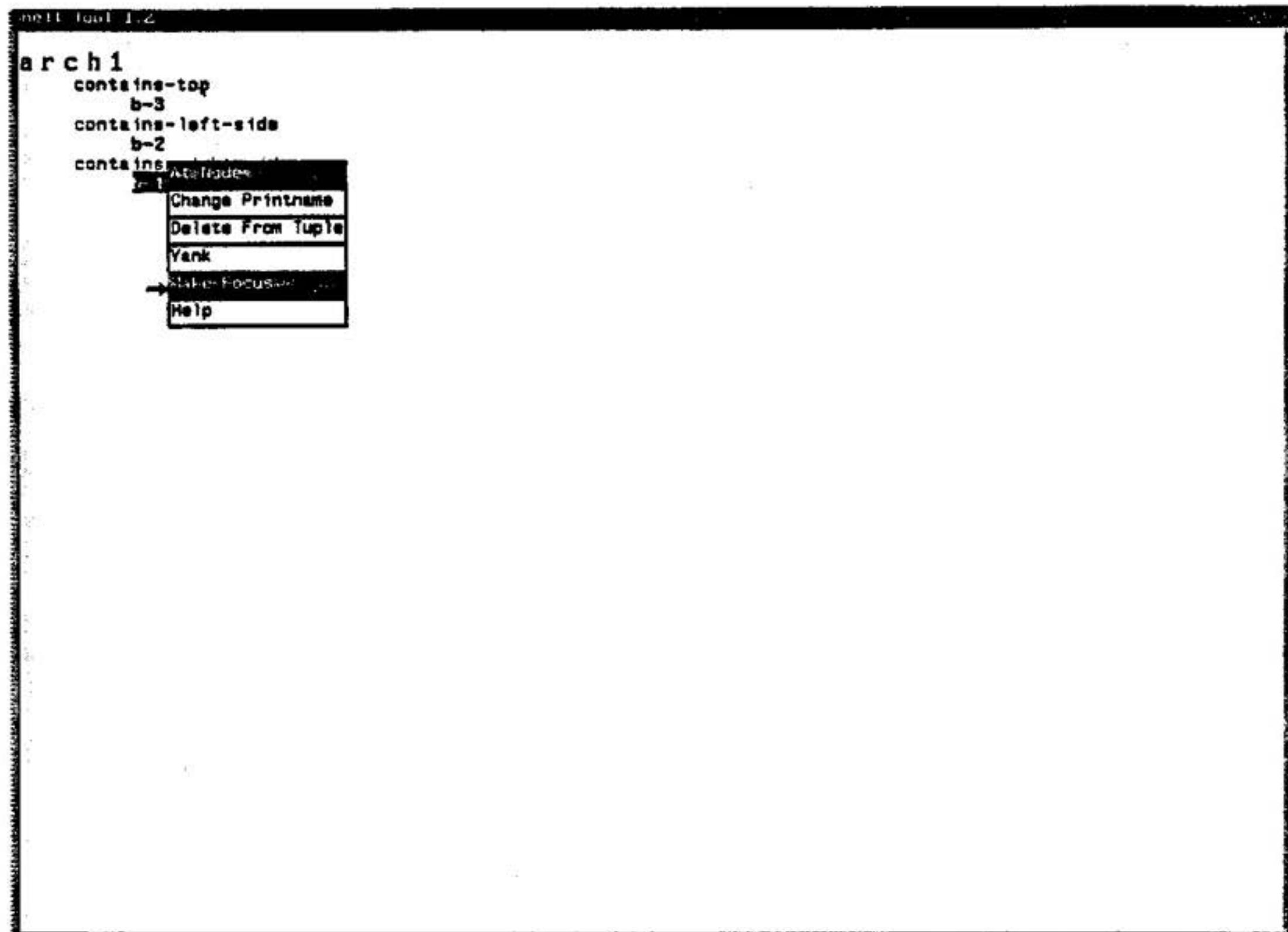
A prompt for menus appears at the bottom of the screen. The user moves the mouse to the "arch1" node, which becomes highlighted. Pressing the middle button while at the node reveals the desired option of adding an attribute (slot).



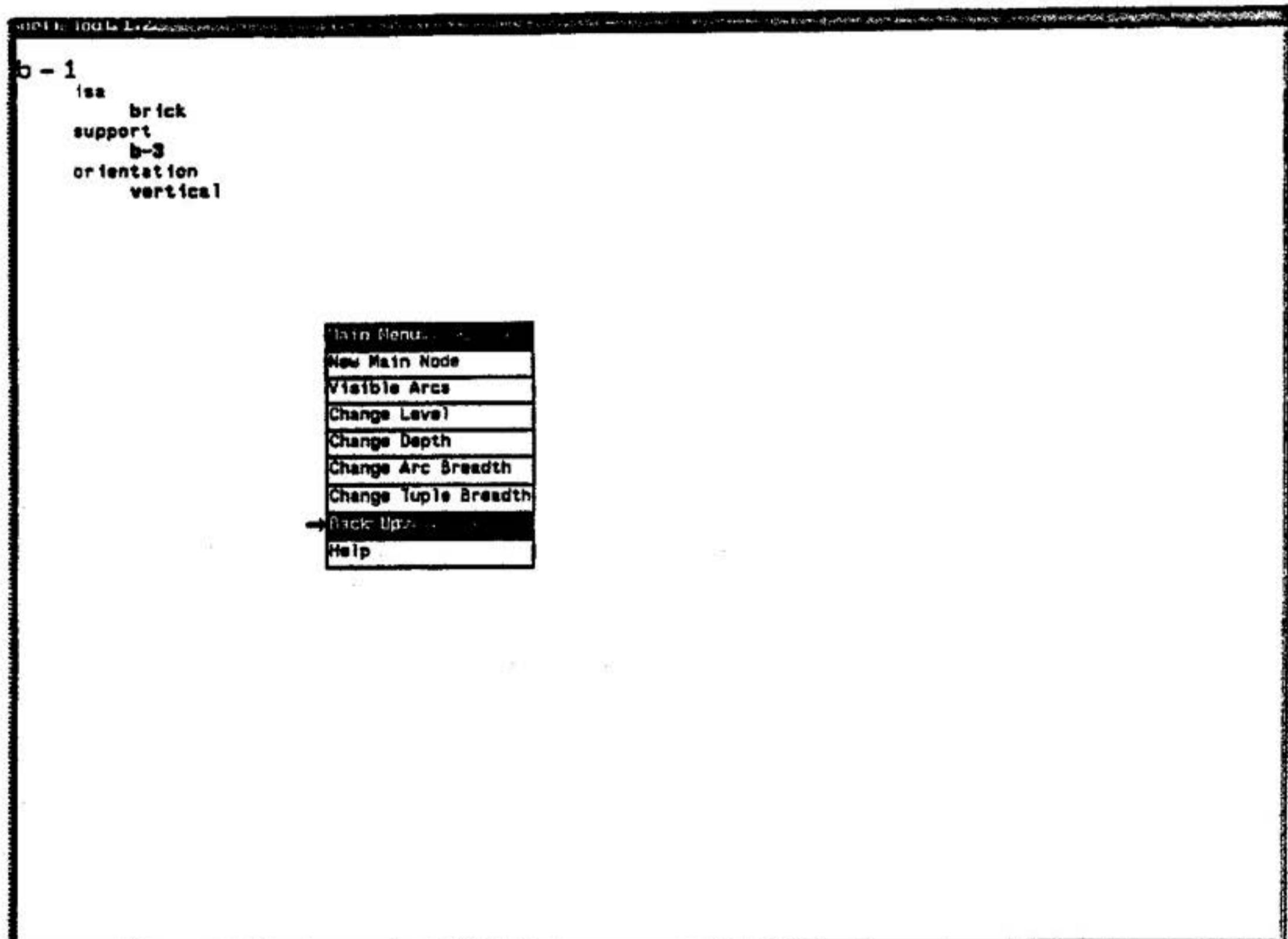
Selecting "Add Attribute" brings up a prompt for the number of nodes (printnames) including the arc in the tuple which the network editor will add into the network.

Next the user is prompted for the printname of the arc. The user types in this name. When the user presses return, the arc appears in the network and the names of the expected number of nodes are solicited. The user is prompted for each node name.

After entering all the nodes in the attribute, the attribute is displayed in the network. The user can now move to the arc to add the next attribute below the first. Continuing as with the first attribute the user has added all the arcs under "arch1". He now wishes to add arcs under the "b-1" node. To do so he must first make b-1 the new focus node.



Now "b-1" becomes the new focus node. The user has employed "Add Attribute" as before to add all the appropriate nodes and is now ready to "Back Up" to the previous root node. "Back Up" is an option from one of two menus that aren't associated with any node. The other is a global menu with options such as editing a new file, writing this file, quitting, etc.



Now the network for "arch1" is complete.

At this point the user may enter a new root node, such as "arch2", and create a network from there, or he may choose to alter the display by changing depth, for example.

```
netlbtode1.2> arch1
arch1
contains-top
  b-3
    isa
      brick
        isa
          building-unit
        orientation
          horizontal
contains-left-side
  b-2
    isa
      brick
        isa
          building-unit
      support
        b-3
          isa
            brick
              isa
                building-unit
              orientation
                horizontal
            orientation
              vertical
contains-right-side
  b-1
    isa
      brick
        isa
          building-unit
      support
        b-3
          isa
            brick
              isa
                building-unit
              orientation
                horizontal
            orientation
              vertical
```

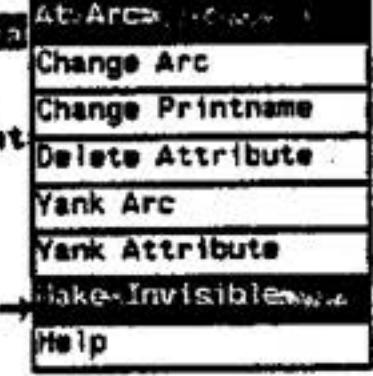
Main Menu...
 New Main Node
 Visible Arcs
 Change Level
 Change Depth...
 Change Arc Breadth
 Change Tuple Breadth
 Help

The user has indicated he wishes the depth changed from an "unlimited" default to 2. Perhaps he is interested in the first two levels of the network and prefers not to see any other information at all. The network display is clipped accordingly.

```
arch1
contains-top
  b-3
    isa
      brick
    orientation
      horizontal
contains-left-side
  b-2
    isa
      brick
    support
      b-3
    orientation
      vertical
contains-right-side
  b-1
    isa
      brick
    support
      b-3
    orientation
      vertical
```

Instead of limiting the depth, the user may have chose to limit the breadth of the network instead. Another way to alter the display is by making arcs invisible. With the breadth and depth reset to their default values, the user wants the "isa" links to disappear from the display.

```
arch1
contains-top
b-3
isa
brick
isa
building-unit
orientation
horizontal
contains-left-side
b-2
isa
brick
isa
building-unit
support
b-3
isa
brick
isa
building-unit
orientation
horizontal
orientation
vertical
contains-right-side
b-1
isa
brick
isa
building-unit
support
b-3
isa
brick
isa
building-unit
orientation
horizontal
orientation
vertical
```



This action, as did the change depth option, caused significant clipping to the display. Still another way to limit the display would have been to specify a subset of the visible arcs as the only ones to be displayed. For example, with the "Visible" the user could select the "contains" arcs and the "isa" arcs to be the only ones visible. This would allow the user to see inheritance relationships.

Now the user decides to end the session. He presses the middle mouse menu for the global option menu. To get a textual representation of the network written to a file the user can choose "Write (Text)". After the user has typed in a file name for the text file, the system notifies the user of success (or failure, such as trying to write to a read-only file, in which case the user can re-select the option with a new file name). The textual representation for this network is shown in Figure 8. Now the user "Quits". The actual network is written to file given at the start of the program.

```

shell tool 1.2.

arch1
contains-top
  b-3
    orientation
      horizontal
contains-left-side
  b-2
    support
      b-3
        orientation
          horizontal
        orientation
          vertical
contains-right-side
  b-1
    support
      b-3
        orientation
          horizontal
        orientation
          vertical

Global Menu
Edit/Create
Write (Backup)
Write (Text)
Quit
Abort
Help

Enter name of text file: arches.text
arches.text : Successfully Written

```

```
(arch1 (
  (contain-top b-3 )
  (contain-left-side b-2 )
  (contain-right-side b-1 )))  
(contain-top ( ))  
(b-3 (
  (isa brick )
  (orientation horizontal )))  
(contain-left-side ( ))  
(b-2 (
  (isa brick )
  (support b-3 )
  (orientation vertical )))  
(contain-right-side ( ))  
(b-1 (
  (isa brick )
  (support b-3 )
  (orientation vertical )))  
(isa ( ))  
(brick (
  (isa building-unit )))  
(support ( ))  
(orientation ( ))  
(vertical ( ))  
(horizontal ( ))  
(building-unit ( ))
```

Figure 6: The textual representation of tuples for "arches".

CHAPTER 5

The QUIN Subsystem

5.1. Introduction

This chapter describes the QUIN (QUery and INference) subsystem of ADVISE.1. QUIN can be viewed as a tool for database management and conceptual analysis. It represents a marriage between relational database and inductive learning technologies. Its purpose is the management of large amounts of data for input to and output from several programs that use induction to generate knowledge from examples or specific facts. It has potential applicability in the logic-based analysis of data and in the creation of knowledge bases for expert systems. This chapter is based on the Master's Thesis by [Spackman, 1982].

QUIN may be used for the management and analysis of data. Management here refers to the creation, retrieval, and modification of the data, while analysis refers to activities that attempt to discover more about 1) interrelationships within data and 2) phenomena that produce those interrelationships. These operations, shown in Figure 7, can be either data management operations or inference operations.

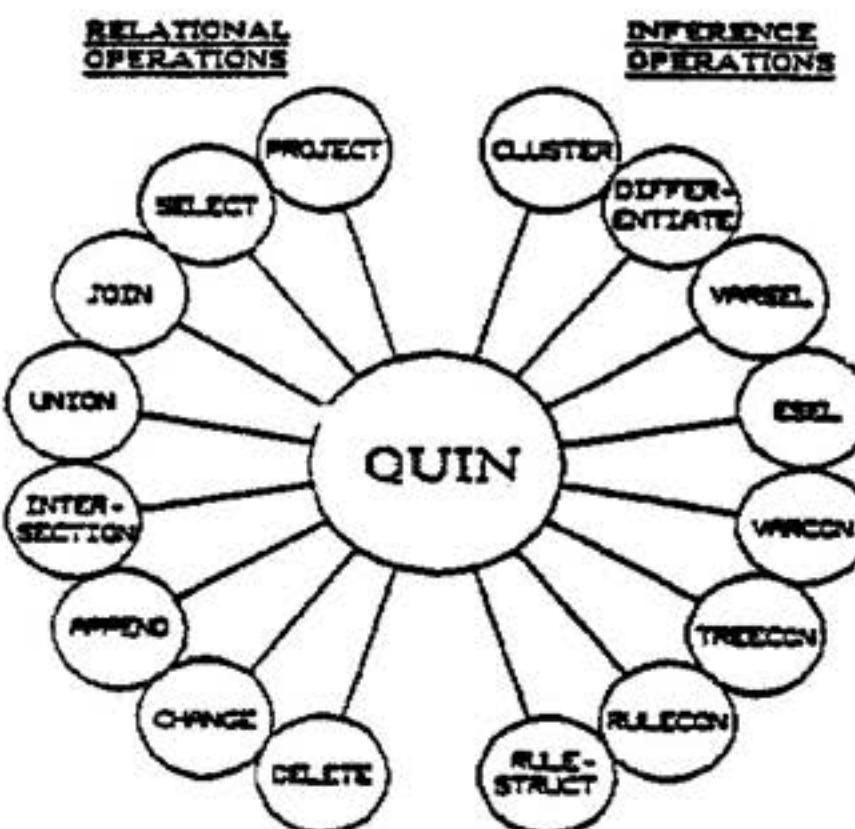


Figure 7 : Operations available in QUIN.

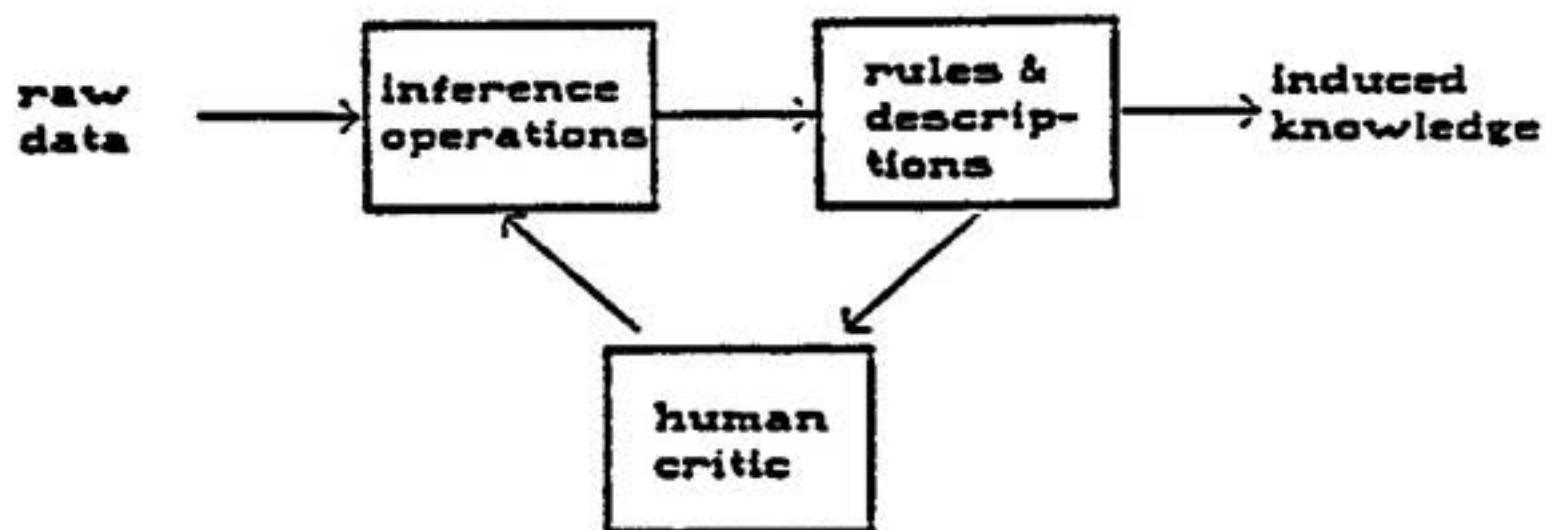


Figure 8 : The knowledge refinement cycle.

The induction programs with which QUIN interacts form a set of inference utilities that can be useful in sequence or in cycles. The databases used to test and experiment with these algorithms are more easily handled with database management techniques that store, modify, and restructure data for input to the inference programs. The cycle of knowledge refinement by iteration of the mechanized inference with a human critic is illustrated in Figure 8.

The major sections of this chapter are as follows: section 5.2 discusses the relational model of database organization, 5.3, 5.4, and 5.5 discuss creation, retrieval and modification (respectively) using the QUIN data language, 5.6 describes the inference utilities available in QUIN, section 5.7 presents a QUIN command macro language, and section 5.8 gives a brief program description and notable implementation details.

5.2. The Relational Model

This section gives a brief overview of the relational model of database organization and describes the interpretation of the model by QUIN. The concept of a table of data and the way it represents the mathematical notion of a *relation* is fundamental to the relational model of data used by QUIN. The model also includes the concepts of *keys*, *normalization*, and *relational operations*, each of which will be discussed in turn.

5.2.1. Relational Tables

A relational table is simply a table that represents a relation. Tables are familiar as a format for representing data. Consider Figure 9, an example of a table of clinical laboratory values obtained from blood specimens.

labvals			
spec#	Hgb	MCV	RBC_morph
1024	10.3	78	microcytosis
891	13.1	90	normal
555	14.2	88	poikilocytosis
423	16.5	85	anisocytosis
425	11.1	78	microcytosis
455	10.4	77	microcytosis

Figure 9 : Clinical Laboratory Values

Each column in Figure 9 corresponds to an attribute and each row represents an individual data object. The values within each row of the table represent the description of an object with respect to each of the attributes. Thus *spec#* refers to the specimen number, while *Hgb*, *MCV* and *RBC_morph* refer to the hemoglobin, mean corpuscular volume and red cell morphology of the specimen. These four names comprise the *attribute list* of the table. The values obtained from each specimen occupy a single row in the table.

A *relation* is a set of ordered rows each of length n , (called n -*tuples*), where the value of the i^{th} column in a tuple (V_i) is drawn from a domain D_i . The relation has *domain sets* D_1, D_2, \dots, D_n , where n is the *degree* of the relation. The table in Figure 9 is of degree four. Its domain sets include the set of all possible specimen numbers, the set of all possible Hgb values, the set of all possible MCV values and the set of all RBC morphologies. These domain sets need not be explicitly delineated in a database, but are important in the mathematical definition of the concept of a relation. For further reading see [Date, 1977].

Relations are intuitively well represented as tables, but relational tables in QUIN differ in some ways from the strict interpretation of a mathematical relation. First, the attributes (columns) are named, and therefore two tables in which the only difference is altered column order are considered to be equivalent. Second, in relations the rows are not considered to be ordered, but QUIN allows rows to be ordered according to the values of attributes, e.g. in increasing order by index number (value-controlled ordering). Third, the “zeroth” row of a table in QUIN is occupied by the attribute list, and data then follows beginning with the next row.

5.2.2. Keys

A *key* is an attribute or combination of attributes that have unique values for each tuple in the relation. In other words, no two tuples in a relation may have identical values of the key attributes. This constraint ensures against duplication of data records. Some examples of keys include an identification number (such as specimen number in Figure 11), a unique name, or a unique combination of two more attributes, such as name and date. To allow purposeful duplication of data for use in the inference programs, a table may optionally have no key defined.

5.2.3. Normalization

A table is said to be *normalized* (in first normal form) if each entry in the table is non-decomposable, i.e., each entry is neither a set of values nor a table, but a single value. Several levels of normalization have been defined (1st, 2nd, 3rd, Boyce/Codd, 4th. Projection/Join - see [Date, 1977]) but the attainment and management of normalization beyond first normal form in QUIN is left entirely to the discretion and effort of the user.

5.2.4. Relational Operations

The relational model includes operations that take relations as input operands and give a relation as output. These operations can be classified as traditional set operations (union, intersection, difference and Cartesian product) and special relational operations (project, select and join). These relational operations are incorporated within the query language provided in QUIN. They are briefly introduced here and examples of their implementation are given in subsequent sections.

- | | |
|--------------|---|
| Union | The union of two relations is the set of all tuples contained in both relations (without duplication). To perform the union of two relational tables in QUIN, they must have <i>identical attribute lists</i> . The same constraint applies to the operations of intersection and difference. |
| Intersection | Intersection comprises the set of tuples common to both relations. |
| Difference | The difference of two relations is the set of tuples contained in the first relation but not in the second. |
| Product | The Cartesian product of two relations is the set made up of the concatenation of each of the tuples in the first relation with each of the tuples in the second. |
| Selection | Selection provides a subset of tuples from a relation that meet certain selecting criteria. It produces a row-wise or horizontal subset of the relation. For example, a selection requiring the specimen number to be less than 500 from Figure 9 would give the result found in Figure 10. |
| Projection | Projection, on the other hand, provides a column-wise or vertical subset of the relation. Redundant tuples are eliminated from the resultant relation. For example a projection of the RBC_morph column would yield Figure 11. |

labvals			
spec#	Hgb	MCV	RBC_morph
423	16.5	85	anisocytosis
425	11.1	78	microcytosis
455	10.4	77	microcytosis

Figure 10 : Results of Selection Operation

labvals
RBC_morph
microcytosis
normal
poikilocytosis
anisocytosis

Figure 11 : Results of Projection Operation

Join

Join is slightly more complicated than selection and projection. It produces a combination of two (or more) tables based on all attributes they have in common. There are three different kinds of join, the most common being the *natural join*. The resultant table of a natural join will have a tuple for each pair of tuples in the original tables that share identical attribute-values for every attribute the tables share. If the original tables have no attributes in common, the resultant table is the Cartesian product of the two tables. If no pairs of tuples have identical attribute-values (assuming a common attribute) then the join results in a null table. For examples, see section 5.4.1. and Figures 15, 16 and 17.

The operations described above can be incorporated into a powerful retrieval language called a relational calculus. The VL data language is the relational calculus used by QUIN. VL commands provide the capabilities for relational table creation, retrieval and modification. The language is easily learned and requires a minimum of procedural specification so that it is reasonable to expect that users with minimal computer background could quickly learn and use it. QUIN has on-line help available to describe the use of each command. Help can be obtained by typing **help** or by simply typing a question mark at a prompt. If specific information is desired about a particular command, the command name should be entered following the word "help," followed by a carriage return.

The following three sections describe the fundamental constructs of the language QUIN uses as such a calculus and retrieval language — namely, table creation, table retrieval, and table modification.

5.3. Table Creation

The instructions for creation of tables are **define** and **add**. **Define** creates an empty table and sets up the specifications for the attribute list and the key, while **add** puts new tuples into a table.

5.3.1. Define

This instruction specifies a new table, its name, the names of the attributes, and (optionally) the name(s) of the key(s). The names of tables and attributes must begin with a letter and can contain any combination of letters, numerals, and the characters "#" and ". ". No two tables may have the same name, nor can a table-name be the same as any

attribute-name or reserved word. A table may not have two identical attribute-names, but two different tables may (and often do) share a common attribute-name. Keys are optional, but, if declared, should be the first (i.e., leftmost) attribute(s) in the table.

Consider as an example the definition of a database that keeps track of results of blood tests on patients, as shown earlier in the "labvals" table of Figure 9. "labvals" stores the information on specimens and the values measured. The unique attribute (key) of each record is the specimen number. To create this table using the define instruction, the user would enter the command:

```
define labvals (spec#, Hgb, MCV, RBC_morph) key := spec#
```

We could also define a table called "spec" to keep track of the dates of individual blood specimens:

```
define spec (spec#, ID#, day, month, year) key := spec#
```

Another table called "ptrc" would store a patient's identification number, his name, and his admitting diagnosis:

```
define ptrc (ID#, name, dx) key := ID#
```

5.3.2. Define Event

An event is a table with only one row. Its purpose is to specify a complete single data object with attributes that may be found in several different tables, so that adding the event to each of those tables is easier and less subject to error. Events are defined by the define event instruction followed by the event name and then a parenthesized list of attribute-value pairs. Continuing the previous example, we could define an event that contains all the attributes of the three tables (ptrc, spec and labvals):

```
define event E1
  (ID# := 988,
   name := Jones,
   dx := iron_deficiency,
   spec# := 1024,
   day := 25,
   month := 6,
   year := 1982,
   Hgb := 10.3,
   MCV := 78,
   RBC_morph := microcytosis)
```

Event "E1" records that a blood test was done on patient number 988 whose name is Jones and whose diagnosis was iron_deficiency. The blood, specimen number 1024, was drawn on 25 June 1982 and the results showed a hemoglobin of 10.3, a mean corpuscular volume of 78, and red cell morphology was microcytosis. The attribute-value pairs in the event definition can be arranged in any order.

5.3.3. Add

The add instruction places tuples (rows) into a table. There are four forms of the instruction: one for single row addition, another for multiple rows, one for adding an event to a table, and one for adding an external file of tuples to a table.

- (1) A single row may be added as follows:

```
add (365, Smith, aplastic_anemia) to ptrc
```

- (2) Multiple rows are added in similar fashion:

```
add to ptrc
(398, Clark, folate_deficiency) (404, Blake, iron_deficiency)
(425, Smith, hemolytic_anemia ) (241, Jones, iron_deficiency)
end
```

- (3) Adding an event to several tables is simple:

```
add E1 to ptrc
add E1 to spec
add E1 to labvals
```

- (4) Adding an external file named "vls" to the "labvals" table would be done as follows:

```
add vls to labvals
```

The external file must be set up in tabular form as illustrated by the "vls" file in Figure 12.

Addition of tuples may be done at the beginning of the table, the end, or before or after any specified row in the table by using a *row condition*. All four forms of add may be used with a *row condition*. If no *row condition* is specified then the addition is done after the last row of the table. For example, the following places a new tuple at the first row:

```
add (425,404,26,6,1982) to specs : [row < 1]
```

891	13.1	90	Normal
555	14.2	88	Poikilocytosis
423	15.5	85	Anisocytosis

Figure 12. Format of File "vls"

The colon is to be read "such that" and the row condition is of the same form as retrieval conditions (see section 5.4.2). The reserved word *last* may be used to insert before the last row:

```
add datafile to specs : [row<last]
```

The condition "[row>last]" would be redundant because that is the default. If there are not as many rows in the table as specified in the row condition, the new tuples will be added at the end of the table.

5.4. Table Retrieval

The retrieval commands are *get* and *let*. Simple retrieval of an entire table requires only listing the table name after the keyword *get*. Selected portions of the table can be retrieved and displayed also (see sections 4.4.1. and 4.4.2.). A new table can be created with the keyword *let* followed by the name of the new table, ":" , and the description of the new table. For example, the following command creates a new table called "tests" which is the same as the "labvals" table:

```
let tests := labvals
```

The new table would be created but not displayed. The command to display the table is:

```
get tests
```

The "tests" table as shown in Figure 13 would then be displayed. It would be identical to the "labvals" table except for its name.

The combined effect of the *get* and *let* could have been accomplished by simply saying:

```
get tests := labvals
```

tests			
spec#	Hgb	MCV	RBC_morph
1024	10.3	78	microcytosis
891	13.1	90	normal
555	14.2	88	poikilocytosis
423	16.5	85	anisocytosis
425	11.1	78	microcytosis
455	10.4	77	microcytosis

Figure 13 : Result of Retrieval

Symbol	Meaning
*	Join
v	Union
&	Intersection
-	Difference
+	Append

Figure 14 : Relational Table Expression Operators

The *let* command is used to create temporary tables that can be used as working copies or can be saved as new permanent tables. It never displays the results of its work. The *get* command always displays its results, and can also be used to create temporary tables. A wide variety of more complicated retrieval instructions can be specified by appending the appropriate modifying expressions to these two commands. These additional expressions are specified by a *relational table expression* (see section 5.4.1.) optionally followed by a *VL condition* (see section 5.4.2.).

5.4.1. Relational Table Expressions

The table expression specifies the table or tables and the attribute or attributes to be retrieved. All operations described in section 5.2.4 except selection can be specified with a table expression. When more than one table is listed in an expression, tables must be separated by an operator. The operator symbols and their meanings are given by Figure 14.

spec				
spec#	ID#	day	month	year
1024	988	25	6	1982
425	404	26	6	1982
850	405	27	6	1982
455	406	27	6	1982

Figure 15 : The "spec" Table

These operators take precedence over all others in the retrieval instruction. All tables listed together with the logical operators (union, intersection, difference) and the append operator must be *union compatible*, which means that they must have identical lists of attributes, in the same order, and of the same type (e.g. if attribute N is an integer in one table, it must also be an integer in the other table(s) in the instruction).

To illustrate the use of operators in table expressions, let us assume we have a table named "spec" as shown in Figure 15.

The following instruction would create a table called "T1" which is a join of the "spec" and "tests" (Figure 13) tables. The common descriptor for the join is the specimen number.

```
let T1 := spec * tests
```

The join of tables "spec" and "tests" would appear as in Figure 16.

If no attributes are listed in the retrieval command (as in the example in Figure 18) then the full set of attributes for all tables is retrieved. If a subset of attributes is specified then the projection of those attributes on the table is retrieved. For example:

```
get spec * tests (ID#, Hgb)
```

will retrieve a table with two columns. It will be the projection of ID# and Hgb on the join of spec and tests. Figure 17 shows the result.

For the purposes of the inference algorithms it is sometimes not desirable to eliminate redundancy when doing a projection, so QUIN provides two other methods of specifying projection. One method, using &, simply does a "column selection" and retrieves all rows even if redundant. The other method, using #, eliminates redundancy but provides an additional column that shows the number of times that a particular row occurs. These three forms of projection instruction are illustrated in Figures 18, 19, and 20.

An attribute may be replaced by a function of an attribute in the retrieval expression. Available functions include min, max, sum, count and domain. Figure 21 gives an example of the use of the min function.

T1							
spec#	ID#	day	month	year	Hgb	MCV	RBC_morph
1024	988	25	6	1982	10.3	78	microcytosis
425	404	26	6	1982	11.1	78	microcytosis
455	406	27	6	1982	10.4	77	microcytosis

Figure 16 : Join of "spec" and "tests"

ID#	Hgb
988	10.3
404	11.1
406	10.4

Figure 17: Projection & Join

get tests(RBC_morph)

tests
RBC_morph
microcytosis
normal
poikilocytosis
anisocytosis

Figure 18 : Ordinary Projection

get tests(RBC_morph,#)

tests	
	#
RBC_morph	
microcytosis	3
normal	1
poikilocytosis	1
anisocytosis	1

Figure 19 : Projection & Count

get tests(RBC_morph,&)

tests
RBC_morph
microcytosis
normal
poikilocytosis
anisocytosis
microcytosis
microcytosis

Figure 20 : Projection with Redundancy

```
get tests( min(Hgb) )
```

tests
Hgb
10.3

Figure 21 : Use of "min"

5.4.2. VL Conditions

A VL *condition* is the part of a retrieval command which specifies selection. The following example illustrates the major features of a VL condition. The condition begins with a colon which should be read "such that."

```
get tests : [Hgb = 14..16] [RBC_morph <> normal] v [Hgb < 14]
```

The command would retrieve all rows from table "tests" in which either a) the Hgb is in the range 14 to 16 and the RBC_morph is not normal, or b) the Hgb is less than 14.

A VL condition thus consists of a disjunction of one or more *complexes*. Complexes consist of a conjunction of one or more *selectors*. In the "get" command above, "[Hgb = 14 .. 16]" is a selector. Selectors may be separated by the conjunction operator &, or simply listed one after the other, as in the complex "[Hgb = 14..16] [RBC_morph <> normal]" above. Selectors or groups of selectors (complexes) may be separated by the disjunction operator v. Thus a condition is a "sum of products" of logical (VL) selectors.

VL selectors are used to specify the body of the condition. A VL selector consists of a left square bracket, an attribute name (the referee), a comparison operator (=, <>, <, >, <=, >=), a comparison value (the reference), and a right square bracket. The comparison value may be a single value (e.g. "normal"), an arithmetic range of values (e.g. "14 .. 16"), an arithmetic expression (e.g. "Hgb + 2.5"), or a list of values, ranges, or expressions separated by the "or" operator (e.g. "3..5 v 7 v Hgb/10").

5.4.3. Ordering of Rows

All retrievals may optionally have an ordering condition. The phrases "order up on" and "order down on" are appended to the retrieval instruction, along with the name of the attribute to be ordered on. For example, the instruction

```
get ptrc order up on ID#
```

will retrieve the table "ptrc" in ascending order of ID numbers.

5.5. Table Modification

The table modification instructions are change, delete, and save.

5.5.1. Change

The change instruction is used to assign new values to existing rows in a table or to change the name or type of an attribute. When the user types:

```
change tablename
```

he enters a "sub-instruction" mode in which all commands refer to the table being changed. A working copy of the original table is made for security in case of error, and the user's prompt is ">>". To leave the change mode the user types abort or exit, with only the latter exit resulting in factual modification of the original table.

There are several commands available in the change mode. The user may use ordinary assignment statements to change the values of each attribute. Some examples are given in Figures 22 and 23. The assignments may be followed by a VL condition that restricts the assignment of values to specific rows of the table. Attribute names may be changed by specifying the condition "[row=0]". The display sub-instruction displays the working table; the get sub-instruction displays the original table before any changes. A simple change instruction sequence may be entered as shown in Figure 22.

The commands in Figure 23 illustrate how the table might be modified within the change mode.

5.5.2. Delete

Delete is used to remove rows or columns from a table, or to remove a table from the database. Each of these three functions is accomplished by a different form of the instruction.

- (1) Deleting rows is accomplished by specifying a VL condition:

```
delete ptrc : [ID# = 250 .. 500]
```

This will delete all rows in table "ptrc" where the ID number is in the range 250 to 500 inclusive.

- (2) Deleting columns is accomplished by specifying a projection:

```
delete spec(day,month)
```

This will delete the day and month columns from table "spec".

- (3) Deleting an entire table or event is done by simply giving the table name:

```
delete T1
```

This will remove the table named "T1" from the database.

5.5.3. Save

Any tables created with the get or let instructions will be given temporary status; save is the instruction that changes temporary to permanent status. Tables with permanent status will stay in the database after a session is completed, whereas tables with temporary status will be deleted at the end of a session.

```
> change tests /* enter change sub-mode */
ok      /* system response */
>> get    /* look at original table */
```

tests			
spec#	Hgb	MCV	RBC_morph
1024	10.3	78	microcytosis
891	13.1	90	normal
555	14.2	88	poikilocytosis
423	16.5	85	anisocytosis
425	11.1	78	microcytosis
455	10.4	77	microcytosis

Figure 22 : Original Table to be Changed

```
>> Hgb := high :[Hgb > 16]
>> Hgb := low  :[Hgb < 14]
>> Hgb := normal :[Hgb = 14:16]
>> display /* look at changed table */
```

change\$table			
spec#	Hgb	MCV	RBC_morph
1024	low	78	microcytosis
891	normal	90	normal
555	normal	88	poikilocytosis
423	high	85	anisocytosis
425	low	78	microcytosis
455	low	77	microcytosis

```
>> end      /* exit, making the changes permanent */
change completed /* system response */
```

Figure 23 : Table with Modified Hgb Column

5.8. Inferential Operators

This section describes the inferential operations on an informal conceptual level. References are provided for more detailed explanations of the algorithms and the theories supporting them. In the current implementation, only "cluster" and "diff" are operational. However, the same methods of interaction can, in principle, be used with all the inference programs mentioned here. There are many inference commands described for QUIN. These all interface with other software through relational tables. The general format of the commands is:

command(parameter1,parameter2,...).

Each command has at least two variants. One variant allows the user to wait for results, and the other allows him to run the inference command as a *background process* (allowing him to proceed on other QUIN commands concurrently with the execution of the inference command). Other variants exist for some commands for specifying a result table name or giving non-default parameters. The current implementation of QUIN has clearly defined the arguments required for the following commands:

<u>command program format</u>		
cluster	CLUSTER	cluster(table,parmtable,results)
diff	GEM	diff(table1,table2, ... ;parmtable,results)
aql1	AQ11	aql1(table1,table2, ...)
esel	ESEL/2	esel(table,parmtable,results)
varsel	PROMISE	varsel(table1,table2,...;parmtable.results)
varcont	CONVART	convart(table1,table2,...;parmtable)

Other commands not yet implemented are:

<u>command</u>	<u>program</u>	<u>format</u>
apply	AQ11	??
treecon	OPTREE	??
varcon	NEWVAR	??
sim	CLUSTER	??
reun	CLUSTER	??
gen	CLUSTER	??

5.8.1. Fetch and Results

In addition to the basic induction commands, there are a couple of commands for managing processes which have been run in background. These are RESULTS, which reports the status of each induction job submitted since the beginning of the current QUIN session, and FETCH, which permits the results of a background job to be read back into a QUIN table (the default action of induction programs is to create files which need to be reread or *fetched*). RESULTS takes no arguments and reports something like the following.

<u>Job-Number</u>	<u>Program</u>	<u>Status</u>
1	Varsel	done
2	Esel	done
3	Gem	running

Here, the results of job #1 and job #2 could be *fetched*. The format of the FETCH command is :

fetch(job-number,result-table) ,

where job-number refers to an entry in the table reported by RESULTS and results-table is the name of a table to be created for storing the results of that job.

5.6.2. Cluster

The purpose of the "cluster" operation is to divide a collection of objects into smaller groups of similar objects based upon some criterion or measure of similarity. Clustering is the process of developing a taxonomy or classification scheme for the objects of a study.

The program invoked by the "cluster" command in QUIN is called *CLUSTER/paf* [Michalski, 1980]. The reader is urged to consult the references cited for more complete explanations of the details of the program's operation and theoretical background. Unlike most numerical taxonomic techniques, this program uses a "concept-based" method of clustering that produces descriptions of the clusters (categories) that it derives. It also permits the user to specify the criteria which are to be used to evaluate clusters. One or several criteria can be maximized simultaneously to produce the optimal clusters. Some of the criteria available to characterize clusters include:

- the fit between the clustering and the data (sparseness),
- the total inter-cluster differences (degree of intersection),
- the number of attributes which singly distinguish among all the clusters (essential dimensionality), and
- the simplicity of cluster descriptions (number of selectors).

The names and numbers of criteria currently available appear in Figure 24.

The "cluster" operation is invoked by the following instruction,

cluster (events,parameters,results)

where "events" and "parameters" represent the names of relational tables within QUIN.

criterion number	brief description
1	sparseness
2	degree of intersection
3	number of events occurring in more than one complex
4	share of events (evenness of cluster size)
5	number of selectors (simplicity of cluster descriptions)
6	essential dimensionality (dimensionality of differences)
7	relevant-variable sparseness
8	relevant-variable-set sparseness

Figure 24 : Clustering Criteria

and "results" is the name of a table which may or may not already exist (if not it will be created). The names of any existing tables may be used in the command. The events table must contain the descriptions of the objects to be clustered with each object occupying one row in the table. Each column represents an attribute of the objects in the table. The parameters table is used to indicate K (the number of clusters to be formed), the criteria to be used and other optional parameters. The optional results table is the table in the database to which the results of the clustering will be returned. If no results table is specified, the output of CLUSTER can be found in a file in the user's working directory.

A simple example of clustering follows, using data similar to that in previous examples. In addition to an events table, a parameters table and at least one criterion table must be prepared before issuing the "cluster" command. Figure 25 illustrates a parameters table and Figure 26 illustrates a criterion table.

The parameters table (Figure 25) has two tuples (rows). The CLUSTER algorithm will therefore be run twice, once for each row in the parameters table. The first time it will split the events into three groups ($k=3$) and the second time it will create only two groups ($k=2$). Criterion "crl" (found in Figure 25) is defined in the criterion table called "crl_criterion" (Figure 26). Criterion "1" is *sparseness*, (see Figure 24). It is the only criterion which will be used by the program in this example. The tolerance is a measure of the degree of error allowed in fitting the clusters to the criterion. The events table for this example is shown in Figure 27.

The values in the events table must all be integers. Attributes, such as those represented here, which ordinarily have continuous linear values must be made discrete before CLUSTER can deal with them. The meanings of the values in the events table are given in Figure 28, with the values from the events table in the "#" column followed by the range of real values which have been assigned to each value.

When the sample given in Figures 25-27 is run, CLUSTER splits the events into three groups as in Figure 29.

This is a particularly simple example, but it gives the flavor of the clustering operation. In this case, CLUSTER has discovered three groups which can be interpreted as being cases of microcytic anemia (group one), normal blood counts (group two), and macrocytic anemia (group three). For further examples of the use of CLUSTER see [Michalski, 1982a].

5.6.3. Diff

Diff (differentiate) takes a number of classes of events that have already been categorized, and attempts to find the conceptually simplest rules that will predict the category of each event, i.e., discriminate among the categories. The algorithm invoked by the command is called *Aq* and is incorporated in the program called *GEM* [Stauffer, 1982].

The following is an example of the use of the diff instruction to create rules for differentiating the groups of objects represented in three tables named grp1, grp2 and grp3. There are a number of differences from the "cluster" operation:

- (1) No criterion tables are needed.
- (2) The parameters table, an example of which is shown in Figure 30, has a different format from the "cluster" parameters table which was illustrated in Figure 25.
- (3) The values in the events tables need not be integer only. Discrete nominal values are allowed in addition to integers. An example of the way an events table might appear is given in Figure 31.

parameters	
k	criterion
3	crl
2	crl

Figure 25 : Parameters Table

crl_criterion	
criterion	tolerance
1	0.0

Figure 26 : Criterion Table

events		
mcv	hgb	mchc
0	1	0
2	2	0
1	1	1
3	4	3
3	4	3
4	1	4
5	2	4
4	1	4

Figure 27 : Events to be Clustered

```

#   mcv
0 <80
1 60 to 89
2 70 to 79
3 80 to 94      (mcv normal = 80 to 94 cu microns)
4 95 to 104
5 >104

#   hgb
0 <8 = 0
1 9 to 10
2 11 to 12
3 13 to 14
4 15 to 18      (hgb normal = 14 to 18)
5 17 to 18
6 19 to 20
7 >20

#   mchc
0 <27
1 27 to 32
2 33 to 38      (mchc normal = 33 to 38 %)
3 39 to 44
4 >43

```

Figure 28 : Meanings of Values in Events Table

Group one : events 1,2,3
 Group two : events 4,5
 Group three : events 6,7,8

Group one is described as:
 mcv<80 and hgb=9..12 and mchc<33
 Group two is:
 mcv=80..94 and hgb=15..16 and mchc=33..38
 Group three is:
 mcv>94 and hgb=9..12 and mchc=39..44.

Figure 29 : Results of Clustering

params	
echo	maxstar
pcve	10

Figure 30 : GEM Parameters Table

grp1		
day	rainfall	hours_sunlight
Monday	light	6
Saturday	none	12
Wednesday	heavy	2

Figure 31 : GEM Events Table

weekdays
names
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday

Figure 32 : Domain Values or Days of the Week

Note that there still must not be real-valued attributes (e.g. 50.2) but nominal attributes are allowed. In addition, the type and domain of an attribute can be declared. In the table in Figure 31, the "day" attribute clearly can take on seven values which are ordered and cyclic. Only three of these appear in the table and they are not in order. To define the domain of this attribute one must first prepare a relational table with one column containing all possible values which the attribute may have. The values should be listed in order, as in Figure 32.

Given such a table, the following command will set up a permanent domain for the attribute "day" which will automatically be referred to whenever the system needs to prepare "day" as an attribute for the "diff" operation:

```
domain (day) := weekdays
```

The "diff" operation recognizes three types of attributes :

- nominal (discrete unordered)
- linear (discrete ordered, such as "rainfall" in Figure 32), and
- cyclic (discrete with cyclical ordering, such as days of the week).

The system assumes that integer values are linear and that alphabetic values are nominal. When the opposite is true or when the attribute is cyclic, the following instruction can be used to define the type of the attribute for use by "diff":

```
type (day) := cyc
```

The abbreviations for nominal, linear and cyclic are nom, lin and cyc, respectively.

The instruction format for invoking "diff" is as follows:

```
diff(grp1, grp2, grp3; params, results)
```

The "params" and "results" tables are optional. The system provides default parameters to GEM if a parameters table is omitted. If the "results" table is included, the discrimination rules produced by GEM will be placed in it. A variable number of groups (event tables) may be submitted. A semicolon indicates the end of the list of event tables, as after grp3 above.

5.8.4. AQ11

AQ11, like diff, takes a number of event classes and attempts to find the conceptually simplest rules that will determine the class of each event. AQ11 also uses the *Aq* algorithm. AQ11 provides a more extended knowledge representation than *GEM*. The reader is referred to [Michalski and Larson, 1978] for details.

5.8.5. Esel

The operation "esel" invokes ESEL/2 [Cramm, 1982], a program that takes a large number of examples and selects a small subset of examples that is most representative of the larger group. The smaller sample will require less execution time in inference programs such as CLUSTER/paf or GEM. Very large numbers of examples (more than 200) would probably require inordinate amounts of processing time by these programs, making it useful and efficient to choose a representative subset.

5.6.6. Varsel

The **varsel** instruction invokes a program called PROMISE [Baim, 1982] which selects the most "promising" attributes for differentiating between classes of events. Its output is therefore intended for use with GEM. The elimination of irrelevant attributes is a horizontal reduction of the database somewhat comparable to the vertical reduction accomplished by "esel".

5.6.7. Varcon and Varcont

The **varcon** instruction (variable construction) invokes a program called NEWVAR [McMillan, 1982] which attempts to use mathematical operations (multiplication, addition) to create new attributes from combinations of existing attributes. The use of ratios or differences of existing attributes sometimes provides simpler and more accurate rules for distinguishing one class from another.

The command **varcont** is used to access a program named CONVART [Davis, 1981], a system for inducing time-dependent information from data. Multiple measurements of an attribute over time can be changed into a single attribute based upon its time-dependent characteristics. The induced description of the time-dependent attribute can then be used in data or input to other inference routines.

5.6.8. Other Operations

The **apply** operation tests the performance of induced rules on new events. It currently a part of the AQ11 [Michalski and Larson, 1978] program. The output is a confusion matrix that gives the percentage of false positive and false negative decisions for each decision category.

Another inference operation, **treecon** uses program OPTREE [Layman, 1979] to produce optimal decision trees from extended entry decision tables [Michalski, 1978a]. It performs the conversion of VL rules to decision trees (branching logic) for the convenience of the user.

There are three low-level inference operations that are used in CLUSTER/paf that also could be invoked separately.

- (1) **Sim** (similarity) takes any two events and calculates a syntactic similarity measure. The similarity of two events is the inverse of the syntactic distance measure used in CLUSTER/paf.
- (2) **Reun** (reference union) takes the values of attributes and "collapses" several events into one event with multiple-valued attributes. For example, the events
 (12, medium)
 (13, large)

could undergo reference union to become

(12 v 13, medium v large).

- (3) **Gen** (generalize) goes one step further to take the resulting events from the reference union and generalize them into more intuitively succinct values. Thus
 (12 v 13 v 14 v 15, medium v large v verylarge)
 would become
 (12..15, >small).

5.7. Macro Language

A macro processor is available to the QUIN user. This is simply a modified version of the standard preprocessor (/lib/cpp) for the C language compiler (/lib/ccom) which runs under Berkeley standard UNIX. This handles macro expansion and include file copying for the C compiler. The significant modifications are a change to the #define directive to allow multiline macros (the original cpp allowed escaped newline characters in definitions, but they were essentially deleted from the macro as soon as processed) and a change to the I/O to allow the preprocessor to communicate with QUIN over a pair of sockets (a standard UNIX message passing system for concurrent processes).

A macro definition begins with #define and ends with the symbol "@". Thus,

```
#define x(y) sqrt(y) @
```

is a macro which does not contain any embedded lines, and

```
#define x(y,z)
    print("y*z = %d",y*z);
@
```

contains embedded lines.

The other constructs (#ifdef,#ifndef,...) are allowed within macros, and the processor is noticeably more powerful than the original cpp. For full details, refer to documentation on the C language or to the UNIX documentation for "cpp". Note that arguments may be omitted from a macro call with the effect that they become instantiated to null strings and returns TRUE for "#ifndef".

5.8. Program Description

QUIN is written in Berkeley PASCAL for the UNIX operating system. It consists of three major segments of about 2000 lines of code each. The files containing these segments are named qqmain.p qqparsr.p and qqex.p. Qqmain.p contains the initialization routines, the session handling routines, and the utility routines. Qqparsr.p contains the command parser, and qqex.p contains the command executor.

Flow of control through the code follows the pattern of:

- 1) start in the session routine,
- 2) parse a command into PT (the parse table),
- 3) go to the executor and execute the command,
- 4) cycle

There are also three other small pieces of code, in files named qqprocs.h, qqconst.h and qqfunc.c. Qqprocs.h contains the external declarations of procedures which are accessed by more than one segment. Qqconst.h contains the constant, type, and variable declarations as well as the definition of the qqstatic area. Qqcunc.c is written in the C language and contains the system dependent operating system calls or process communication that are not available directly from PASCAL.

When inference commands are invoked, there are two options for the invocation of the inference program. If the session is interactive and the user specifies a "results" table, then the inference program is invoked and QUIN waits for it to finish before continuing. In any other circumstance (not interactive or no results table specified), the inference program is spawned as a separate process independent of QUIN.

CHAPTER 6

The Rule Parser

6.1. Introduction

This chapter describes the GVL₁ rule parser. The parser takes a knowledge base written in GVL₁ and puts out a network representation compatible with the tuple manager. The parser was originally coded by Robert Stepp. Carl Uhrik has made numerous enhancements to the original program.

The network output by the parser is simply a stream of parent nodes and descendant nodes. The basic unit sequence is a parent followed by a list of descendants. This list is internally a vector, called a TUPLE. Hence, the input to the PARSER is a specification of variables, constants, rules and functions, and the output is a series of tuples which are subsequently assigned an interpretation in the context of a control scheme during execution of an application system. This interpretation is partly fixed by a convention of predefined symbols called parser marks (See appendix A.3).

6.2. Language for Rule Representation

This section outlines the language provided by ADVISE for describing and entering rule-based knowledge.

6.2.1. Rules

Rules have two major components: right hand sides (RHS) and left hand sides (LHS). LHS are conditions to be satisfied, and RHS initiate actions or decisions based upon the values of variables. The LHS and RHS have weights (α and β respectively) that correspond to the strength of the assertion in either a backward or forward direction (See Section 1.2.2).

6.2.2. Right Hand Sides (RHS)

A RHS is a concatenation (conjunction) of one or more of the following constructs:

```
[variable = value]
[variable = variable]
[variable = expression]
```

An expression is an integer, real number, symbolic value, variable, function or any arithmetic combination of these. The action caused by evaluation is to take the expression value on the right of the equal sign and assign it to the variable on the left of the equal sign. The strength associated with the implication is attached as a confidence to the value assigned to each variable.

6.2.3. Left Hand Sides (LHS)

The structure of the LHS features a wider variety of constructs than the RHS. These constructs are listed at increasing levels of detail below:

- (1) A LHS: consists of one or more linear modules. Linear modules are separated by disjunctions (\vee).
- (2) A Linear Module: consists of one or more linear module parts. Linear module parts are separated by sums ($+$). An optional weighting coefficient or pair of coefficients may be placed before each linear module part. In the former case, the coefficients must sum to one. In the latter case, the coefficients represent Bayesian LS/LN pairs as described for the BABY System (see section 13.5.2).
- (3) A Linear Module Part: consists of one or more selectors. Selectors are separated by one of the following logical operators: 1) OR (\vee), 2) AND (no symbol), 3) implication (\rightarrow), 4) equivalence (\leftrightarrow) or 5) exception (\backslash).
- (4) A Selector: consists of an expression, a relational operator and a reference surrounded by square brackets ($[]$). The relational operator is one of { $<$, $>$, $=$, $<=$, $>=$ }.
- (5) An Expression: can be either a value, integer, real number, variable, function or any combination of these separated by arithmetic operators. This operator is one from the set { $+$, $-$, $*$, $/$, $\%$ }. The symbol ($\%$) denotes modulus.
- (6) A Reference: consists of one or more reference values separated by commas. Each reference value can have a weight associated with it. The value/weight pair is separated by a colon. Each weight can be one or two real numbers. In the second case, the first number is the truth weight and the second number is the falsity weight (i.e., how much the selector's failure contributes to the falsity of the condition containing it). A reference can be an expression, and a weight can be an expression.

An example of a LHS is shown in Figure 33. The example consists of a single linear module with three linear module parts. Parenthesis are added for clarity. The first linear module part has two selectors in a disjunction. In all the selectors the expressions and references are simple variables.

6.2.4. Rule Groups

A rule group consists of a named set of rules. Rule groups provide a mechanism for focusing the evaluation of rules toward a particular problem solving strategy or subproblem area. Each rule group references a block of variable declarations and, optionally, a block of

```
0.3([F = MKT][G = MKT]) +
0.4 0.01 ([H = MKT]) +
0.5 0.02 ([J = MKT][I = MKT] V [H = MKT])
```

Figure 33 : A LHS.

function declarations.

Examples of the use of multiple rulegroups can be found in the PLANT/ds expert system (see Chapter 11) and the ALFALFA expert system (see Chapter 14). In the first, one set of rules (derived inductively by machine) focuses the consultation on a short list of candidate soybean diseases, while a second rulegroup (written by a human expert in plant pathology) performs a more detailed evaluation of the plausible diseases. In the second system, twelve distinct rule groups are employed to descend through various levels of identification of insect pests found in alfalfa fields. Dynamically changing goals, preconditions and postconditions on rule groups expressed in the rule language, direct the control scheme in execution of appropriate rule groups.

6.2.5. Variables

Variable definitions are parsed independently of the rules and define each variable's domain. Variables in ADVISE can be of two types:

- (1) **Nominal:** The values are simply symbolic names and no ordering is implied between the names. For example, the colors "blue", "green" are might be values of a nominal variable.
- (2) **Interval:** The values of the variable can be a range of integer or real numbers, or symbolic names (e.g. June..September). In the case of interval variables, an ordering is implied.

Variables are declared in blocks, and each block is assigned a name. If a variable is declared from the special GLOBALS block, it is implicitly known to all of the rule blocks following.

6.2.6. Functions

Since the ability to parse functions constituted a major impetus for this project, fuller details of the syntax and semantics are given here. Functions come in two varieties: 1) trap functions or 2) memo functions. Both the bodies and calls to memo functions are parsed by the parser, but only the calls to trap functions are handled by the parser, not their definitions.

Trap functions correspond to procedural knowledge and are handled as jumps to specific pieces of PASCAL code. Use of trap functions is described in Chapter 10.

Memo functions correspond to the classical notion of functions, except that two simultaneous specifications of their definition may be active: 1) a tabular definition consisting of a table of domain/range values or 2) a formula definition from which function values can be computed. When a function is called, a check is made to see if a table exists. If the table does exist, and the values for the arguments are found in the table, the function value is simply retrieved from the table. If no table is present or no entry for the function arguments is found, the formula definition is used to calculate the value and a new table (or table entry) is added.

Different rule groups can share the same function or variable blocks. In general, it is the responsibility of the user to verify the compatibility of the variables and functions between their definition and their use in the rule group.

A function in ADVISE is a mapping of values of arguments to a (value,confidence) pair. The confidence here is a default which may be alternatively specified by a control scheme override in the evaluation of a rule set. To represent the description of the correspondence $F(arg_1, arg_2, \dots) = (value, confidence)$, we have in tabular form:

Funename [$\arg_1, \arg_2, \arg_3, \dots$]
 ($\text{varval}_{11}, \text{varval}_{12}, \dots$) ($\text{funcval}_1, \text{conval}_1$)
 ($\text{varval}_{21}, \text{varval}_{22}, \dots$) ($\text{funcval}_2, \text{conval}_2$)
 .
 .
 .
 ($\text{varval}_{n1}, \text{varval}_{n2}, \dots$) ($\text{funcval}_n, \text{conval}_n$)

Where...

Funename is the name of the function;
var is a variable name;
varval is the value of the variable;
funcval is the function value for some argument values and
conval is the corresponding confidence.

Note that for the use of weights, a convention was adopted to set the function value to the weight and set the confidence to unity (1.0) if no confidence is explicitly given. This convention also applies to a function used as a reference value.

Imposing an outer structure to allow for multiple function definitions in a block, consistent with the block structure of variable declarations and rule definitions, we have the function grammar specification indicated in Appendix B. Note that there is no provision for type specification. This is due to the fact that basically two types of domains exist in ADVISE :

- (1) NUMERIC - real and integer which can take on any of an infinite number of values which are in common with other variables,
- (2) SYMBOLIC - identifiers which are unique to a variable by virtue of its declaration (even when two ids have the same printname, they have separate internal names).

In the latter case, a function defined with one set of arguments is bound to those arguments. In the former case, the generality of the internal representation allows the liberal interpretation desired. The responsibility is left to the control scheme to verify that a rule utilizing a function is calling the function with proper arguments. This is facilitated by the elements of the function tuple, which indicate the number of arguments, argument characteristics, etc.

Functions can take one of three possible forms which are distinguished primarily by their declaration syntax. The three forms are illustrated in skeletal form below:

- 1) numeric functions -- for integer/real values in weights, number variables, etc., which have the declaration,

```
function name ( arg1 , arg2 , ... )
  [ n11  n12 ... funcval1 ]
  [ n21  n22 ... funcval2 ]
  .
  .
  .
```

:= arithmetic expression of arg1 , arg2 , ... ,

where the arguments, and either the table of values or the expression are optional;

- 2) boolean functions — for the representation of selectors or logical components of a rule's LHS which have the declaration form,

```
function name ( arg1:type1 , arg2:type2 , ... ) : BOOLEAN
    { val11  val12  ... funcval1/
    { val21  val22  ... funcval2/
    .
    .
    .
    := VL expression of arg1 , arg2 , ... ,
```

where the table of values or the expression are optional, but types are explicit, or implicitly assumed to be integer/real if omitted;

- 3) symbolic functions — for the representation of variable values within selectors, having the form,

```
function name ( arg1:type1 , arg2:type2 , ... ) : ftype
    { val11  val12  ... funcval1/
    { val21  val22  ... funcval2/
    .
    .
    .
```

where the table of values is mandatory and types are explicit, or implicitly assumed to be integer/real if omitted.

Note that the expression forms are functions that return either the reserved type BOOLEAN (TRUE/FALSE) or numbers. No expression can return a symbolic value. Any variable which is declared in the variable block assigned to this function block (or in a GLOBALS block) may appear in the expressional definition of the function, not just in the argument "dummy" variables. Note that a variable is first checked for a possible local context and failing that, the variable is assumed global. Further, any use of a global in a function declaration temporarily makes it local in scope. A GLOBALS block is intended to affect all the blocks to follow it by entering symbols permanently into the symbol table. The syntax is similar to PASCAL. An example follows. (Note however that the TYPE keyword is unreliablely implemented — i.e. no cross-checking.)

```

GLOBAL
TYPE
BINARY = ( PRESENT, ABSENT );
COLOR = ( RED, BLUE, GREEN );
VARS
RULEGROUP : ( INDUCTION, EXPERT, TREATMENT );
DAMAGECOST: INTERVAL (0..1.0);
USER   : (INEXPERIENCED, EXPERIENCED, SUPERUSER);
END
```

```

VARSX VARS
  X1 = ( DRY, WET, NORMAL );
  X2 : BINARY;
  X3 : BINARY;
  X4 = ( BIG, SMALL );
  T = ( 0.0 , 1.0 );
  Z = ( 0.0 .. 10.0 );
END

```

```

FUNCX FUNCS
VARS = VARSX;
F1 [X1:PRES_ABSENT,Z3965,W235] : COLOR
  (ABSENT , 25.3 , 0.01 , RED , 1.0)
  (PRESENT, 0.23 , -6.7 , BLUE , 0.9)
  (ABSENT , 0.0 , 0.0 , GREEN, 0.8) ;
F2 [Z3966,W236]
  (PRESENT, 0.1 , 0.1 , 0.5 , 1.0)
  (PRESENT, 0.2 , 2.5 , 0.7 , 0.1)
  := 3.0 * Z3966 - 2.3 * W236 * Z3966 + 8.789 ;
END

```

```

RULESX RULES
VARS = VARSX
FUNCS = FUNCSX
  *
  *
  *
[X1=PRESENT][X2=ABSENT:F2(ABSENT,23.1*12.6,Z*8.2)] :> [T=1.0];
END

```

Note : The VARS declaration in the function block is optional unless a function requires an implicit parameter for a functional expression computation. The type of a function or arguments not specified are assumed to be integer/real. The only predefined type is LHS, and it is reserved for function typing (ie., it may not appear on a variable). Further, a function of type LHS cannot have any predefined tabular values. There is a local override in effect if a variable is declared to have a symbolic value already possessed by a GLOBAL type. The possible local context of a symbolic value is checked before it is deemed GLOBAL. Both variables and the symbolic types that occur in the GLOBAL block are global. This means that a global variable will be entered into the dictionary for the network. Also, a global is not required to be used as a variable, i.e., it may exist solely for the purpose of sharing a type between two or more variables that are local (but wish to pass their symbolic values into functions or back from functions or test for equality of symbolic values).

8.2.7. Properties

Properties can be attached to rule groups, rules, variables or functions. These properties are used by the control scheme to perform various tasks. For example, in the PLANT/ds expert system (Chapter 13) uses properties to:

- 1) provide text for asking questions,
- 2) determine whether, for a particular variable, the user should be asked to provide a confidence in their response,
- 3) to provide a detailed description of variables and
- 4) to bias a variable's importance.

Properties are treated as simple pieces of text and are not involved in rule evaluation.

8.3. Parser Construction

Some of the code that makes up the compiler was generated by the YACC compiler compiler. YACC provides a general tool for imposing structure on the input to a computer program. The YACC user prepares a specification of the input process. This includes grammar rules describing the input, code to be executed when these grammar rules are recognized as applicable, and a low level lexical routine to do basic input. YACC then generates a procedure in the form of a C-program to control the input process. This is in fact the PARSER. When compiled and executed, it calls the low level input routine (lexical analyzer) to detect the basic input symbols (tokens). These tokens are organized according to the input rules; when one has been recognized as an applicable grammar rule, the user code supplied with the grammar rule is invoked, returning values and using the values returned by previously executed rules.

This allows the PARSER to be specified by a number of production rules involving nonterminal symbols and terminal tokens. Precedence of operators may be specified to resolve any ambiguities in the grammar and certain error actions can also be invoked. The nature of the PARSER as consequence of YACC, specifically that it is a left-accept left-reduce (LALR) shift PARSER, restricts the form of many constructs. For instance, recursion is preferably left-recursive : $A \rightarrow Aa$, rather than $A \rightarrow aA$. For a full discussion of YACC, the reader is referred to the YACC user manual available through Bell Labs [Johnson, 1978].

8.4. Basic Features

All parser input is free-format. Blanks between lexical entities are ignored. Line boundaries are not significant, but the user should be aware of a limited print buffer. Use of wide line input is consequently not recommended. In the case of reading an empty file, the PARSER does nothing. Some information regarding input format and error handling is presented in Appendix A.

8.4.1. The PARSER Production Cycle

The YACC compiler takes as input a context free grammar with calls to semantic action routines supplied among the production rules. These routines are designed and coded by the implementor and ultimately produce any output apart from duplication of the input symbols. They accomplish the unit operations of the PARSER based on the current input symbol in the context of the particular current state of the parse tree. One of the most important of these routines is the lexical analyzer which recognizes the surface forms (terminals) of the grammar and is called somewhat implicitly (ie., it does not occur within any production rules of the grammar).

YACC output is a file called `y.tab.c`, which is intended to feed the C-compiler to

generate a load module containing a simple automaton which executes an LR parsing algorithm whenever a parse is desired and in effect has two components :

- (1) Parser State Table - an integer representation of the states needed in execution for maintaining the status of a parse. It is similar to the states in a deterministic finite automaton.
- (2) Parser Semantic Invocation Code - the core of the interpreter which actually calls the semantic routines given the proper states as they arise based on the input stream and the given grammar. It is essentially a case statement indicating which semantic actions to call at any point in execution.

Normally, YACC generates C-code for use by the standard compiler supplied with the UNIX system, but the semantic actions are more naturally expressed in PASCAL routines which draw on the rich library of the existing ADVISE system, especially the TUPLE MANAGER module which places and manipulates tuples in a network. Since the code implanted in the grammar is functionally only a specification of calls to procedures, the C-compiler is easily displaced with a PASCAL PREPROCESSOR which edits the C-code to conform to Pascal code.

This is accomplished by splitting off the state table to be read by our Pascal program and editing the executable C-code to look like Pascal. Part of this code is static for all generations of the PARSER and remains untouched, hence the program that does the conversion is somewhat simplified. The files "edparsem" and "edpartab" contain edit commands to extract the constituents (1) & (2) from the file y.tab.c and produce the files "parsem.i" and "partab" respectively. Then, partab is instated in the ADVISE "kbtext" library, and parsem.i in "kplib", reduced to a case statement as to what semantics to invoke, is automatically transplanted into the file "prsourc.p" upon compilation by a "#include" statement . The lexical analyzer and semantic action routines are already resident there and remain relatively static as they are completely oblivious to the YACC compiler (see Figure 34).

In general, this system succumbs to two types of errors. One is in the grammar and the mechanics of grammatical parsing devoid of the semantic routines. The grammar can be ambiguous or allow undesired language constructs. The other type of error is in the coding of the semantic action procedures. These can be changed directly without touching the interpreter code installed from PARSEM.

6.4.2. Operation and Use of the Parser

The file prsourc.p is compiled to produce a load module which reads in the PARTAB file as its first priority. Subsequently, it reads input from the file specified as standard input, generates tokens, parses these tokens, generates tuples, and writes to the assigned OUTPUT file. Additional textual information, mainly diagnostics, is written to the file 'msg'. This process is illustrated in Figure 34.

On the VAX11-780, the current host computer for ADVISE, the parser is executed by typing the following:

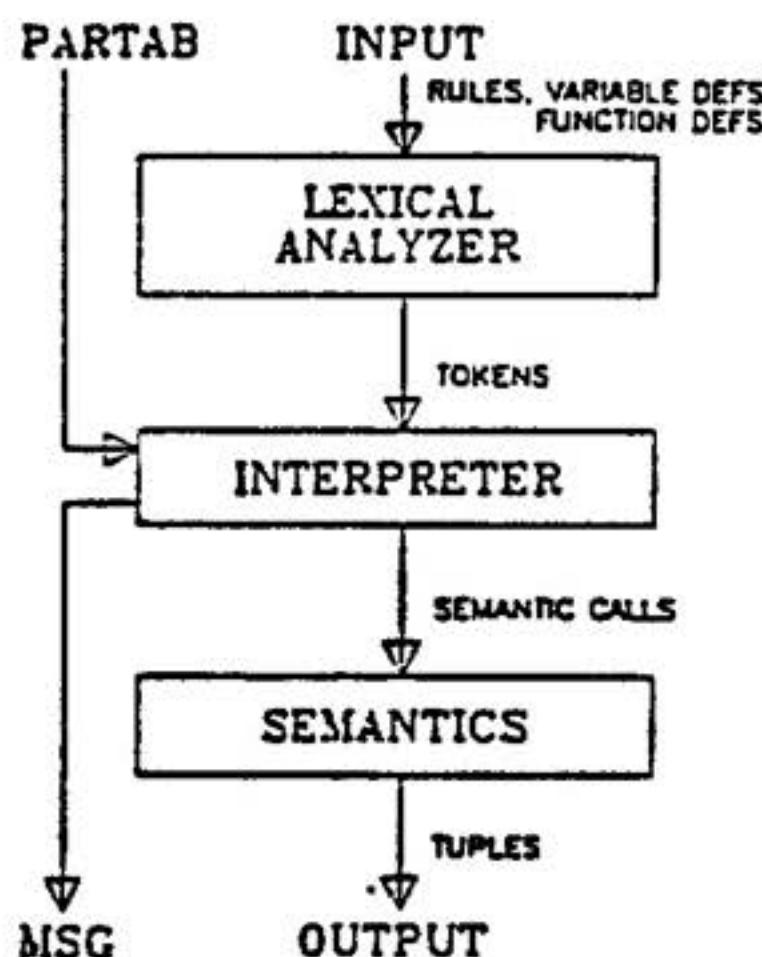


Figure 34 : Relation of Compiler Components.

\$ADVISE/parser/prsourc <input >output

where the following files are involved...

input is a set of variables, functions, and rule knowledge bases;
output is a trace of the input file with error messages;
msg contains program debugging and performance information and
backup contains a network representation of the knowledge
 base in a form that can be read by the TUPLE MANAGER.

The knowledge base can at this point be examined by running the "tmtestr" program as below.

```
$ADVISE/kplib/tmtestr
? os          open standard network
? rb          read backup file
backup file : backup
? wt          write text file
text file name : net
? q           exit
```

At this point, a text format of backup file "backup" will have been written to "net", and this is very useful for debugging.

6.5. Updating the PARSER

The process of updating the PARSER is outlined as follows :

- (1) Assess the need for any new tokens, symbols, or PARMARKs. Modify the lexical analyzer to accept these and return a valid token.
- (2) Derive a set of grammatical productions which unambiguously parse the desired constructs.
- (3) Determine the desired convention for the structure of a tuple to represent the construct in network form. This will depend on the intended use in the rule evaluator or control scheme.
- (4) Write any new semantic action routines that are required to build the network of (3). These routines should ultimately go into "\$ADVISE/kplib/prsourc.p".
- (5) Insert the semantic action routine calls in the proper points in the grammar of (2). The current grammar is in "\$ADVISE/parser/gramf".
- (6) In the directory "\$ADVISE/parser", type the UNIX command, "YACC <infile>", where <infile> denotes the name of the file containing the grammar. The file "y.tab.c" results which contains the compiler written in the "C" language.
- (7) This step and the following two steps edit the "y.tab.c" and extract portions that are inserted in the PARSER. If the shell script "pyacc" can be executed successfully (w/o errors), these steps can be eliminated. First a portion of the grammar, represented by a large "case statement", is generated by typing: "ed <edparsem y.tab.c". The file parsem.i results.
- (8) The state table is extracted by typing: "ed <edpartab y.tab.c".
- (9) This step searches for labels in "y.tab.c" in order to construct the "OTHERWISE" portion of the code that encapsulates the case statement. Type

```
prlabels > prlabels.i
```

to collect together all the allowed labels that occur in the case statement of "parsem.i". Both "parsem.i" and "prlabels.i" are automatically transplanted into "prsourc.p" at compilation by "#include" statements. The case statement portion (parsem.i & prlabels.i) should be installed in the ADVISE library "\$ADVISE/kplib".
- (10) Install the partab file back in the library "\$ADVISE/kbtext",
- (11) Remake the prsourc file (i.e., compile prsourc.p and load prsourc.o), according to the file "Makefile" in kplib.

To assist users in making changes to the parser, the current grammar definition as seen by YACC is presented in Appendix B. Note that steps 6-9 can be built into a convenient shell script.

CHAPTER 7

The Rule Evaluator

7.1. Introduction

This chapter describes the the Rule Evaluator, a module of the ADVISE.1 system that evaluates rules and propagates hypothesis-supporting evidence through the rules. The Parser module parses rules into a tree structure in pre-order form. The rule evaluator recursively descends into this tree in order to determine the “truth value” of some portion of it. In its simplest form, the rule evaluator returns the “truth value” of the left-hand side of a rule. Another user procedure, the rule interpreter, executes a rule’s right-hand side if such action is called for (according to the control scheme).

7.2. User Procedures

The rule evaluator is accessed via three procedures: `reinit`, `evaluate`, and `interpret`.

reinit (schema,schematuplen)

Where:

`schema` is a tuple where the user specifies the evaluation scheme to be used (see section 7.3). Each address in this tuple corresponds to a user selectable option that specifies how certain rule structures are evaluated.

`schematuplen` is an integer value corresponding to the number of options specified in the schema tuple above. There are 17 possible options the user might specify. If fewer than 17 are selected, those not selected are set to their default values.

The `reinit` procedure must be called before using the rule evaluator. It can be called multiple times if the user wishes to use different evaluation schemes within a single consultation session.

evaluate (lhs, val, errnode, error)

Where:

lhs is a rule's left-hand side or some structure within a rule's left-hand side (down to a single selector). It represents the item to be evaluated;

val returns the result of the evaluation as a real number (**val**). It corresponds to the truth value of the rule part passed to the evaluator. In the current implementation, **val** will range between 0.0 and 1.0;

errnode returns the internal name of the node in which an error (if any) occurred and

error is the error type encountered above. These errors may correspond to unexpected conditions that arose during evaluation, but might also contain information useful to the Control module. For example, the error that a rule was not completely evaluated may prompt the Control module to gather more information from the user.

As mentioned earlier, the schema tuple determines how various rule structures are evaluated. Another mechanism for controlling the behavior of the **evaluate** procedure is via accessory control (ACCTRL). The **evaluate** procedure consults the ACCTRL variable and performs certain activities based upon its values. The variable ACCTRL is a Pascal set type which can take on none or more of the values listed below :

- **storintres** : This instructs the **evaluate** procedure to store intermediate results obtained during evaluation.
- **useintres** : This instructs the **evaluate** procedure to use the results stored above. **Storintres** and **useintres** are most commonly used together. They can significantly speed up the time it takes to evaluate a rule, but are not appropriate in certain cases. For example, if the both of these values are added to the ACCTRL set, intermediate results are stored and used at all levels of the parse tree descent. If the Control module has changed the value of a variable in a rule already evaluated (for example, if the user has asked to change the answer to a question), then the evaluator will not descend into the parse tree deep enough to detect this fact. Therefore, intermediate results should not be used if variable values are expected to changes in rules after evaluation.
- **continue** : This instructs the rule evaluator to continue evaluating even if it encounters a variable that has never been assigned a value. The evaluator will still return the error that the rule is not completely evaluated. Variables who have no value assigned are given a "truth value" of 0.5.
- **min** : Used in conjunction with **continue**, this causes the truth value 0.0 to be assigned to variables that have no value. Useful in determining a worst case evaluation of a rule.
- **max** : Used with **continue**, causes the "truth value" 1.0 to be assigned to variables that have no value. Determines the best case rule evaluation.

interpret (rhs,strength,reset,errnode,error)

Where...

rhs is the rule part you wish to interpret; normally a rule's right-hand side;

strength is a real number corresponding to the degree of truth returned by the **evaluate** procedure combined in some fashion with the strength of the assertion of the decision symbol (α).

This value is control scheme dependent;

reset is meaningless (an artifact of a previous function);

errnode is the node that caused an error (if any) during interpretation and

error is the name of the error (if any) that occurred.

In the current implementation, the **interpret** procedure will function correctly only if the **rhs** consists of a simple selector or conjunction of simple selectors.

7.3. Configuring the Evaluator

How the rule evaluator evaluates a rule is specified in the schema tuple, parameter of the **reinit** procedure, which, as was noted earlier, must be called before calling **evaluate** or **interpret**. The numbers below represent the position in the schema tuple. Each position is followed by the meaning of the position to the evaluator, a colon, and, in boldface, the options available for that position. In the current implementation, many positions have only one option — **default**.

- (1) Linear module (evaluation) : **default**.
- (2) Conditional statement (evaluation) : **default**.
- (3) Disjunction of rule parts other than selectors (evaluation) : **default, max, suppes**.
- (4) Conjunction of rule parts other than selectors (evaluation) : **default, min, prod, ave**.
- (5) Disjunction of selectors (evaluation) : **default, max, suppes**.
- (6) Exception (evaluation) : **default, excpt1**.
- (7) Equivalence (evaluation) : **default, equivmaxmin, equivmaxprod, equivmaxave, equivsuppesmin, equivsuppesprod, equivsuppesave**.
- (8) Implication (evaluation) : **default, impmax, impsuppes**.
- (9) Conjunction of selectors (evaluation) : **default, min, prod, ave**.
- (10) Selectors (evaluation) : **default**.
- (11) Conjunction of rule parts other than selectors (interpretation) : **default**.
- (12) Disjunction of rule parts other than selectors (interpretation) : **default**.
- (13) Selectors (interpretation) : **default**.

- (14) Conjunction of selectors (interpretation) : **default, min, prod, ave.**
- (15) Functions (evaluation) : **default.**
- (16) Behavior (evaluation) : **default.**
- (17) Functions (interpretation) : **default.**

The definitions of the available options are listed below :

- **min** : minimum.
- **max** : maximum.
- **ave** : average.
- **prod** : product.
- **suppes** : $x + y - (x * y)$
- **excptl** : x / y (if $y = 0$ then the result is the truth value of x).
- **equivmaxmin** : value = $\max(\min(x,y),\min(1-x,1-y))$
- **equivmaxprod** : value = $\max(x*y,(1-x)*(1-y))$
- **equivmaxave** : value = $\max(\text{ave}(x,y),\text{ave}(1-x,1-y))$
- **equivsuppesmin** : value = $\min(x,y)+\min(1-x,1-y)-\min(x,y)*\min(1-x,1-y)$
- **equivsuppesprod** : value = $x*y+(1-x)*(1-y)-x*y*(1-x)*(1-y)$
- **equivsuppesave** : value = $\text{ave}(x,y)+\text{ave}(1-x,1-y)-\text{ave}(x,y)*(\text{ave}(1-x,1-y))$
- **impmax** : value = $\max(1-x,y)$
- **impsuppes** : value = $(1-x)+y-(1-x)*y$

CHAPTER 8

Knowledge Base Paraphrasing

This chapter describes the paraphrasing capabilities in ADVISE. Section 8.1 describes the design of a general paraphrasing mechanism that has yet to be implemented. Section 8.2 describes a pre-existing implementation of a rule paraphrase module based on the rule evaluation module.

8.1. Design of a General Paraphraser

A general paraphrase mechanism has been designed to paraphrase any tuple in a network. Remember that internally, every rule or table can be represented as a tuple in the low-level ADVISE network. Given a tuple, the paraphraser first locates a paraphrase program for that tuple, then executes that program to produce the appropriate paraphrase e.g. a GVL rule, an English rule, or a fancy table display. The paraphrase program for that tuple may be located in one of several places depending on its generality. Programs with little generality are located directly under the main node of the tuple to be paraphrased. More general programs are found under subnodes (nodes other than the main node) of the particular tuple, or under another node in the network, locatable through an inherit relation or indirection mechanism from nodes in the tuple. For example, the paraphrase program for a selector in a rule may be found directly under the main node of the selector, under a subnode of the selector (e.g. a parser mark that determines the type of selector), or under the rule or rule group in which the selector appears.

The paraphrase program consists of a series of tuples under a given node. The second node in each tuple (the first node would be the main node) is a keyword that identifies the tuple as being part of a paraphrase program. This identifier will be one of MKPARAPHRASE, MKPROG, and MKPROGEND. These are defined below. Subsequent nodes in each tuple will determine a specific paraphrase action. Common actions are finding a new tuple to paraphrase, printing text associated with a node, and setting up a means to paraphrase the tuple iteratively.

The following are the definitions of the keywords recognized by the paraphraser.

MKPAPHRASE	Denotes the start of a paraphrase program. Optional arguments denote the style of paraphrase (i.e. rules, networks or tables) and may also denote a restriction on the tuples that can be paraphrased by this program.
MKPROG	Identifies statements in a paraphrase program.
MKPROGEND	Marks the end of a paraphrase program.
MKLOOP	Marks the beginning of a loop in a paraphrase program.
MKPAPAINDX	This serves as a "program counter" in the sense that it denotes what node in the current tuple is being paraphrased
MKSHIFT	Assigns a specific node to the "program counter" relative to the current position. Default is the shift one node. Further shifts may be specified by an additional argument.

MKABSHIFT	Assigns a specific node to the "program counter" absolutely.
MKLOOPEND	Marks the end of a loop.
MKPRINT	The printname of the node argument which follows is output.
MKPARAPRINT	Paraphrase the tuple that matches the following left context of this tuple.
MKPRINTPOS	Output the printname of the node found at the position in the current tuple specified by a position argument
MKPARAPOS	Paraphrase the tuple that matches a tuple constructed by adding the remaining left context of the MKPARAPOS tuple to the node at the position in the current tuple specified by a position argument.

Figure 35 shows a sample paraphrase program. The first tuple in the program is the MKPARAPHRASE tuple. It marks the start of this particular program. (There may be other programs under the main node RULEGROUP.) This particular MKPARAPHRASE tuple indicates that it can only paraphrase tuples with left context MKEXEC MKAND, which translates to conjunctions of selectors in rules. Note that this program is located under a RULEGROUP node, a likely place from which it can be inherited by all rules in the rule group.

If this paraphrase program has matched the tuple to be paraphrased, then that tuple becomes the current tuple and must be a conjunction of selectors from a rule. With this in mind, the paraphrase actions can easily be understood.

The first MKSHIFT causes the current position to become 3 in the current tuple. This is always a node representing the first selector in the conjunction. The next step (MKPARAPOS) is to paraphrase the node at the current position followed by the remainder of the MKPARAPOS tuple (i.e. MKEXEC). In other words, paraphrase the tuple matching (<selector node> MKEXEC). The paraphraser then recursively finds this tuple, makes it the current tuple, and paraphrases the selector. Assume now that the first selector has been paraphrased and the paraphraser has returned to this program.

```
(RULEGROUP (
  (MKPARAPHRASE MKEXEC MKAND)
  (MKPROG MKSHIFT 3)
  (MKPROG MKPARAPOS MKPARAINDX MKEXEC)
  (MKPROG MKLOOP)
  (MKPROG MKSHIFT)
  (MKPROG MKPRINT " & ")
  (MKPROG MKPARAPOS MKPARAINDX MKEXEC)
  (MKPROG MKLOOPEND)
  (MKPROGEND) )
```

Figure 35: A Sample Paraphrase Program.

The next tuple in the program is an MKLOOP. This begins an iteration of the instructions from here until a matching MKLOOPEND. The loop terminates when an MKSHIFT falls off the end of a tuple. The steps in the loop are easy to follow. The first is an MKSHIFT which makes the next selector (node) in the conjunction (tuple) the current node (position). If this succeed an "and" sign (&) is printed and the tuple corresponding to the selector is paraphrased as before. This continues until there are no nodes (selectors) left in the tuple (conjunction). Here the loop terminates and this paraphrase program completes.

Implementation of the general paraphraser is pending decision on the adoption of an object-oriented environment. The paraphraser's notion of inheritance would be well supported by an object-oriented approach. The only existing paraphrasing mechanism in ADVISE is a rule paraphraser which is described in the next section.

8.2. The Existing Rule Paraphrase Module

The existing paraphrase module is designed to "unparse" rules. It is capable of converting rules in their network form to the GVL₁ form. The rule evaluator was modified so that could also paraphrase rules. The rule evaluation function was retained.

There are two user callable procedures, PAINIT and PARAPHRASE. PAINIT is used to initialize the module and has the same calling sequence as REINIT, namely:

PAINIT (schemetuple,schemetuplelength);

Where...

schemetuple is the tuple specifying the rule evaluation schemes as in REINIT and

schemetuplelength is the length of the above tuple.

PARAPHRASE was adopted from procedure EVALUATE in the rule evaluator module and is called thus:

PARAPHRASE (rulepart,width,textfile,value,errornode,error);

Where...

rulepart is the part of the rule to paraphrase and can be a whole rule down to a selector,

width is the width for the text to be placed in **textfile**,

textfile is the text file to place the results of paraphrasing in,

value is the value of evaluating the rule part and is not defined for whole rules,

errornode is the same as in procedure EVALUATE, and

error is the same as in procedure EVALUATE.

The internal structure of the paraphrase module is the same as the rule evaluator module. The modifications done to the rule evaluator procedures are marked with the comment, (*PA*). This is only a preliminary version of the PARAPHRASE module. Beware of undocumented features.

CHAPTER 9

The Tuple Manager

9.1. Introduction

The Tuple Manager (TM) is an implementation of a generalized network manager similar to that described by Baskin [Baskin, 1979] in his PhD dissertation. By offering a variety of high level primitives with which to manipulate the network, it allows the user considerable abstraction from the Pascal data types actually used to store the network. The user works directly with n-tuples ($n < 256$) which are arrays of symbols to be manipulated. There are actually three simultaneously active networks, which may be distinguished as public and private networks, or allow experiments with a scratch network. The TM also supports procedures that will write out backup files and text files to re-create the network state.

9.2. The Data Types

A tuple is represented by a Pascal array of symbols, which in turn are represented by a variant record. This is analogous to lists in LISP with the exception that the list elements cannot themselves be lists, but must be atoms. Another difference is that the tuple manager's list elements can be accessed directly rather than having to traverse the list with compositions of 'cars' and 'cdrs'.

The four types of symbols supported by the tuple manager are integers, reals, textnodes, and symbol nodes. The most general type of symbol is allowed to have attributes attached and manipulated. These attributes are similar to LISP properties, however the attributes are referenced and stored by keying on as many of the leftmost symbols as desired, rather than one. This will be made clearer in the discussion of some of the tuple retrieval procedures.

Textnodes are designed for the efficient storage of text and therefore, like the integer and real symbols, are not allowed to have attributes. From the user's perspective these n-tuples of symbols are just entities that can be stored, retrieved, and modified in a variety of ways. The semantics of tuples is defined by the user program, but the capabilities of the tuple manager allow for representation of a generalized graph and any specializations. So far use of the tuple manager has been in the realm of rules, tables in a relational data base, and a directed acyclic graph.

For efficient storage and retrieval of the tuples, they are indexed by their first symbol. A symbol consists of a node and its (possibly empty) set of attributes. A node is represented by a Pascal variant record which contains its printname, internalname (a unique node identifier), a pointer to its set of attributes, and various housekeeping variables. Each attribute is stored as a linked list of subnodes. These lists in turn are elements in the attribute list of the node. Each subnode consists of the internalname of the node that it is representing. By referencing other nodes in the attribute list using only their internalname rather than a pointer, the TM can more easily make disk versions of the networks and reread them into main memory.

The tuple manager supports a dictionary for retrieval of a symbol given its printname. When a network is read in from a backup file only the symbols in the dictionary, and any symbols reachable from those, are accessible to the user. The function of the dictionary is similar to the function of the LISP oblist. During a session, though, not all of the network's

symbols may be reachable from the dictionary due to local creation and storage of symbols. The LISP analogy is a 'gensym' atom, which is not directly reachable from the oblist. The tuple manager implementation of the dictionary is a hash table that uses an intable collision resolution scheme.

The procedures for entering, removing, and looking up a symbol are discussed in the next section.

9.3. The User-Visible Procedures

The user-visible tuple manager procedures are listed in alphabetical order below. For the exact calling sequence of these procedures, the user should consult the tmprocs.h file in the source.

Procedure ADDATTRIBUTE. ADDATTRIBUTE takes a tuple as a parameter and adds it to the existing network. This is done by first finding the parent node of the tuple using FINDNODE (section 9.4). The proper place for the remainder of the tuple (the attribute of the node) is determined by a parameter passed to the procedure. This slot is then located and the incore subnode structure of the attribute is created.

Procedure ASCIINT. This procedure converts the ASCII representation of an integer into the machine representation of that integer. This is a machine independent algorithm.

Procedure ASCIIREAL. The real number counterpart of ASCIINT. Also machine independent.

Procedure CLOSENETWORK. CLOSENETWORK saves the state of all three levels of networks in the user's files. See OPENNETWORK. The dictionary is saved in the global level. The housekeeping pointers and variables are saved in each of the files. If one closes a network, signs off the system, and then opens the network, the state of that network will be unchanged.

Procedure CLOSENODE. This procedure takes as parameters the internalname of a node, a flagword to store with the node, and a level on which to find the node. It then locates the node with a call to FINDNODE, stores the flagword with the node, and then decrements the open counter (opencount) of the node. It does not alter the status of the node. A node with an opencount of zero is considered to be closed, and unavailable for access until opened.

Procedure CREATENODE. This procedure takes a flagword as a parameter and a level on which to work, and creates a new regular node, assigning a unique internalname to the new node. The node is not opened - it is just created. The opencount, usagecount, and attributecount are all set to zero. The printname is set to the null word so it will need to be filled in by calling the procedure SETPRINTNAME.

Procedure CREATETEXTNODE. This is a simple procedure that will create a new text node and assign it a unique internalname to it. An integer flagword is passed to this procedure and this is placed in the new text node. The internalname is returned by CREATETEXTNODE. This procedure does not open the node - it just creates the node. The new node is initialized with the opencount, currentline, and number of lines of text all set to zero.

Procedure DELETEDICT. DELETEDICT removes an entry from the dictionary. If it is not found then it does nothing. The stringpool is unaltered by this procedure, but its information is made unusable as is its storage area occupied by the printname. Therefore repeated calls to this procedure will create wasted space in the stringpool that cannot be recovered in the present implementation.

Procedure DELETESLOT. This procedure is the inverse of ADDATTRIBUTE. Given a particular attribute to be removed, it just deletes it from the in-core representation. It does this by first locating the node using FINDNODE, and then traversing the linked list of attributes until it has traversed as many as passed in the parameter slot. The subnode structure of the attribute is then removed. The value for slot that is passed as a parameter must be found by using the procedure GETATTRIBUTE.

Procedure DELETETUPLE. This is also the inverse of ADDATTRIBUTE. Unlike deleteslot, however, it takes a tuple as input. DELETETUPLE locates its parent node using FINDNODE, locates the slot under the parent node containing the tuple, and then deletes the tuple using DELETESLOT.

Procedure DESTROY. DESTROY takes as parameters the internalname of the node to be destroyed and the level on which to destroy it. It will then wipe out the node by first finding the node with FINDNODE, and then returning the node structure to the free node pool. This procedure will not remove a node from the dictionary. This is because a node can exist simultaneously on different levels. An error is returned if a request is made to destroy a real or integer node.

Procedure ENTERDICT. This procedure takes as input an internalname and its printname and stores it in the dictionary if it is not already in the dictionary.

Function EQ. This boolean function takes two internal names as parameters and returns TRUE if they are equal or FALSE if they are not. It first checks the type of internalname(node, text, real, or integer) and then checks for equivalent values depending of the type. In the case of a real node, an acceptable margin of error can be specified using the procedures TMREALDIFF and TMREALINTDIFF for comparisons with other real numbers and integers, respectively.

Procedure GETATTRIBUTE. This procedure takes a tuple and its length as parameters along with a location to begin the search, and returns the complete tuple and the slot under which it was found. The tuple passed to GETATTRIBUTE can be a left context tuple or a complete tuple. If a left context tuple is given then only the first matching complete tuple will be returned.

The procedure works by first finding the node using FINDNODE. It then counts down the attribute list looking for the attribute at which to begin the search for the particular tuple. From there it searches the attribute list in order looking for a tuple that matches as much of the tuple that was passed for the search. As soon as a matching tuple is found, the entire tuple is loaded into the return tuple and the slot under which it was found is loaded into the return slot.

Procedure GETPRINTNAME. This procedure takes as a parameter the internalname of a node and returns either its printname or its value, depending of the type of node. If it is a textnode or a regular node then the printname is returned. If it is a real node or an integer node then its ASCII representation is returned. For the latter two types of nodes the procedure converts the node value to the ASCII representation with calls to either ASCIIREAL or ASCIINT. In the case of a text node or a regular node the procedure first has to find the node with a call to FINDNODE. It then packs the printname into the return parameter.

Procedure INHERITATTRIBUTE. Like GETATTRIBUTE, this procedure takes a tuple and its length as parameters along with a location (slot) to begin the search, and returns the complete tuple and the slot under which it was found. The tuple passed to INHERITATTRIBUTE can be a left context tuple or a complete tuple. Unlike GETATTRIBUTE, however, the search is not restricted to tuples under the main node of the given tuple. INHERITATTRIBUTE also follows inheritance arcs to other nodes to match tuples (ignoring the main node, of course). INHERITATTRIBUTE also looks under subnodes

of the tuple (by substituting the subnode for the main node) for the desired attribute (tuple). The search is exhausted vertically (along inheritance arcs) at each subnode before beginning recursively from the next subnode (horizontally). Additional parameters specify the maximum numbers of inheritance arcs and subnodes to search. The number of inheritance arcs and the number of subnodes searched are also returned. INHERITATTRIBUTE calls GETATTRIBUTE with the appropriate substitutions for the main node.

Procedure INTASCII. This procedure is a machine independent algorithm that converts an integer into its ASCII representation.

Procedure INTERINT. This is a bidirectional conversion procedure that converts to/from internal name representation from/to the Pascal integer representation. Presently there is no difference between the two and the procedure just outputs the proper part of the internalname record passed as the parameter, or fills the proper part of the record depending on the direction of conversion.

Procedure INTERREAL. This is the real number counterpart of the INTERINT procedure.

Procedure LOOKUP. LOOKUP is the dictionary access procedure which takes a printname as input and returns an internalname and nodetype. If a real node or an integer node is passed to LOOKUP, then the ASCII representation is returned. LOOKUP works by searching the dictionary hash table for a match of the printname.

Function NTYPE. This function takes as input an internalname and returns as its value the nodetype of that node. The function is very fast and should be preferably to opening a node just to find out its type.

Procedure OPENNETWORK. OPENNETWORK opens three user networks at the three different levels (global, local, private). It creates the lists of free nodes and subnodes, and arranges all of the housekeeping pointers and variables. The dictionary is also read into the memory.

Procedure OPENNODE. Opennode takes as a parameter the internalname of a node and the type of access desired and opens the node if the access matches the allowed type of access for that node. The node is opened at a particular level, depending on what level is requested, also a parameter. Values returned by the procedure are the number of attributes, the flagword, and the type of node. If one is opening a node merely to determine its nodetype, then a better option would be to use function NTYPE.

OPENNODE works as follows. If the node requested is a regular node or a text node then FINDNODE is called to locate then node and bring it into the core memory if it is on disk. If the node was not located, and if the level requested was not global, this procedure will look at progressively higher levels for the node and copy it to the requested level if the node is found. If the node is located then its opencounter is incremented and the return parameters are set. If the node is not located then an appropriate error message is returned.

Procedure READBACKUP. READBACKUP takes a backup file and re-creates the network exactly as it was when the backup file was written.

Procedure READTEXT. This procedure takes a text file of the network and creates a new network based on the contents of the text file. Every node in the network is entered into the dictionary, making this procedure of very limited utility except for small networks. The procedure will have problems if one attempts to load more nodes than will fit in the dictionary. Currently the limit is 499. This procedure is in need of some modification in the future.

Procedure REALASCII. This is the real counterpart of INTASCII. It is also machine independent.

Procedure SETPRINTNAME. This procedure is the inverse of GETPRINTNAME. This procedure does different things for different types of nodes. For real or integer nodes it sets the value of the node to the Pascal implementation of that value. This is done with a call to either ASCIIREAL or ASCIINT. For regular nodes the node is first located with FINDNODE. Then the node's printname is assigned the printname that was passed as a parameter. For text nodes the node is located and then the printname, passed as a parameter, is interpreted as a line of text which is added to the end of the existing text for that text node.

Procedure TMREALDIFF. See function EQ.

Procedure TMREALINTDIFF. See function EQ.

Procedure WRITEBACKUP. This procedure writes out a backup file that will allow the exact state of the network to be preserved. The dictionary is included in the writeout procedure. Nodes having the same printname are not merged, as in WRITETEXT below.

Procedure WRITETEXT. This procedure will write a text file of the network to a specified filename. The algorithm used will cause nodes having the same printname to be merged. The dictionary is not saved.

9.4. An Important User-Invisible Procedure

Procedure FINDNODE. By far the most frequently used procedure in the Tuple Manager, FINDNODE is designed to be as fast as possible given the limitations of a linked list representation of the network. This procedure takes as a parameter the unique node identifying number and returns a pointer to the internal memory representation of that node. It works as follows. All of the nodes in the memory are hung onto a hash table using a linear chaining collision resolution scheme. The proper hash table entry is determined from the node identifier and that particular linked list is searched sequentially for the node.

In an earlier implementation, the tuple manager was directly responsible for virtual memory management — i.e., ensuring that memory used for nodes did not exceed the core limits, and swapping nodes from disk to core when core memory was full. In that implementation, if FINDNODE did not find the node in core then the area on the disk that should be occupied by the node was examined. If the node was on the disk then it was brought into the core network. A return pointer was then set to the node. If the node was not in memory and not found on the disk then nil was returned.

In the current implementation, all virtual memory management is performed by the Unix operating system. From the user's point of view all of the nodes are in core at the same time.

CHAPTER 10

General Utilities — TESTER and the Special Functions Module

This chapter describes two utilities available for use in ADVISE. The first is the Tester Program which can be used as a low-level network debugger for ADVISE knowledge bases. The second utility is the special functions module: a set of hard-coded 'trap' functions available for reference from rules in the knowledge base.

10.1. The Low Level Network Debugger

The Tester program was originally developed as a tool for testing the tuple manager. It has evolved into a program that can be used to perform tests on the rule evaluator, the paraphraser and the tuple manager. In addition, it allows the user to debug or otherwise manipulate a knowledge base after it has been parsed. This section describes the functioning of the Tester module. The program was coded originally by Heinrich Juhn. Lance Rodewald and Albert Boulanger have supplied additional enhancements.

10.1.1. How to Use the Tester Program

The Parser reads in a rule base and outputs a network or *backup* file. The Tester program allows the user to examine the *backup* file and edit it. Before looking at an example of a portion of the *backup* file, some definitions are in order:

- (1) **node:** A node is simply a symbol. Each node has an internal name that is hidden from the user and is accessed via a printname. A printname is simply a string of characters that represents a unique identifier for a node. These printnames are collected into a dictionary that is manipulated by the tuple manager.
- (2) **tuple:** A tuple is a vector of nodes, integers or real numbers. A knowledge base is represented internally as tuples. The node in the first position (slot 1) in a tuple has a special significance. It represents the place where information about the node is stored. This same node may appear in other tuples in other than the first position. In this case, the node can be viewed as a pointer back to that tuple in which the node appears in the first position.
- (3) **slot:** A tuple's slot corresponds to its vector address. For example, a tuple is declared as:

```
tuple: array [1..tuplelength] of internal_names;
```

Slot number n ($1 \leq n \leq \text{tuplelength}$) is the nth element of the vector (tuple[n]).

- (4) **attribute:** An attribute is a tuple representing a property about that node that appears in position 1. A node may have several properties "hung under" it.

For example, consider a knowledge base that contains an attribute called TEMPERATURE. It can take on the values BELOW_NORMAL, NORMAL, or ABOVE_NORMAL. It would appear in the following form when viewed via the Tester program:

```
(TEMPERATURE (
    (MKDOMAIN MKNOMINAL BELOW_NORMAL NORMAL ABOVE_NORMAL)
))
```

The node TEMPERATURE (its printname) has under it a single attribute that specifies the possible values for the node. The identifiers MKDOMAIN and MKNOMINAL are semantic markers used internally by the program to identify this attribute as representing the domain of the variable TEMPERATURE, and that domain is scaled nominally.

10.1.2. Tester Commands

This section contains a list of commands available in the Tester program. Each command is one or two letters, and the user may be prompted for additional information after requesting a particular operation.

Basic Commands

These commands perform the basic network manipulation functions.

- h:** The (h)elp command displays a list of commands available to the user.
- os:** The (o)pen (s)tandard command prepares the program for (r)eading a (b)ackup file. In general, this is the first command a user should issue. More than one *backup* file can be manipulated in a single interactive session with the Tester program. In this case, the user should issue the (c)lose command before another (o)pen (s)tandard command.
- rb:** This command (r)eads a (b)ackup file. The user is prompted for the name of that file. The user should not issue another rb command in the same interactive session without issuing the (c)lose command first.
- wb:** Used to (w)rite out a (b)ackup file. The user is prompted for the name of the file to create for storing the network. This is useful when the user wishes to keep changes that have been made to the network.
- wt:** This command will output the entire network in a readable form. It is useful for hand-checking parser output and gaining some insight into how a knowledge base is stored.
- rt:** This command reads a file in the readable form that is output by the wt command. It is particularly useful for adding new rules to a network without going through the trouble of re-parsing.
- c:** The (c)lose command frees internal storage used by the Tester program. It should be issued before reading another *backup* file in the same session.
- or:** This command functions identically to the os command in the current implementation.
- b:** This command allows the user to set the level of tuple manager debugging information. This information is always placed in the file named "msg" which can be examined at the end of a session.
- n:** Display a (n)ode and all of its attributes. The user is prompted for the name of the node to be displayed. The command may fail if the node's printname is not in the dictionary (initially, very few nodes are in the dictionary). A node's printname can be entered in the dictionary using the (m)ark command.
- g:** This command can be used to (g)et an attribute hung under a node. It establishes this attribute (if the command does not return an error) as the *current tuple*. A *current*

tuple is assumed to exist when using other commands (See, for example, the (m)ark command). The attribute tuple is accessed by specifying node printnames beginning with the node in slot 1. The user is asked to provide a "left context" for identifying the attribute he wishes to access. This "left context" is simply one or more printnames in the order they appear in the tuple, beginning with slot 1. The user is prompted for the number of printnames he wishes to use as a "left context" and then is asked to enter those printnames. For example, in the TEMPERATURE tuple presented earlier, this tuple could be accessed by entering the number "2" when prompted for length and then the names "TEMPERATURE" and "MKDOMAIN" when prompted for the printnames. Remember that the printnames must be in the dictionary for this command to succeed (you may have to use the (m)ark command to enter them). Also note that the case of the letters is significant when matching printnames. A simpler mechanism for looking at the network is provided by the "s" commands (sw,sp,etc.).

- gi: This command also gets an attribute from the network as does the g command. The gi command, however, uses an inheritance mechanism to find an attribute. The inheritance mechanism is described in the previous chapter under INHERITATTRIBUTE.
- a: This command (a)dds a tuple to the network. As in (g)et, the length of the "left context" is requested and printnames of that context. The tuple is hung under the node in slot 1 of the "left context" if that node exists in the dictionary, otherwise the node is created.
- d: This command is used to (d)elete a tuple. It functions just like (g)et and (a)dd. (See also st below.)
- m: Used to (m)ark a node (i.e. enter its printname into the dictionary). The user is prompted for which slot (element) in the *current tuple* he wishes to mark. A slot is referenced one of two ways: either by its slot address as a positive integer (counting from the left) or by its slot address as a negative integer (counting from the right). For example, if the *current tuple* has length 3, its last slot can be referenced by either 3 or -1.
- l: This command causes a (l)evel change. Three copies of each knowledge base are maintained: the global, local and private level copies. Initially, the global level copy is manipulated by the Tester program.
- p: Used to (p)urge a printname from the dictionary. The user is prompted for that name.
- x: The x command deletes a node that has no attributes. If the node has some attributes hung under it this command will fail (the d command should be used first to delete all attributes).
- tr: Retrieves and displays a line of text from a textnode.
- ta: This command will retrieve and display all the lines of text in a text node.
- ts: This command will store a line of text under a textnode. The user is prompted for that line of text.

Network Traversing Commands

The group of commands that follow have been added in order to simplify the process of descending into a knowledge base network. In particular, they allow the user to look at parts of the network without having to enter a series of (m)ark commands followed by (g)et

attribute commands to descend into the network.

- sg: This command gets the first tuple in the network. The user is prompted for its printname and it becomes the *current tuple*.
- sw: Prints all of the attributes under the *current tuple*.
- sn: This command is followed by a space and then an integer. It prints the n-th attribute under the *current tuple*.
- sd: This command is followed by a space and then an integer. It descends to that node in the *current tuple* as specified by the integer.
- sr: This command ascends one level.
- sa: This command will add an attribute under the current descent level.
- st: This command is used to transfer the *current tuple* as seen by the "s" commands into the *current tuple* as seen by the "d" command. It is useful for deleting tuples.

Module Testing Commands

The commands below are used to test the rule evaluator and the rule paraphraser. All of these operations affect the node in slot 1 of the *current tuple*. This node in most cases corresponds to the printname of a rule.

- js: Initializes the rule evaluator. This command must be issued before attempting to evaluate or interpret a rule.
- jg: Used to put a value/confidence pair under a rule. Before attempting to evaluate or interpret a rule, the variables involved in the left-hand side of that rule should have values. These values can be inserted by using this command.
- jg: This will get the value/confidence pair under a node (if it exists). For example, the user may want to see if a value/confidence pair was placed under the rule's right-hand side after interpreting that rule.
- je: This command will evaluate the current node (usually a rule).
- ji: This command will interpret the current node.
- ju: Used to initialize the rule paraphraser. Must be issued before attempting a jt command.
- jt: Used to paraphrase the current node.

10.2. The Special Functions Module

The Special Functions Module (aka the TRAP Module) provides facilities for invoking binary coded procedures and functions from rules in an ADVISE network. The rule evaluator uses the TRAP module to evaluate TRAP functions that are in the rules. A TRAP function has the form:

TRAP(trap-number,p1,p2,...,pn)

Where...

trap-number is a unique integer identifying the function, and

p1,p2,...,pn are the parameters to the trap function.

TRAP functions are typically used as in the rule below.

[MOTH_COUNT <> *UNKNOWN*] ::> [EGGS = TRAP(6,MOTH_COUNT)|TRAP(99)]

This rules says that if the moth count is known, first call trap function 6 with MOTH_COUNT as a parameter, then execute trap function 7 with no parameters. Trap function 6 is an actual trap function used by PLANT/cd (Chapter 12). The function computes the egg population from a count obtained from moth traps [Troester82a]. Trap function 99 does not presently exist in ADVISE. It is used as an example here to show that trap functions need not be included in selectors or return a value. Trap function 99 could simply be a procedure which outputs results on a terminal.

Execution of trap functions begins in the rule evaluator, which passes the TRAP number and the parameters (if any) to procedure TRAPFUNC in the TRAP module. Procedure TRAPFUNC has a case statement in which the TRAP numbers serve as case labels. Upon branching, several operations are performed. The TRAP parameters are converted from their network representation to a Pascal representation. Parameters that are nominal are usually mapped into a Pascal scalar variable. Next the semantics are executed. This is usually a call to a procedure or function. Finally the outgoing parameters are converted from their Pascal representation to their network representation. These procedures are lexically scoped within procedure TRAPFUNC, although there are several utility procedures outside. One such procedure is TRINIT which the backward control scheme code calls to initialize the TRAP module.

Currently the trap module contains 21 TRAP functions all used in PLANT/cd. These functions implement the *deep* model of black cutworm damage (BCW) as developed by Steve Troester [Troester, 1982a,b,c,d]. These functions are specific to the cutworm problem and are not likely to be of general use. For a detailed description of the trap functions, the reader is referred to [Boulanger, 1983].

CHAPTER 11

The PLANT/ds System: Rule Based Inference Control

11.1. Introduction

The PLANT/ds consultation program is an experimental expert system used to advise farmers on diseases common to soybeans in Illinois. The program acts as a diagnostician, by asking the user questions regarding problems observed in the diseased crop and returning a list of the most likely disease candidates. A general description of the problem domain and an earlier program can be found in [Chilausky, 1979]. This chapter describes the realization of PLANT/ds within the environment of the ADVISE knowledge programming system, with particular emphasis on the control strategy employed. The program and most of this documentation were prepared by Mark Seyler.

The responsibility of the control scheme is to make the knowledge embedded in the program available to the user by conducting a consultation. The search space represented by the PLANT/ds problem domain is of sufficient size that the control strategy utilizes search heuristics that attempt to focus the consultation on only that knowledge relevant to the user's problem. An approximate reasoning method found useful in this particular domain is selected (from the several available in the ADVISE system). The control scheme is also responsible for explaining some aspects of its reasoning process and providing access to the knowledge it contains. Each of these aspects of control will be discussed in turn.

11.2. The Knowledge Base

Knowledge in the PLANT/ds system is represented as decision rules in the GVL₁ formalism [Michalski and Chilausky, 1980]. There are two types of rules, those that represent hierarchical relationships among variables (candidate rules) and those that reflect the relationship between the variables and the SOYBEAN DISEASE goal variable (goal rules). An example candidate rule is:

```
[LEAF_SPOTS = ABSENT]
  ::>
[LEAF_SPOTS_MARGIN = DOES_NOT_APPLY]
[LEAF_SPOT_SIZE = DOES_NOT_APPLY]
```

In this case, the variable LEAF_SPOTS represents a grosser level of detail than the other two variables. This use of two levels of detail is one mechanism by which attention can be focused as early as possible on that class of diseases relevant to the problem at hand. An example goal rule is:

```

0.8[TIME_OF_OCCURRENCE = AUGUST..SEPTEMBER|
  [CONDITION_OF_LEAVES = ABNORMAL]
  [CONDITION_OF_STEM = NORMAL]
  [LEAF_MILDEW_GROWTH = ON_UPPER_LEAF_SURFACE]
  +
  0.2[PRECIPITATION < NORMAL]
  [TEMPERATURE >= NORMAL]
  ::>
  [SOYBEAN_DISEASE = POWDERY_MILDEW]

```

There is one rule of this sort for each of the possible diseases (currently 20). All of the variables in the rule's left hand side are those deemed relevant to that disease. They are grouped together and weights assigned to each group (linear module) according to the importance of that variable to the disease in question. The weights sum to unity and evaluating a rule of this type can be viewed as accumulating positive evidence that the disease in the rule's right hand side is present.

There are two sets of goal rules (rule groups): those output by the inductive learning program AQ-11 [Michalski, 1978] and those compiled by the expert plant pathologist. The AQ-11 program attempts to find the minimum number of variables that successfully discriminate the various classes of soybean disease. These rules are used to reduce very quickly the number of disease candidates considered by the program. Once the number of candidates is reduced to 5 or fewer, the expert compiled rules are evaluated and used in the remainder of the consultation.

11.3. Control Scheme

It was recognized early that knowledge regarding the consistent diagnosis of soybean diseases was not complete. For this reason, a method of representing and combining approximate reasoning is employed. Furthermore, confidence in a particular observation may be prone to error and for this reason the user is asked to select a confidence in his answer. Details of the evaluation scheme used in PLANT/ds can be found in [Michalski, 1982b].

Aside from the candidate rules, which are used to reduce the search space, the PLANT/ds knowledge base is two-leveled. This does not lend itself well to classical chaining (backward or forward) control strategies that establish intermediate goals. Instead, the problem is one of selecting among one of many possible terminal goals (soybean disease). The primary PLANT/ds control loop is outlined in a Pascal-like pseudo code below:

```

repeat
    select the most useful variable;
    if the value is not known ask the user for it;
    find all those rules in which this variable occurs;
        (in the left-hand side)
    for all of the rules found above do begin
        evaluate the rule;
        if its value is below the threshold then
            eliminate it from consideration;
    until all of the variables have been selected;

```

The control scheme attempts to minimize the number of questions asked of the user. In this respect, it acts in much the same fashion as the human expert, who eliminates those aspects of the problem irrelevant to the task at hand and focuses on what is most relevant. PLANT/ds uses two mechanisms for focusing attention to the most likely disease candidates. The first is by the use of a hierarchical variable structure (the candidate rules). The second is rule elimination by thresholding. Any time a goal rule falls below a certain threshold (determined by the knowledge engineer and domain expert) it is eliminated from consideration. The algorithm for selecting the most useful variable can then be formulated as follows:

```

select a variable that may satisfy one of the
candidate rules;
if none of these are available then
    select a variable with the maximum utility where utility
    is defined as the number of rules in which the variable
    occurs;

```

By defining utility in this fashion, the program will attempt to eliminate as many goal rules as it can as early as possible. The viability of a rule in the PLANT system is defined as the degree of truth of its left hand side. This degree of truth cannot be calculated exactly until all of the variables on the left hand side have a value. However, a best case truth value can be calculated at any stage by assuming those variables that do not currently have a value are satisfied. If a rule's best case truth value falls below the threshold, the rule can also be eliminated from consideration. By eliminating goal rules, any variables they involve can be eliminated, thereby reducing the number of potential questions. In experiments done to date, of a possible 40 questions possible only about 15-20 are asked by the program.

Another aspect of control embodied in PLANT/ds is that of allowing the user access to its knowledge and some aspects of its reasoning. This aspect of transparency is highly desirable in expert systems both because it increases user acceptance and can be used as a tool for testing new knowledge bases. During the variable value request portion of the program a number of options are made available to the user:

- (e) Provides assistance in clarifying the question being asked.
- (v) Lists those diseases still being considered and their best case confidence.
- (w) This option displays the change that would occur in the confidence in each disease for each possible response.
- (r) Displays those rules in which the current variable is found.
- (b) Allows the user to return to the previous question.
- (m) Allows the user to modify his answer to the current question.

After the program has displayed its conclusions, the user can make further inquiries of the knowledge base:

- Program execution statistics can be examined.
- Any rule in the knowledge base can be displayed.
- For any goal rule, a list of those selectors that failed can be displayed. This can be used to determine why a particular disease was ruled out or how the confidence in the most likely disease might be improved.

In addition to the capabilities mentioned above, the control scheme can be reconfigured prior to entering its consultation phase. The user can elect to use only the machine generated or expert rule groups. The breadth of the search can be adjusted by choosing to gather packets of information before evaluating. These packets represent groups of variables whose values are asked of the user prior to evaluating any rules. The thresholds used for rule elimination can be set as a means of balancing the performance of a particular rule set. A final option allows the user to select a novice or experienced mode of consultation. In the novice mode, additional help and guidance is given throughout to consultation as a means of training new users in its function and operation.

11.4. Using the PLANT Program

The PLANT program resides on the VAX-11/780 at the University of Illinois, Urbana-Champaign. It represents one of the earliest experiments in expert systems to be developed within the ADVISE environment and is still undergoing improvement. In parallel, the PLANT program was being "downloaded" onto a small computer (the IBM PC). A sister system TURF was subsequently developed by Greg Smith, and later Bob Reinke and Jiarong Hong readapted this for the Unix environment for rapid compiled execution.

The PLANT program can be executed on the VAX by typing \$ADVISE/kblib/plant. It generates diagnostic and performance messages which can be examined after the session in a file called msg. The program can be configured to run on a particular terminal by re-loading the program with the appropriate device driver or loading the "gkuniversal" and "gktermcap" routines developed to support UNIX termcap conventions (in which case the shell TERM variable should be properly initialized).

11.4.1. Data Structures

The architecture of the Pascal modules is such that each should remain active throughout the life of program execution. This means that a static memory area is set aside for each module. In the case of the control scheme modules, such as PLANT/ds, this area contains a number of useful data structures pertinent to the regulation of rules and variables. This section elucidates the various elements to the PLANT/ds control scheme static area as defined in "csconst.h".

The following constants are defined in the static area:

esc ascii escape character (note: UNIX cbreak/raw mode required to detect this),
 bs ascii backspace character,
 sp the space or blank character, also available as the constant "blk"
 qm the character "?",
 e the ascii character "e",
 o the ascii character "o",
 maxnumvars the maximum number of variables = 500, simply to set array dimensions,
 maxnumrules maximum number of rules = 1000, also determines array size,
 mnblocklen refers to the total number of blocks
 (rule groups + var defns + function defns) = 14.

The following types are declared:

byte a single byte quantity,
 intype a half byte quantity,
 foldtype for possible options (min,prod,aver,full) of *folding* together truth value
 weights, (*noprotect* is delete old valcons, *protect* is save them)
 updatetype for specifying what to do with old value/confidence pairs of variables,
 noprotect is delete old valcons, *protect* is save them,
 pktypes specifies question packet options (pkone = one at a time, pkall = all at once,
 pkhalf = more important half of the questions first),
 rwtype form generation options = (readin,writeout).

All the information stored in the static area is kept within a record of type *csstatic*. Within this record the following fields are declared:

error main error indicator, there is an extensive errortype declared to indicate
 which of multiple errors have occurred
 answers array of variables' state values,
 goodrules list of the currently viable rules,
 frmenbl a boolean array indicating whether or not a predefined frame (screen
 image) is enabled or not (to be asked); certain frames or forms are
 preconstructed, but exactly which ones should be filled in are a function of
 the state of the consultation
 backflg whether or not the current state is *go backwards*, i.e., the option to go back
 to a previous state has been requested,
 fastexit whether or not to immediately exit consultation,
 iplflg whether or not to reinitialize the session, i.e. completely reset system and
 start over
 earlyexit boolean flag, whether or not a premature exit is being taken, not exactly
 the same as immediate exit,
 exitmsg what to print out as we exit session,

rg	pointer to current rule group,
viablegoals	current # of viable goals,
expert	intnamerec of expert rulegroup,
machine	intnamerec of machine rulegroup,
stop	number of viable goals to stop with,
escexit	boolean condition which indicates ESC exit of form,
changed	boolean array of variables, indicates if a variable value was changed/updated,
unpropagable	boolean array of variables, indicates if a variable can not propagate
currform	number indicating the current form in the consultation,
maxform	number indicating the maximum form number, including dynamically constructed forms at a particular point in the consultation
lastform	number indicating the form asked in the consultation, not the same as currform-1, and useful to the "back" operation
scrn8in	file pointer to the scrn8in file, which is used to dynamically construct new forms beyond the static forms,
scrnout1	file pointer for a work file, used in dynamic form construction
scrnfil	file pointer for a work file, used in dynamic form construction
scrnout2	file pointer for a work file, used in dynamic form construction
startslot	csget-getattribute startslot, cstools global
trerr	trap function error-unused here, cstools global
dummy	dummy intnamerec parameter to cstools
debug	boolean flag indicates whether to spit out debug messages,
evalenable	boolean flag (true = turn on evaluator)
thresh	machine/expert rules confidence threshold, optional to erthresh/mrthresh
erthresh	expert rules confidence threshold
mrthresh	machine rules confidence threshold,
usemachine	boolean flag, if true use machine rules,
useexpert	boolean flag, if true use expert rules,
firsttime	used to convey 1st variable in consultation,
level	indicates one of 3 network levels: global, local, private,
backup	text file pointer to backup file to be read/written.
paraf	text file pointer to paraphraser scratch file,
noopen	boolean flag tells if any nodes open,
dspname	boolean flags, if true dump printname else dump property,
memavail	no longer used,
elapsedtime	time since last call to preport,
jobtime	total elapsed jobtime,

walltime	wallclock time at start of program,
wayback	boolean flag (if true back to beginning of a rulegroup),
waywayback	boolean flag (if true back from expert to machine rules),
approxeval	boolean flag (if true enable approximate evaluation),
packetoption	option as to number of questions to ask before sending a question, can be one, half, all,
newuser	if true, then is new user,
olduser	if true, then is old user,
confirmed	boolean flag, tells whether hypothesis is confirmed,
nqueries	number of questions asked
xintnamerec	a spare pointer, for development work,
vars	a record with the following elements: list - an array of records, one for each variable, giving following id — the variable internal "intnamerec" reference asked — boolean flag, if true variable was asked of user propagated— 0 == unpropagated, 1..n == when propagated tells whether all consequences of a variable value have been spread (number indicates which frame actually solicited the variable value) candidate— boolean flag, true == candidate for this rg freq — integer indicating lhs frequency of variable nvars — integer indicating length of variables list npropagated — integer indicating # of variables propagated
rules	a record with the following elements: list — array of records, one for each relevant rule with following name — the rule designation as an intnamerec viable — boolean flag, if true rule still viable *) nrules — integer giving length of rules list
erules	a record with rules to be evaluated after variable query with the following elements: list - tuple having intnamerecs of relevant rules cp - whether to use printname or blanks in printing nrules - number of rules in list
goal	status of goal node record, contains following items id — intnamerec representation of goal variable val — array records pertaining specific goal variable values id — intnamerec representation of value conf — real number specifying confidence of value viable — boolean flag indicating whether value is viable nvals — integer number of goal variable values nviable — total number of viable goals
imkxxxxx	network is accessed by determining the internal names of the mark types of form imkxxxxx; allows convenient access to the tuples used by the control schema; procedure plinit (section 11.4.3) determines the internal names of the necessary mark types as part of the control schema initialization by looking them up in the dictionary,

icsxxxxx	similar to imkxxxxx, but more control scheme use than general use
----------	---

11.4.2. Control Scheme Tools Package

The code in "cstools.h" was felt to be of general utility to both PLANT/cd and PLANT/ds in writing control schemes. A brief description of each procedure follows. Note that each routine takes a pointer to a csstatic area as its initial argument.

csherror	This procedure is used to handle error conditions. Given an errortype and msgstr, it displays the error to the user and puts a dump in the message file.
cseqpname	Given 2 nodes, this function returns true if the printnames of the two nodes are equal.
csgetpname	This procedure gets the printname of a node. notice that it uses dynamic dimensions for the array that is holding the print name.
csput	Given a tuple and tuplelength, this procedure puts it in the network, however first checks to see if there is a tuple with the same left context up to a specified tuplelength (possibly of shorter length than the whole tuple). If there is one it deletes it, then adds tuple. Thus, this procedure can also replace.
csmakename	This procedure make a node with a print name in the dictionary if it is not there already.
csgetreal	This converts the network representation of a real number to the Pascal representation.
csputreal	This procedure puts the Pascal representation of a real number into network form.
csginteger	This converts the network representation of a integer number to the Pascal representation.
csputinteger	This procedure puts the Pascal representation of a integer number into network form.
csrealp	This function is used to check to see if the node represents a real value, generating real value if it is.
csintegerp	This function returns true if testnode represents an integer, generating integer value if it is.
csopen	This procedure is used to open nodes under a restricted protocall. This is meant to be used to mark the fact that a node should not go away through several closely placed (in time) calls to tuple manager routines that use the node. This assumes that a use counter is now implemented for nodes. One further note: most of the routines in cstools are node-open-or-close-status preserving if a use counter is implemented. Thus to keep the use count above 1 to state that a node will be around for a while, then do a csopen on the node at the beginning of transactions on the node, then do a csclose at the end of the transactions. This procedure and its companion are here for efficiency reasons.

csclose	This is the companion routine to csopen. It decrements open count.
cshasprop	This boolean function is used to check on the existence of a property (intnamerec) under a node.
csputvalcon	This procedure is used to store value (intnamerec), confidence (intnamerec) pairs under the variable, vari (intnamerec). It calls in turn the system wide procedure to do this. Assumes updatetype = { protect,noprotect}, i.e., how to handle previous values, is defined on top of this procedure.
csgetvalcon	This procedure is used to get the tuple "duples" containing value/confidence pairs under a node. The returned tuple has its 1st elt the "node" intnamerec, its 2nd elt the "mkvalcon" intnamerec, and the following elts will be the value/confidence pairs. The boolean argument "novals" indicates whether no value/confidence pairs were found.
csdumptext	This is used to dump the text of a property under a node out to a textfile (to be later displayed on the screen).
csgetint	Given a marktype, this procedure returns its internal name by looking in the dictionary. If it isn't in the dictionary, a new node is created.
csrevaluate	Given a rule (intnamerec) to evaluate, this routine sends the necessary information to the evaluator and returns the rules truthvalue and a boolean flag indicating whether evaluation completed (e.g., whether all necessary variable values were present). If the rule lhs has been completely evaluated, complete is set to true.
csresults	Prints to a file the confidence values of a variable which is assumed to be a goal variable. If no values are viable, a boolean flag is set.
csgetlabdata	This procedure is used to get a list of labdata variables for the ruleset (intnamerec). The tuple of labdata variables are asked for at the beginning of trying a rule group.
csget	Gets an attribute tuple under main node c1 by matching context (c1 c2 c3 c4). note that level is global to this and other procedures in this utility package. Accnotfound flag specifies whether to error out if not found.
csstrengthofire	Given a confidence from the evaluator, this routine calculates the strength to fire the rhs from alpha. Four schemes for folding the confidence with alpha are available : 1) min (minimum of confidence and alpha) (default) 2) product (confidence * alpha) 3) average ((confidence + alpha) / 2.0) 4) full (alpha).
csinterpret	Given a rule(intnamerec), its truthvalue and foldscheme. this routine invokes the interpreter with the necessary information after calculating its strength to fire.

11.4.3. PLANT/ds Tools Package

The PLANT/ds system was originally encoded simply in terms of the appropriate "cstools" constructs and a set of procedures and functions. Subsequently, these were modified to be of more general value in generating "PLANT/ds-like" systems. This package is referred to as "pltools" and is summarized here.

plgetstr	Gets a msgstr from file or terminal and determine length as integer value.
pldumptext	Dumps property under node to standard output.
plemptyvar	Clears avar (intnamerec) of all its valcons.
plrankresults	Sorts and prints results (i.e., goal variable value/confidence pairs) in descending order according to confidence.
pldispname	Puts inames (intnamerec) printname in iofile and returns its integer length.
plreport	Reports msgstr on some current job statistics.
plputbanner	Puts advise banner on screen.
plpage1	Displays a PLANT/ds specific introductory panel on screen.
pusers	Asks the user his name to check if he is a privileged user, checking against the names file for number of times he has previously used the system (>5?). Update in nnames by incrementing his entry there.
plpage2	If a privileged user, various options can be set as to the control scheme. This is meant to follow a check for privileged users (pusers).
plresdump	Dump to msg the results data structure along with note msgstr.
plgrabresults	Assumes rule (intnamerec) has fired; looks under the goal node for results of firing and stores the results.
plresults	Print results of plant consultation. Print indicated diseases (within .2 of max confidence and > ethresh). Also print contra-indicated diseases (diseases ruled out).
plputdomain	Outputs the variable name, its domain and provides selection of items for each domain element. Also returns the domain tuple and its length: for integer-domain[1] - start of domain, domain[2] - end of domain domlength - negative of # of values in domain for real-domain[1] - start of domain, domain[2] - end of domain domlength - 0 *)
plparaf	Paraphrases item (intnamerec) which is a rule or rule subexpression and places text on the screen. It handles paging through text which is too big to fit on one screen.
plreqrule	Prompts the user for a rule option request, i.e., asks user which rule he wants to see, then calls <i>plparaf</i> .
plreirules	Prepares display of all rules relevant to a variable (intnamerec).
plstatusofgoal	Copies all viable rules in the goal record onto a screen frame. If boolean flag pviable = true then prints viable goals, otherwise prints non-viable goals.
plgoalrule	Returns true if rule (intnamerec) has a rhs which contains a goal.
plisgoalviable	Check whether goal value of rule (intnamerec) is amongst the viable candidates.
plgetrhsval	Given a rule (intnamerec) and the goal variable (intnamerec) gets the internal name of the reference in the first selector of its rhs. If the selector found is not a goal selector plgetrhsval returns false.
plrulesdump	Dump to msg file the rules data structure, along with note msgstr.

plerulesdump	Dump to msg file the erules data structure, along with note msgstr.
pigetvars	Gets a lhs variable list for a rulegroup designated by rg in the control scheme static area and copies the information into the vars data structure.
plcandidates	Marks as candidates for network propagation every variable in vars that occurs in the rulegroup's lhs.
plgetrules	Loads the rules data structure with the rules in the current group (rg) and initializes that data structure.
plisruleviable	Returns plisruleviable = true if the rule (intnamerec) passed is currently viable.
plruleviable	Marks the rule (intnamerec) as no longer viable: if settrue = true then rule set to viable else rule set to not viable.
plgoalviable	Marks the goal value sought by specified rule (intnamerec) as no longer viable: if settrue = true then goal is set to viable else goal is set to not viable.
plvarunknown	Returns true if their are no values under variable avar (intnamerec) or its value is unknown.
plvarselect	Select a variable avar (intnamerec) that has yet to be propagated through the network. Use precedence: 1) select var whose value is known but not yet propagated(a rhs non-goal variable), 2) select the first biased variable encountered, 3) select the variable with the max lhs frequency. If avar has no value, askforit is set to true. If all the variables have been propagated, found is set to false. loc returns a pointer in the var vector to the variable selected.
plmarkpropagated	The propagated flag of avar (intnamerec) is set to mark.
plclearerules	Clear the erules data structure.
pimatchrules	Finds all the rules in rulegroup rg that have avar in their lhs's and places them in the erules data structure. It will only insert those rules that are not already there.
piselectrule	Select next rule from erules data structure. If none is available, found is set to false.
plibuildfreq	For each candidate, un-propagated variable in the vars record sets its frequency field to an integer value corresponding to the number of viable rules' lhs's in which it occurs.
pismstats	Display some job statistics.
pismfailed	Given a rule node and the location of its corresponding location in the goals list, displays all failed selectors.
plfailed	Prompts the user for the goal rule he wishes to examine, then calls plsmfailed to display it.
plsumm	Displays program summary information.
plinit	Initializes system, reading in backup file backupfn.
plgoalinit	Initializes the goal structure.

plexprep	Prepare expert rules for use.
plhypelim	Displays a frame listing all eliminated hypotheses since the last call. Also provides several user options similar to <i>plgetdomain</i> .
plgetdomain	Given a variable <i>varcurrent</i> , retrieves a value and confidence pair for it from the user. If <i>back</i> is true the user requested going back to a previous question.
plwhy	User option that looks at all the possible answers to the current question and reports to the user how they would effect the status of the goal.
plvardump	Dump to msg file the vars data values.
plevalall	Evaluate all the rules in the rules list using the threshold <i>thresh</i> regardless of current viability.
plcloseout	Perform various closing tasks.

CHAPTER 12

The PLANT/cd System: Rule Based Inference Control

12.1. Introduction

This chapter describes the backward chaining control scheme used with the PLANT/cd knowledge base. The program was written by Albert Boulanger.

12.2. User Description

The PLANT/cd control scheme was patterned after the EMYCIN [vanMelle, 1979] view of control. In addition to this basic control, PLANT/cd features antecedent rules (limited forward chaining), "labdata" variables (variables that are marked to be asked before any other variable), and multiple goals for a rule group.

This control scheme makes use of properties. Properties are pieces of text a rule or variable can possess. These are very much like LISP properties; i.e. they have a type and contents. (See description of the parser.) The control scheme uses the PROMPT property of a variable to ask for the value. The TRANS property of a variable is used to display the value(s) of the variable. This property is a better description of the variable than the variable's name.

ANTECEDENT rules allow for limited forward chaining in a backward chaining mechanism. Whenever a variable is asked for, or it is updated by the execution of a RHS (right hand side), all ANTECEDENT rules that refer to that variable on their LHS (left hand side) are tried.

GOAL variables are variables whose value/confidence pairs are displayed to the user at the end of a session. They usually represent the entities that the user seeks advice on. This present PLANT/cd control scheme allows for more than one GOAL variable.

LABDATA variables are variables that are used often and usually represent basic information. LABDATA variables are sought out (either by asking or inferring) before GOAL variables. Unlike EMYCIN, in which such variables were always asked, PLANT/cd allows these variables to be inferred.

12.3. Control Scheme Details

The most important parts of the backward chaining control scheme, a program called BWARD, are illustrated in flow chart form in Figures 36, 37 and 38. The called procedures are in capital letters in these figures. The procedure names are idealized a bit since the actual names start with the module 2-letter prefix BC to distinguish them from other ADVISE modules. Also many of the low level procedures are in a set of utility procedures, called CSTOOLS (see Control Scheme Tools Package in previous chapter, that is shared between PLANT/cd and PLANT/ds. These procedures are prefixed with CS.

The top level (Figure 38) contains a loop that enables multiple consultations to be performed within one session. The items that need to be initialized once across several consultations are initialized in procedure START. The initialization that is needed before each

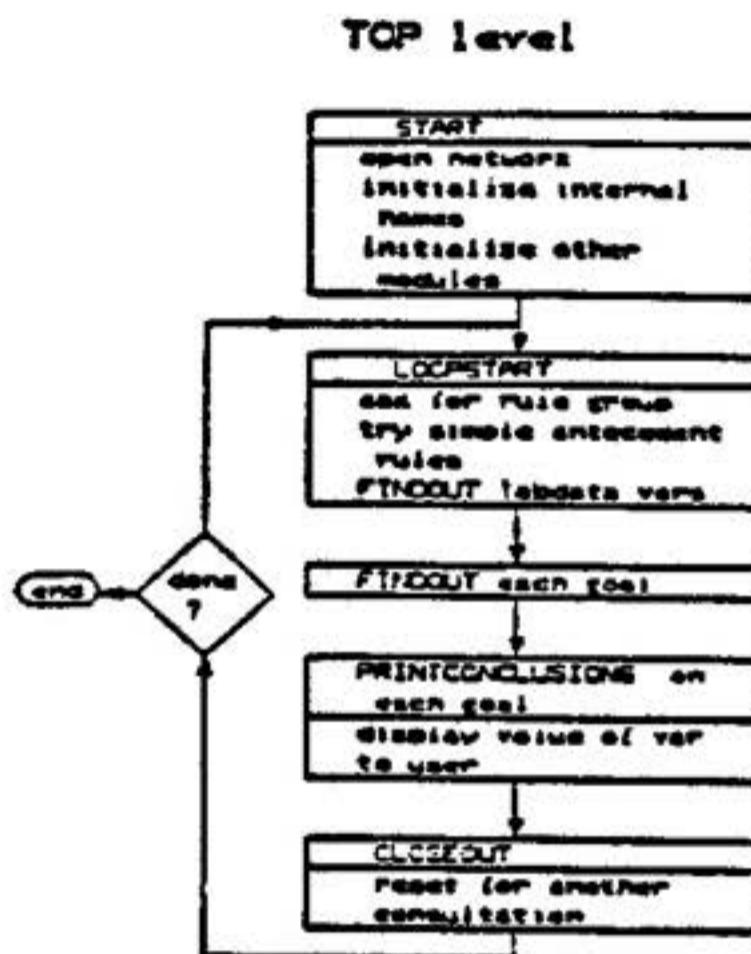


Figure 36: Flowchart depicting the top-level structure of PLANT/cd.

consultation is performed in procedure LOOPSTART. In LOOPSTART, the rule group (the knowledge base) name is requested, the GOAL variables for this rule group placed on a list, a list of ANTECEDENT rules obtained, a list of "simple" ANTECEDENT rules obtained, and the list of LABDATA variables retrieved from the network. The simple ANTECEDENT rules are next tried. Finally, values for each of the LABDATA variables are searched for using the FINDOUT procedure described below.

After this initialization, each of the GOAL variables is assigned a value using the FINDOUT procedure. This is the main part of the consultation. After the GOAL variable values are obtained, they are printed with their confidences using procedure PRINTCONCLUSIONS. The consultation is ended by cleaning up for a new consultation and calling LOOPSTART again.

The FINDOUT procedure (Figure 37) is responsible for finding the value of a variable: either by asking for it, or by using rules to infer it, or both. It first uses the ASKFIRST function to check if the current variable has the ASKFIRST property.

If it does have the ASKFIRST property, then procedure ASKFORIT is called to get the value from the user. After obtaining the value from the user, procedure CHKANTE is called to evaluate and possibly fire any ANTECEDENT rules that has the variable in their LHS. The SHOULDINFER function is then called to see if the variable also needs to be inferred.

The decision to infer a value involves several considerations. In the most general case, there is the value/confidence pair requested from the user and there is the value/confidence pair inferred from the rules. The process of resolving any differences in these value/confidence pairs

FINDOUT procedure

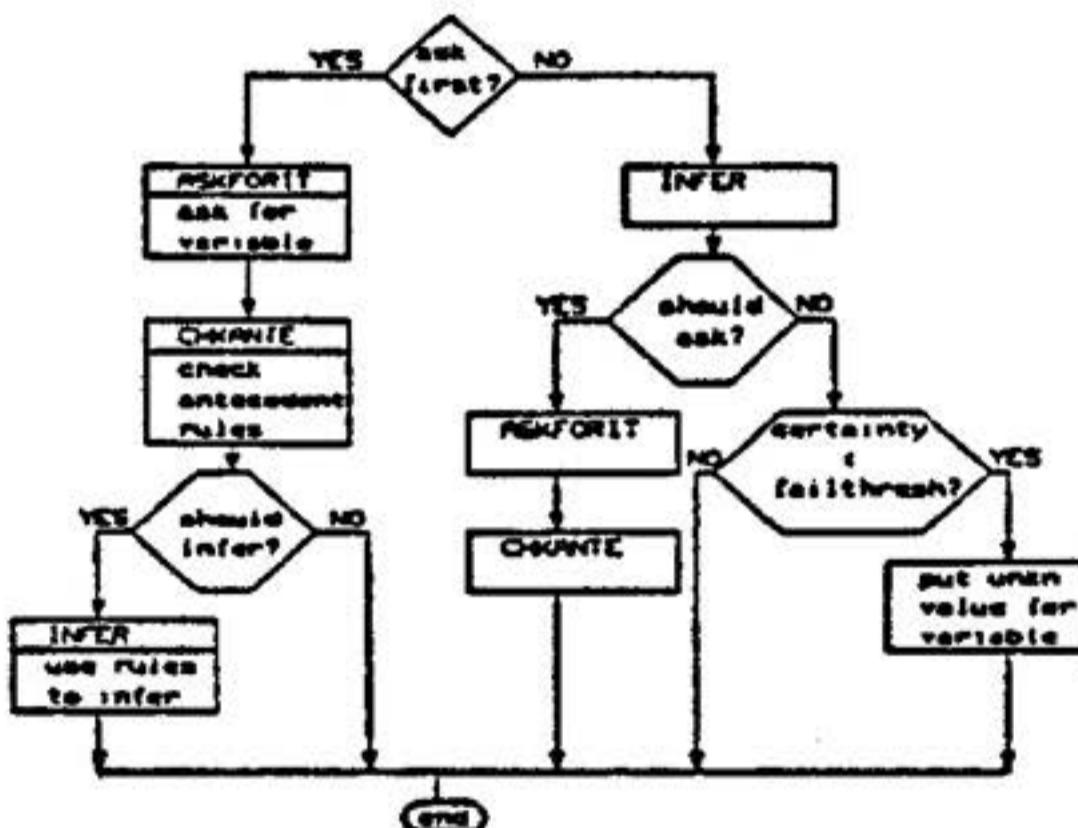


Figure 37: Flowchart depicting the FINDOUT procedure.

will be called voting. There is a scalar PASCAL variable, VOTESCHEME, that indicates the voting method. The method used in this control scheme is to replace the current value/confidence with the more current one. In the general case, the decision to infer after asking for the variable is a function of the voting scheme used. In the case of this control scheme, the variable is inferred after asking for it if the current value/confidences are below SATISFYTHRESHOLD.

If the current variable should not be asked first, then the variable is first inferred. The SHOULDASK function is called to see if ASKFORIT needs to be called. To be asked, the variable has to have a PROMPT property. Also, the maximum certainty has to be less than SATISFYTHRESHOLD. If the variable should be asked, then ASKFORIT as well as CHKANTE is called. If the variable should not be asked and the maximum certainty is below the FAILTHRESHOLD, then the value UNKNOWN is assigned.

The INFER procedure (Figure 38) uses the rules to determine the value of a variable. INFER first gets the list of rules whose RHS update the current variable. This is done by procedure GETRULES. Next, RANKRULES is called to rank this list. This procedure is currently empty. At this point, a loop is set up to try the rules in the above list. SHOULDCONTINUE is called to check if the maximum certainty of the value/confidences for the current variable is above SATISFYTHRESHOLD. This is one terminating condition for the loop; the other is when all rules have been tried.

Inside the loop, the list of variables used in the LHS of the current rule is obtained by calling GETVARLIST. This list is ranked in procedure RANKVARS. A simple rule is one that

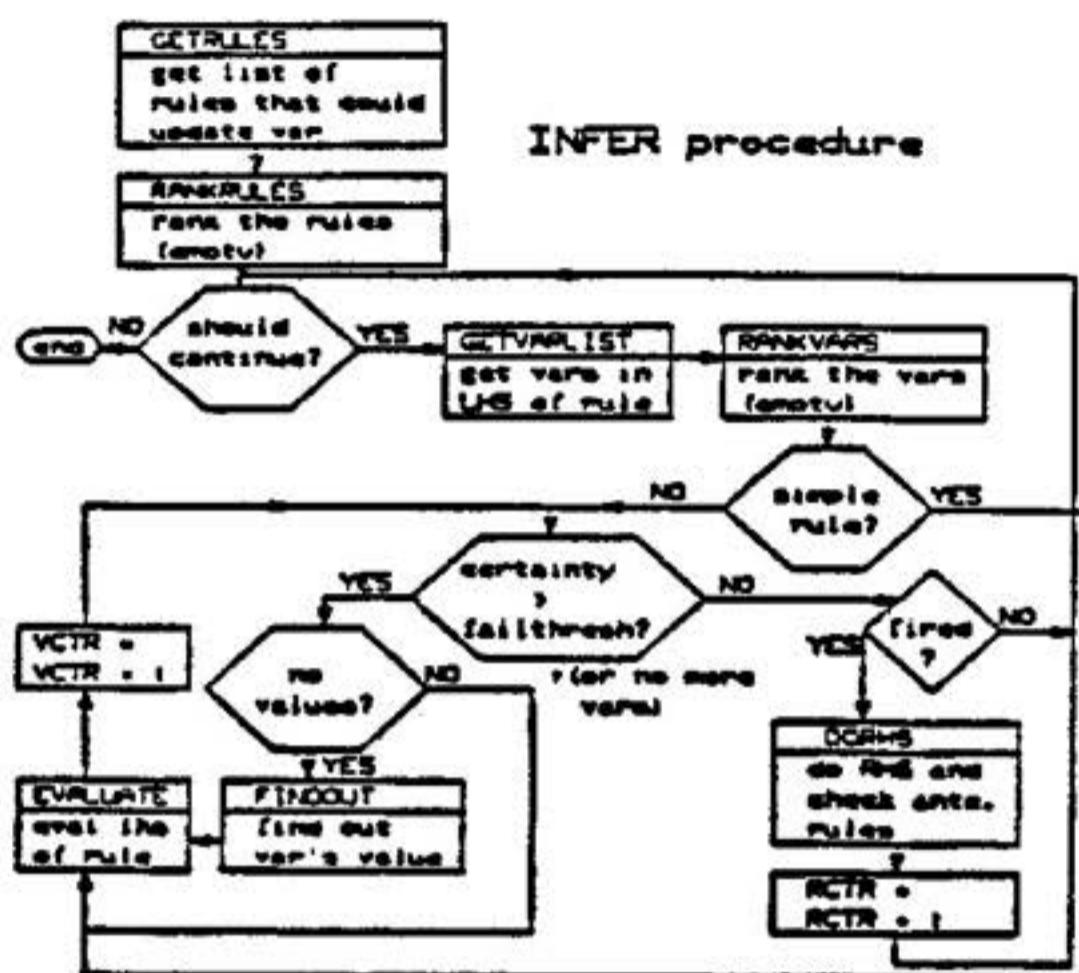


Figure 38: Flowchart depicting the INFER procedure.

does not use any variables in its LHS. There is a check after getting the list of variables for the current rule whether the rule is a simple rule. If it is, then the LHS is evaluated and the rule is checked to see if it fired. For rules that are not simple, then another loop is set up to use the FINDOUT procedure on each of the variables in the LHS variable list. For each variable value determined, the LHS is reevaluated to see if the certainty of the LHS falls below FAILTHRESHOLD. If it does, the loop is terminated, and the rule will not fire. The other terminating condition for the loop is having run out of variables. Upon exiting the loop, the rule is checked to see if it fired, using the function FIRED. If the certainty of the LHS is greater than FIRETHRESHOLD, then DORHS is called to execute the RHS. Since executing the RHS may update variables, CHECKANTE is also called on all the variables that are updated in the RHS of the current rule.

12.4. Future Research Goals

There are a number of improvements that might augment the power of the present version of the control scheme:

- Self referencing rules.
- EMYCIN-style contexts and between-context reference.
- Variable blocks. These are groups of variables that can be requested in a group. This could be done by setting up an empty table for the user to fill in, as in EMYCIN, or by

- multiple display module blocks on the same screen.
- Consultation transcript stored in a file.
- The ability to save and reload prior consultations.

CHAPTER 13

The BABY System: Context Driven Data Annotation

13.1. Introduction

This chapter describes BABY, an expert system to aid clinicians who manage patients in a Newborn Intensive Care Unit (NICU). BABY was developed and partially implemented by Lance E. Rodewald. Throughout this chapter, 'user' refers to a clinician using BABY as a monitoring aid; 'expert' refers to a person designing the medical knowledge base; and 'programmer' refers to a writer of expert system software.

Overview of the Clinical Environment. BABY's task is to find clinically important patterns in the medical and demographic data about NICU patients. It is targeted specifically on the NICU for two reasons—there is a need for a system like BABY, and the chance for success is good due to peculiarities of neonatology. There are few areas in medicine where the amount of data, especially numeric data, is as great as in a NICU. The volume of information to be comprehended can be great enough to overwhelm the clinician, and much data is either within normal limits or changing slowly, creating the potential for a decreased index of suspicion of new findings. In contrast with adult medicine, the relative importance of the monitoring data to physical exam findings is greater because the babies often do not demonstrate obvious physical signs with serious disease. The vast majority of data can be made available on-line for a computer system [Frayer, 1980][La Gamma, 1980]. Machines already monitor many physiologic parameters and have been used for extraction of significant events from the stream of signals coming from monitoring equipment [Freedman, 1979]. Additionally, the past medical history of a newborn is much more concise than that of an adult. Because the number of diseases is limited, the amount of medical knowledge needed for interpretation of findings is reduced.

Overview of System Function The design philosophy behind BABY is that the system should metaphorically act as a neonatologist observing all on-line data in the nursery, keeping track of the clinical states of the patients, suggesting further evaluation for important findings, and answering questions about the patients. This places BABY in the relatively benign role of the observer, who answers questions, rather than that of the attending physician, who asks the questions.

BABY, although it uses a knowledge base, differs significantly from other medical expert systems in that mandatory human interaction is de-emphasized. This departure from the norm is done for two reasons. Most important, it was felt that the expert system, to have a good chance for success, should integrate itself into the daily routine instead of forcing a change in this routine. The second reason is that there are many inferences to be made from the automatically collected data that are either not attempted, or not done well, by current laboratory computer systems. In other words, there is an environmental niche for an expert system in the NICU to intelligently annotate the incoming data.

The role of BABY forces several requirements on the expert system structure. First, the man-machine dialogue must be primarily initiated by the user. This, in turn, requires that the workings and current state of the machine be made as transparent as possible by the user interface. In particular, the system's current impression of the patient must be explicit and accessible, as should the means for inferring that assessment—the lab values and knowledge

structures.

The automatic nature of the non-user input means that the system must be able to interpret information not specifically requested. To do this accurately the machine must assess the data in the context of global knowledge about the patient. A computer representation of the patient's state which is capable of conditioning the input interpretation is therefore needed. BABY represents clinical states as prototypic data patterns that are matched against current information in the patient's database. Embedded in these patterns is information indicating the clinical context required for an appropriate match so that interpretation will not be attempted with a pattern that is out of context.

Having BABY suggest further work-up to clarify problems identified in the automatic data implies that the system be able to identify the tests most likely to supply valuable information. Ideally, this ability should be derived from the machine's assessment of the patient's condition in order to use global knowledge for test selection. Pattern prototypes that can match partial or incomplete data and signal to the user the missing information provide BABY's solution to the problem.

Overview of System Structure. There were two guiding forces in the BABY design. Most important, the system was designed from the user interface inward to maintain the desired type of man-machine interaction. Second, it was integrated into the ADVISE system in order to use the knowledge engineering tools developed at the University of Illinois.

The previously mentioned metaphor for the system poses a number of engineering problems. A knowledge representation must allow data-driven interpretation; the control scheme has to be semi-autonomous; and the representation of the patient state must be made available to the user interface operators. The resulting BABY structure is shown in Figure 39. The patient state and user interface are central to the system since the other modules interact through them. The ADVISE system provides the software environment for both system development and operation.

BABY needs to interface with at least the hospital information system to obtain the lab values and demographic information. In addition, if monitoring input is to be used, the stream of raw data must be preprocessed to extract significant events. In other words, BABY is not a machine for data storage or signal enhancement; the existence of that capability is assumed. Rather, BABY is a knowledgeable interface between clinician and NICU computers that makes sense of the patient's data by putting it into a clinical context.

13.2. Clinical Perspective

As an aid to the neonatologist, BABY aims to help with the tedious tasks rather than those in which the clinician has the most individualistic approach and takes the most pride. Therefore, it was felt that BABY should intelligently follow the data about the patients and not concentrate on the differential diagnosis.

13.2.1. Role of BABY in Diagnosis

In their description of clinical problem solving, Eddy and Clanton identify four main tasks [Eddy, 1982]. They are: selection of a pivot, generation of a cause list, selection of a diagnosis, and validation of the selection. A pivot is a pathophysiologic state around which a differential diagnosis can be developed. BABY concentrates on the first task—it is strictly concerned with finding pivots in the data. Selection, pruning, and validation are left to the clinician.

Sidestepping the medical diagnostic process has at least two advantages. Most important, false positives become acceptable because the burden of diagnostic proof remains on the physician. Having BABY provide only pivots also has the advantage of simplifying system architecture and knowledge base construction. The complexity and size of

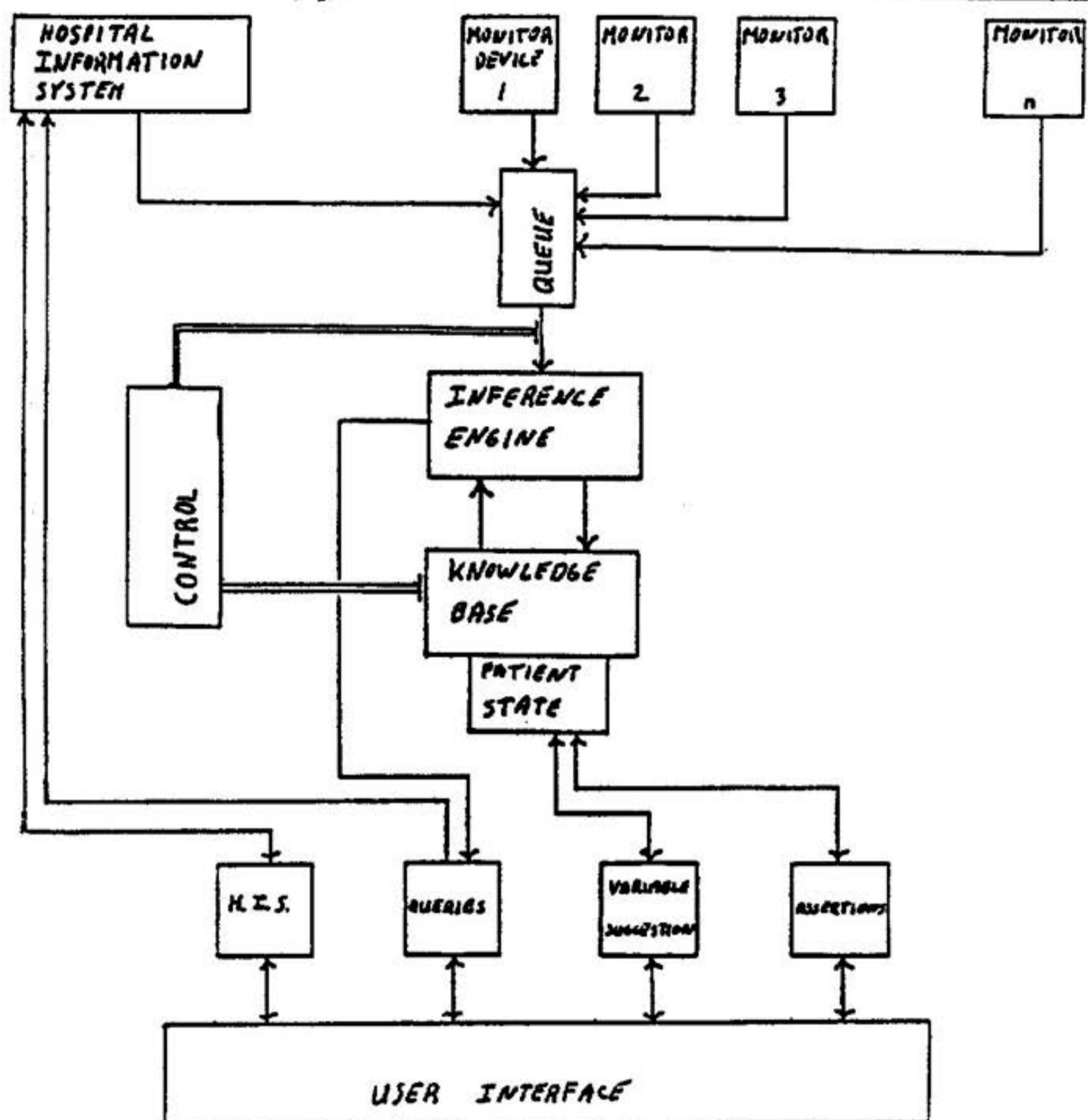


Figure 39: System Structure of BABY

CADUCEUS's knowledge base [Pople, 1982] testifies to the difficulty of computer differential diagnosis [Pople, 1982].

13.2.2. Clinical Patterns

There are three pattern types that need to be distinguished:

- patterns annotating a single variable,
- patterns composed of more than one variable or subpattern, and
- patterns that assign risk value. The risk assessment pattern is a special case of patterns composed of other patterns, but its clinical usage is quite different.

Variable Annotation Patterns. Given a laboratory value or significant event from monitor output and the clinical state of the patient, an expert system should be able to decide if the value is low, normal, or high. Determination of the reference range for a single variable depends on demographic and pathophysiologic parameters of the patient. These parameters, embodied in the patient state, drive the variable annotation. For example, the relation of the serum sodium level to an appropriate normal reference range depends on the state of renal maturation, which itself depends on the age and length of gestation.

In order for a system to be accurate in its annotation, global interpretation of the variables is needed. In contrast, most computer systems that indicate a lab value's location in a reference range use a single range for all patients, regardless of other values known about the patient. Often the lab systems do not have demographic and clinical information with which to make inferences, and without a large amount of patient information any attempt at accurate annotation is bound to fall short. Neonatologists notice this problem often because adult normal values differ so markedly from newborn normal values. After some experience with inaccurate annotation, the clinicians tend to look only at the values while ignoring the computer's assessment of those values.

Global interpretation puts a large demand on the accuracy of the machine version of the patient's clinical state. If annotation could be done well, there would be much value to it. However, the problems of representing clinical information in machinable form is quite difficult. BABY approaches the problem of global interpretation by having appropriate pathophysiologic states condition the interpretation of incoming data. The portion of the knowledge base to use in annotating the data depends on the state of the patient.

Combination Patterns. Different lab values or monitoring events that must occur together to signify a finding comprise patterns of the second type. These groupings of data usually indicate pathophysiologic phenomenon in the patient. An example is the finding of metabolic acidosis, which itself depends on more primitive patterns such as acidosis, hypocapnia or normocapnia, and hypoxemia. The distinction between data annotation patterns and combination patterns is that one should always be able to annotate a variable, while the presence of the latter type of pattern is less predictable. The combination patterns to expect depends on the clinical situation.

Risk Assessment Patterns. While the above combination patterns represent pathophysiologic states, another type of combination pattern could be used to assign a degree of risk for the occurrence of an event or complication of therapy. Although they do not strictly represent pathology, it is desirable to identify these patterns if they are evident in the automatically collected data. As an example, consider a data pattern that tries to predict the risk of an iatrogenic pneumothorax in a ventilator-dependent patient. Such a pattern would need data about the respiratory pressures and rate, the patient's disease, his age, a measure of the lung compliance and its trend of change, the breathing pattern (fighting the ventilator?), and a history of prior pneumothoraces. A monitoring system that could accurately assess the risk in different patients might be able to prevent this catastrophic occurrence by raising the clinician's index of suspicion. This particular pattern probably could be built and tested in the clinical setting.

13.3. Clinical Context and Patient State

The clinical context within which the data is gathered presents a problem. Few values can be interpreted in isolation of other findings or parameter measures. Consider the value of the variable AGE. The actual age of the patient is really a statistic used to help assess the parameter of physiologic maturity of the patient. In a baby, knowledge of age is not enough since the length of gestation is also needed to determine the degree of physiologic maturity. Therefore, a system to analyze lab data containing variables whose reference range varies with renal maturation, for example, should be prepared to take into account the patient's physiologic age.

Although this seems straightforward enough, the problem becomes less clear when one considers how the normal reference range is defined and used. The interpretation can vary from a Gaussian distribution of lab values to an aesthetic ideal of the most perfect value for an individual [Galen, 1980]. In the former interpretation, the range is usually defined as those values falling within two standard deviations of the mean. While this is the most common derivation of "normal" and is easy to determine, it often is incorrect in a given clinical setting because heterogeneity of the target population is accounted for statistically rather than individually. More precise norms could be defined for smaller populations. This problem is seen in pediatrics where normal growth and maturation are constantly changing the normal range. In general, the creation of context-dependent normal ranges is difficult because of problems in defining the contexts. In BABY the reference ranges are defined by the knowledge base author, as are the contexts. Ideally, a system should allow clinicians to define their own contexts in order to tailor the system to meet their own specific needs.

Searching for these three pattern types within the clinical context of a patient is the primary purpose of BABY. The immune system provides a biologic analogy to the type of information processing performed by BABY. The animal's antigenic stimuli become the data, while the antibodies become the patterns to be matched against that data. Like the antibody waiting for its antigen, the pattern does nothing until a match is found in the data which then causes an action to take place. The clinical context of the patient is used to condition the activity of individual patterns in the set of possible patterns. As with the immune system, and unlike the nervous system, there is no centralized control for the patterns; it is distributed among its individual elements. In practice, of course, BABY has to simulate this decentralization because of its implementation on a non-parallel processing machine.

13.4. Knowledge Engineering Environment

13.4.1. Clinical Data

There are three sources of input to BABY: significant events extracted from monitor equipment output, information available on a hospital information system, and user input during an interactive session. BABY needs no user input at all; it will make whatever inferences it can from the automatically collected data. If a user wishes to interact with the machine, more inferences can be made with the additional information he or she supplies.

The data available in the NICU can be classified in two different ways—by function, or by collection method. Functionally, it is either automatically collected, or it is requested to answer a specific question. Distinguished by collection method, data is either obtained continuously by machine, intermittently by machine, or is observed in the process of a clinician's physical assessment. Continuously sampled physiologic indicators would need to have significant events extracted from the analog signals.

The significance of this classification is that it delimits the function and structure of BABY within the limits of the metaphor describing its purpose. Functionally, the standard

input is automatic data, while the standard output is assertions about the patient along with new data to clarify clinical findings in the data. Structurally, the standard input only excludes physical signs. Often BABY will have to make its inferences within the scope of the allowable data.

13.4.2. Uncertainty in Medical Data

Certainty of medical data decreases with time; the more time that elapses since a measure was taken makes the value less reliable. The rate of certainty decline varies considerably from test to test, and within the same test, depending on the stability of the patient. To complicate matters further, the loss of certainty, although monotonic, can be non-linear. It is desirable for the knowledge representation to express temporal uncertainty, but to do it accurately is far from easy.

For a monitoring expert system, the NICU has the advantage that data is refreshed at frequent intervals. Temporal uncertainty is reduced most by this mechanism, but by using comparisons to previous values, uncertainty due to factors intrinsic to the test or collection procedure can also be decreased. BABY deals with uncertainty in a probabilistic sense. That is, certainty (and uncertainty) are explicitly represented as probabilities of truth in an assertion. Combining different sources of uncertainty into a single probability is done with a method similar to that used by the PROSPECTOR system [Duda, 1978a].

13.4.3. The Knowledge Base

Two important aspects of a knowledge representation are its notational efficiency and expressive adequacy [Woods, 1983]. Expressive adequacy is concerned with what the representation is capable of stating about the world. Notational efficiency has to do with the ease with which it can be stated. Of the two attributes, expressive adequacy is probably the most important because it limits the potential of the representation.

The knowledge base often has a single representation format that limits its expressiveness, but makes inference engine development easier and facilitates knowledge base creation and modification. The representation method selected usually allows for some degree of domain independence. In addition, modularization is strived for in expert system design so that the component parts can be more easily modified. Therefore, control and knowledge ought to be developed separately, but in practice it is often difficult to completely remove elements of control from the knowledge base. For example, in MYCIN the antecedent ordering in a rule affects evaluation order in the backward chaining.

As mentioned before, the BABY knowledge representation consists of rules connected together by a network that guides the control scheme to a limited number of inferences. BABY has a large element of control in its representation which is expressed with context nodes that signal to the inference engine a portion of the knowledge base is usable for inferences. These context nodes act as keys, indicating that a pattern is potentially relevant and should be searched for. During knowledge base design the context nodes let the expert confine his or her reasoning to a smaller problem within the defined context, easing the creation task somewhat.

13.5. Baby Implementation

The current implementation of BABY consists of two Pascal programs situated within the ADVISE system on a Vax 11/780 computer at the University of Illinois. One program is a tool for knowledge base creation and is responsible for translating a text description of the rule network into a machinable form. The other program uses this machinable representation to interpret the NICU data and run the user interface.

At any point during runtime, BABY is either updating information, waiting for information, or interacting with a user. While updating information, the knowledge base is unstable and unavailable for user queries or input. As inferences about the patient are made, they are scrolled onto the appropriate output window where they remain until new information makes them invalid. Following knowledge updating, the variables most likely to influence the patient state are reported and the user has a chance to interact with BABY until new information is supplied by the laboratory computer or patient monitors. The result is a mixed-initiative dialogue with the user in control of the questioning phase and the machine intermittently supplying a summary of its findings.

13.5.1. User Interface

The user interface attempts to maximize the man-machine communication bandwidth, while at the same time minimizing the need for a specific communication language. Using windowing techniques and a pointing device, the number of verbs needed in the interface is kept to a minimum. Central to the user interface is a bit-mapped graphics terminal which communicates serially with the ADVISE computer. All non-numerical user input is done with a mouse pointer acting on pop-up menus.

There are five physical windows in the current implementation that correspond to four conceptual views of the system and one scratch window for interactive system output. Two windows deal with output, and two windows deal with variables.

BABY has two types of standard output—assertions about the patient state and suggestions for refinement of that state. Assertions are scrolled onto one window and any user queries concerning the patient state or list of current assertions takes place within this window. The suggested variables are scrolled onto a second window. If the user wants to supply any of these variable values, he or she would move the pointer into the window to select the proper variable. If the variable type is nominal, a pop-up menu with its domain appears and selection is again made with the mouse. The keyboard would be used only to supply numeric values.

The third window gives a view into the variables of the hospital lab system, and would be used as a regular interface with the lab. The other window gives a view into the BABY variables. Here requests can be made to see a variable's value or domain and values can be provided for variables that were not suggested by BABY.

The graphics terminal supports many primitive functions needed to manage windows, including window creation, clipping, saving contents, context switching, and pointer management. The programmer uses these procedures in routines which bind data base and knowledge base information to the windows. For example, a menu can be placed in an ADVISE textnode, which in turn can be executed by a user interface procedure. This procedure is textnode independent, and the user choice is returned as a small integer to the calling procedure where binding of the appropriate subroutine to that choice is made.

13.5.2. The Knowledge Representation

A BABY knowledge base provides a representation for a collection of prototypic data patterns that are to be searched for in the laboratory and monitoring input. Any prototype that matches an appropriate combination of data elements from the input will be asserted to the user. Each of these prototypes has a truth value representing how well the data substantiates the prototype, so an assertion is made only if the truth value is above the pattern's threshold.

The pattern prototypes of BABY wait for data to justify them in a bottom-up, data-driven manner. The pattern prototypes that are active and thus trying to match data from

the input depends on the past history of the data. This is accomplished by having only patterns present that are in clinical context; thus only appropriate patterns will be available for matching.

The prototypes have the form of directed, acyclic graphs with unique roots. The graph consists of nodes connected by directed arcs. At one end of the graph is a root that represents a clinical or pathophysiological condition and always has a current truth value, or degree of belief in that condition. At the graph's other end are leaves to act as data entry points. The BABY leaves take the form of VL1 rules [Michalski, 1974] which, at the minimum, compare a variable's value to a reference value, returning a truth value as its result.

See Figure 40 for an example of a knowledge base pattern.

The BABY rules are used as predicates for matching a condition to a truth value. The ADVISE subset of the VL1 is the language that defines both variables and rules. For a description of variables and rules, see Chapter 6. Rules are connected to the graph structure with arcs that act as conduits for propagation of truth values. The variables contained within these rules may be used in a large number of unrelated rules, creating the problem of mapping variables to their respective rules, a necessity arising from the data-driven nature of BABY. The VL1 parser creates a symbol table for this mapping that is used by the inference engine to locate the pattern leaves currently in context and available for matching.

Patterns are recursive structures composed of nodes, arcs, and rules in a manner similar to a PROSPECTOR knowledge base network [Duda, 1978a]. The information content of a pattern is summarized with a single number between zero and one, its truth value, representing the degree of confidence with which the pattern is believed to be supported by the input data. The recursive nature of the patterns arises because they can be used as intermediary nodes in other patterns, as their truth value is propagated upward to any patterns of which they are a part.

The truth values travel along arcs in the direction of leaf to root, being processed along the way at the intermediary nodes. These arcs are currently of two types, EVIDENCE and CONTEXT, having quite different semantics. Both arc types carry information which, when present, supports the truth of the parent node. However, the CONTEXT arcs have explicit control information that indicates whether or not the parent node is clinically relevant. If in context, the parent node is made a part of the active knowledge base and can be used to match input data. Otherwise it is essentially excluded from the collection of current patterns until the context changes. Thus, the CONTEXT arcs act as keys to their parent nodes, either locking them in or out. By this mechanism the size of the active knowledge base is reduced to a small subset of the global knowledge base, resulting in both storage and computational savings.

Every node in the network has a truth value ranging from zero to one, which represents both subjective probability and certainty of truth. Zero indicates certainty that the assertion is false while one indicates certainty that the assertion is true. Initially, no information about an assertion is represented by the node's prior value. As the truth values of the nodes change, increasing values represent rising certainty in the truth of the assertions, and vice versa.

There are five node types supported by the BABY inference engine. They are the logical types AND, OR, and NOT, the predicate type RULE, and a type responsible for folding evidence and uncertainty together, the type BAYESIAN. The RULE nodes are the leaves of the patterns and are evaluated by the ADVISE rule evaluator (Chapter 7) according to semantics set by the programmer. The three logical nodes have very simple semantics. The truth value of an AND node is the minimum of its children's truth values,

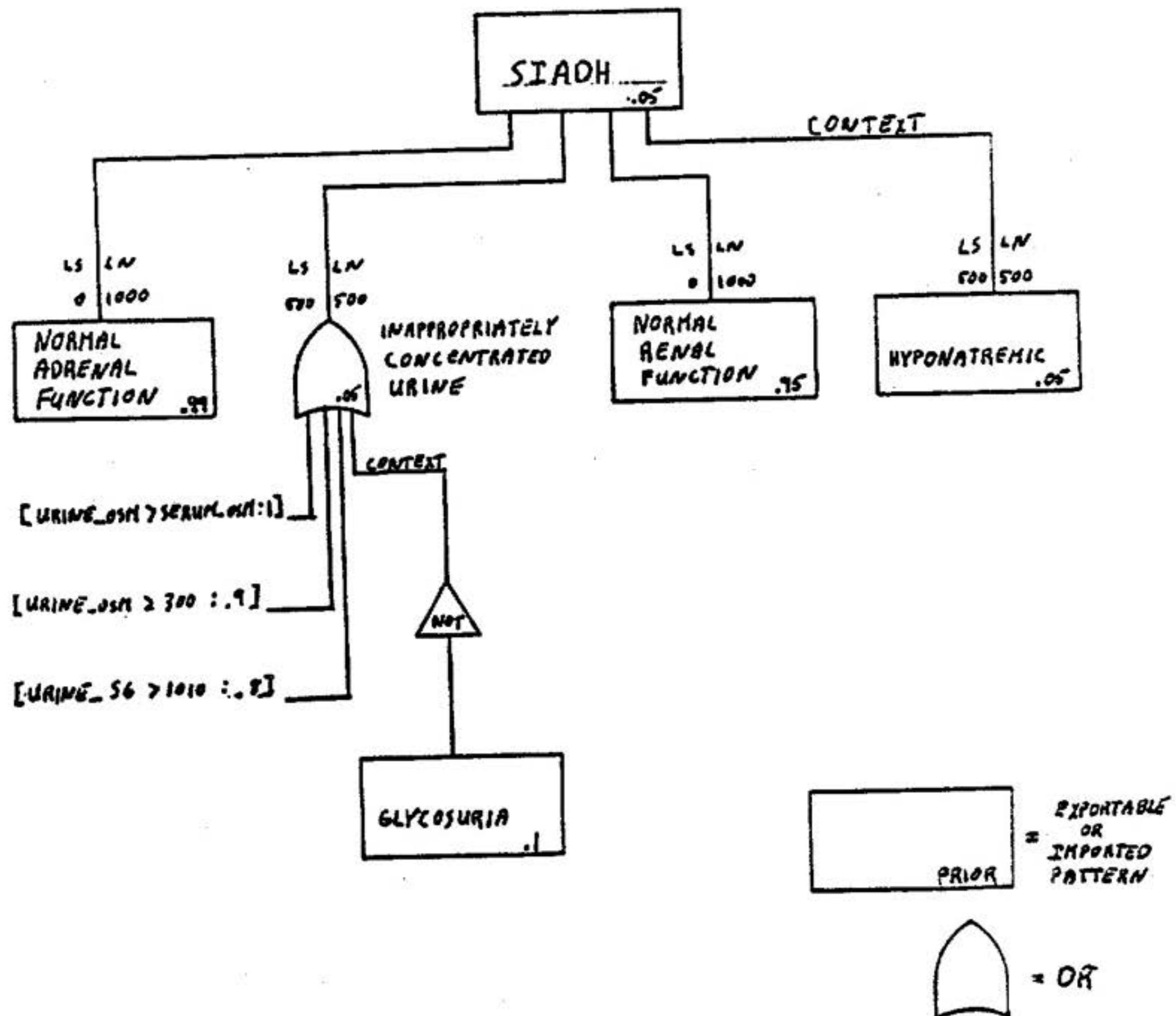


Figure 40: A BABY Knowledge Base Pattern

while an OR node takes the maximum value of its children. NOT nodes invert their child's truth value by subtracting it from the maximum truth value. A high truth value of any one child will give an OR node a high value, but all children of an AND node must have high values to cause it to also attain a high value.

The BAYESIAN node type is used to fold together independent pieces of evidence to calculate a truth value. These nodes can have an arbitrary number of children attached by EVIDENCE arcs, each with their own assertion strength that the child's evidence implies the parent's hypothesis. In addition, these child nodes must have a prior truth value that

represents the probability of the evidence being true before anything is known about the patient. Associated with each connecting arc are two numbers representing the strength of the inference from child node to parent. The first number, LS, is the strength of the inference if the current child node probability is greater than that node's prior probability. Conversely, LN represents the inferential strength when the current child node probability is less than its prior probability. An entirely equivalent method of specifying LS and LN is to specify the maximum and minimum posterior probabilities that the parent node can attain, given maximum and minimum child node current probabilities. These numbers must be supplied by the expert and he or she can use whichever form is most convenient.

The truth values of all five node types can be calculated with information local to the node, assuming that the values of its children are stable. Implied, then, is that a single node can only affect the values of nodes between it and the pattern root, limiting the number of nodes whose values need propagation when new data arrives. By locking out portions of the knowledge base from activity, nodes connected by CONTEXT arcs also help reduce the list of node values to propagate.

13.5.3. The Inference Engine

The network structure of BABY's knowledge base makes the inference engine quite simple. It is a data-driven, forward-propagating algorithm that only needs a small planning phase because of the context links. This is quite different from Prospector in that the planning is minimized, propagation is a parallel process, and the context links are central to control rather than to question planning [Duda, 1978b]. It is easiest to understand the algorithm by way of examples.

Initially assume that there are no context links in the knowledge base. When a burst of lab data comes into BABY, the appropriate system variables are bound to the lab data values. For example, when the serum sodium value comes in as 135 meq/l, BABY's variable for serum sodium is located and set to 135. Next, all rules using serum sodium in the currently active knowledge base are located via a look-up table and placed on a queue called the NODES-TO-PROPAGATE (NTP) queue. Each variable may be used by several predicates, all of which are enqueued. In other words, the serum sodium value may be used in several different patterns. The table managing the binding of variables to predicates is created by the ADVISE VL1 parser.

After the rules are enqueued, they are dequeued, evaluated, and their truth value is propagated to any reachable parent nodes. A node is reachable if there is a direct link to it and if it is in the currently active knowledge base. Propagation implies calculating the parent node's new value based on the value of the node just dequeued. After the new value is calculated, that node is enqueued onto the NTP queue. If the parent node is a pattern root, and if its new value is greater than its assertion threshold, its value is presented to the user. This cycle of dequeuing and enqueueing is repeated until the NTP queue is empty. The algorithm resembles a breadth-first graph traversal due to the use of a queue rather than a stack.

Context arcs complicate the algorithm slightly, but also add considerable potential. If, in the above example, a context link was crossed during an upward value propagation, then a check would be made to determine if either the context node's value changed from above the context threshold to below it, or if it changed from below the threshold to above it. In the former case, the parent node would then be in context, and should be added to the active knowledge base. A recursive algorithm is called that activates this new portion of the knowledge base. The situation where a piece of the knowledge base can be a part of different patterns, each with their own respective contexts, is handled by an ADVISE

primitive.

The activating algorithm activates all children of the node that is now in context. It then calls itself to activate the children's children. If one of these nodes has a context link, only that link is followed by the algorithm's recursion, otherwise all links are followed. When a leaf is reached, a check is made to see if any of the rule's variables has a value. If so, the leaf is added to the NTP queue so that the rule can be evaluated and the value propagated to the now in context area of the knowledge base. If it were deemed desirable, this is where a goal-directed, backward-chaining algorithm could be added. Then, if a variable's value was not available a check could be made to see if that value could be inferred. Currently such a capability is not available in BABY.

13.5.4. The Patient State

At any given time, the system's model of the patient is contained in the patient state: that portion of the knowledge base that is in context and therefore has been activated. Included in the definition of the patient state are all the variable value bindings and the intermediate node values. The size of the active network will be much smaller than the entire knowledge base, since only a small portion will be in context at any time. This storage economy also results in the processor time economy because inferences are not made on any non-active portion of the knowledge base. Indeed, the inference engine cannot even access non-active knowledge.

The use of reduced portions of the knowledge base for individual patients could allow the system to keep track of more than one patient at a time by providing each with their own patient state. The ADVISE system facilitates this capability by having its storage divided into three levels: global, local, and private. Each lower level retains all the connectivity of the higher levels, but only those parts of the graph requested from a higher level will be explicitly stored at the lower level. Thus, an individual patient state could reside at the private level, while the entire global knowledge base would be stored in the global level. As the patient state is expanded by coming into context, it is stored at the private level. The converse must happen when graph areas go out of context.

Two other aspects of the patient state are facilitated by ADVISE features. Since the contents of the three levels are mapped onto three independent areas of secondary storage, switching from patient to patient can be easily done by swapping disk areas at the private level while leaving the global level alone. Also, any values belonging to a particular patient are stored in the patient state, so that data security can be maintained at the file level. ADVISE also supports security at a more primitive level during run time.

13.5.5. Variable Suggestion

In BABY there is a variable suggestion phase that follows propagation of the data and activation of the knowledge base. The variables selected are listed in one of the user interface's windows to indicate to the user that knowing these values is desirable. Ideally these variables would provide the most information to the patient state, or be able to effect the greatest change in the current context. Two different algorithms for scoring the variables are implemented; one is done during knowledge base instantiation, and the other requires a separate partial patient state traversal.

The latter algorithm will be discussed first. At each node, in addition to the current value, BABY stores the minimum and maximum possible values, given the minimum and maximum values of its children. If nothing is known about a RULE node's value its maximum is one, minimum is zero, and current value is its prior value, otherwise all three values are set equal to the current value. The differential, if any, between the three values

will ripple upward during regular value propagation. The scoring algorithm makes use of all three values. A variable's score is roughly equal to the sum total of probability that could be added to or subtracted from the patient state if that variable's value were known with absolute certainty. To calculate this value the active network is traversed top down, keeping track of the difference between the maximum value and the current value, and using the assertion strength values, LS and LN, to determine the multiplier for the difference in odds. This algorithm is expensive for a large patient state.

The other scoring algorithm is much simpler, but it does not account for assertion strength. This one works during knowledge base activation upon reaching a leaf. If any of the leaf's variables has no known value its suggestion score is incremented. At the end of data propagation, all variables whose values were desired but unknown have a score. The variables that were inspected the most have the highest score, and are assumed to be the most desirable. This method, although fast, looses much of the knowledge base information; it only retains the physical layout of the graph while discarding the probabilistic elements.

In either case, the highest-scoring variables are mentioned to the user, who has the option of supplying any known values.

13.5.8. Network Parser

The creation of the knowledge base is entirely expert dependent. The BABY network parser, in conjunction with the ADVISE VL1 parser, facilitates transformation of a textfile description of a network into a machine usable form. BABY accepts any rule structure that is supported by both the VL1 parser and the rule evaluator. It is up to the expert whether to make a rule more complex in order to simplify a network, or vice versa. The programmer has the option of choosing among several different semantics for uncertainty evaluation within the predicates, extending the network expressiveness.

The network parser is a recursive descent parser that creates and links the nodes with their appropriate arc types, and calls the ADVISE parser as a subroutine to create the rule parse trees and a look-up table of rule to variable bindings. A limited syntactic check is performed also to insure that at least the newly created network structure will work in BABY. In addition, the network is printed out in an in-order, indented format so that the expert can see if the traversed prior values are in accordance with expectations.

The language accepted by the network parser is quite limited and rigid, but it is not difficult to translate from a picture of a network into the text format. If a node is to be either an assertion, or it is to be used by more than one pattern, it must be uniquely named. This name will be the one shown to the user. If a node does not need a name, the parser will supply its own unique one.

Figure 41 illustrates the mapping of the pattern of Figure 40 into the network language. Appendix C shows the corresponding mapping to the ADVISE internal storage format.

13.6. Conclusion

The BABY project presented a description and partial implementation of an expert system for assisting clinicians caring for babies in an NICU. The current prototype version of BABY is part of a learning process attempting to identify the bottlenecks and difficulties that will be encountered during implementation of various ideas. The main difficulties in BABY center around the knowledge representation of the patient state.

The expressiveness of a knowledge representation sets an upper bound on the performance quality; if a type of knowledge cannot be made available, inferences and explanations based on

```

NEONATE_PATTERNS
EVIDENCE AND NEONATE_PATTERNS
EVIDENCE BAYESIAN SIADH .05 .8
[500 -500] CONTEXT .75 BAYESIAN HYponatremic ;
[0 -1000] EVIDENCE BAYESIAN NORMAL_RENAL_FUNCTION ;
[0 -1000] EVIDENCE BAYESIAN NORMAL_ADRENAL_FUNCTION ;
[0 -1000] EVIDENCE OR *
CONTEXT .75 NOT *
EVIDENCE BAYESIAN GLYCOSURIA ;
;
EVIDENCE RULE * .1 .8 [URINE_OSM > SERUM_OSM : 0.1] :> MKNULL;
EVIDENCE RULE * .1 .8 [URINE_OSM > 300 : 0.9] :> MKNULL;
EVIDENCE RULE * .1 .8 [URINE_SG > 1.010 : 0.8] :> MKNULL;
;
;
;
```

Figure 41: Network Description of Figure 40

that knowledge cannot be performed. Temporal, spatial, physiologic, and developmental relations between variables are not representable by the current knowledge base formalism. Explanations indicating supporting evidence for an assertion still could be provided for in the future.

Explicit temporal reasoning is not possible with the current knowledge representation. Trends and predictions of variable values are not calculated, even though a value is represented as a domain element and a time stamp. It was felt that rather than implement an ad hoc mechanism for handling time relationships, it would be better to develop a more complete theory of medical, temporal reasoning. Such a theory would have to include prediction of future values based on not only past trends, but also on the clinical context of the patient.

Although there is work to be done in improving the knowledge representation, the functionality of the representation should probably be preserved since it meets the data flow requirements of an NICU. The main improvement required of the knowledge base is the addition of semantics for handling temporal relations within the clinical context of individual patients. Prediction of future values and decrease in value confidence need explicit representation so that conclusions can be stated as accurately as possible, but without overstatement. Additionally, tabular representation and a graphics network editor are needed for better notational efficiency.

CHAPTER 14

The ALFALFA System: Entomological Pest Identification

14.1. Introduction

The ALFALFA field pest identification system was developed to exercise various control scheme features in an expert system which actually requires more than one rule group with more than one control scheme. The system is the result of a collaborated effort among entomologists William Lamp and Lane Smith of the University of Illinois, and Jiarong Hong and Cari Uhrik of the University of Illinois Department of Computer Science.

An automated key to insects should use descriptors which are easily observed, remembered and reported. Further, the key should proceed through a layered series of identifications, becoming more specific as more information is provided. The identification takes the form of the classical Linnean system at the higher levels (Class, Order, Family), but as often as not, the exact species is not critical and a common name is acceptably assigned (e.g., "thrips"). Incomplete data should result in, rather than no identification, a more general identification. Thus, the key targets different levels of identification using different sets of attributes for each stage of identification (Figure 42). Conceptually, different groups of rules apply to the various levels, each group possibly claiming its own uniquely important variables, pertinent to a particular stage of the consultation. Moreover, the rule group in use at a particular moment often requires its own special variation of a control scheme. Thus, one can see the inherent needs of the ALFALFA System for multiple rule groups, multiple control schemes, and dynamically changing goals.

The desired approach to implementing the system called for minimal changes to existing modules. Modules changed are the parser and control scheme. Certain features of the rule evaluator were changed to be consistent with previous usage but allowed additional features. For example, the ?, *, \$ patterns for matching various flavors of *unknown* values were not implemented consistently in the previous Rule Evaluator.

The general system development goals may be grouped according to the following:

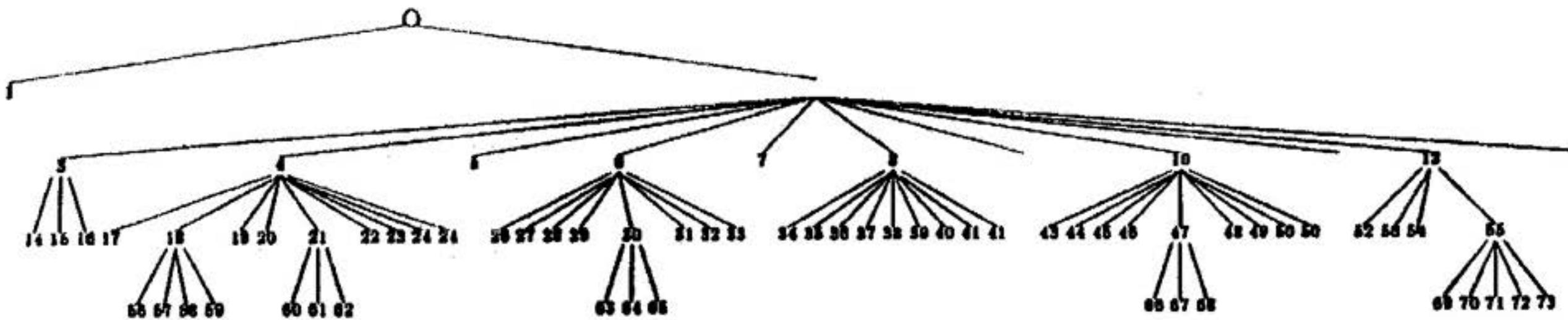
- 1) Preconditions and termination conditions on rule groups,
- 2) Multiple non-homogeneous goal values or goal variables,
- 3) "OR" in the RHS of Rules,
- 4) Enhanced Data Acquisition features .

A description of each of these areas follows and is the focus of this chapter.

14.2. Preconditions and Termination Conditions

Prior to the ALFALFA system, the only attributes of rule groups were variables blocks, functions blocks, and the rules themselves. There was a need for making explicit the various control information implicit in the PLANT/ds and PLANT/cd control scheme. This was especially required to enforce some sort of sequencing discipline among the rule groups representing the different levels of pest identification.

Figure 42 : Goal Structure of the ALFALFA System.



g1 Id = Spiders_and_Mites
 g2 Class = Insecta
 g3 Order = Neuroptera
 g4 Order = Coleoptera
 g5 Order = Collombola
 g6 Order = Homoptera
 g7 Order = Diptera , Id = Flies
 g8 Order = Lepidoptera
 g9 Order = Odonata, Id = Damself_Flies
 g10 Order = Hemiptera
 g11 Order = Thysanoptera, Id = Thrips
 g12 Order = Orthoptera
 g13 Order = Hymenoptera, Id = Bees_&_Wasps
 g14 Family = Chrysopidae, Id = Green_Jacewings
 g15 Family = Hemerobidae, Id = Brown_Jacewings
 g16 Generic_id = Lacewings
 g17 Family = Coccinellidae, Id = Lady_bug
 g18 Family = Curculionidae, Id = Weevils
 g19 Family = Siphidae, Id = Carrion_beetles
 g20 Family = Staphylinidae, Id = Rove_beetles
 g21 Family = Meloidae
 g22 Family = Lampyridae, Id = Lightning_bug
 g23 Family = Cantharidae, Id = Soldier_beetle
 g24 Family = Elateridae, Id = Click_beetle

g25 Generic_id = Beetles
 g26 Family = Coreidae, Id = Meadow_spittle_bugs
 g27 Family = Delphacidae, Id = Other_plant_hoppers
 g28 Family = Aphididae, Id = Pea_aphids
 g29 Family = Aphididae, Id = Spotted_alfalfa_aphids
 g30 Family = Cicadellidae, Id = Leaf_hopper
 g31 Family = Aphididae, Id = Blue_alfalfa_aphids
 g32 Family = Membracidae, Id = Treehoppers
 g33 Generic_id = Aphids_Leafhoppers_and_spittle_bugs
 g34 Family = Pieridae, Id = Alfalfa_caterpillar
 g35 Family = Nymphalidae, Id = Military
 g36 Family = Nymphalidae, Id = Buckeye
 g37 Family = Nymphalidae, Id = Vanessa_atalanta
 g38 Family = Nymphalidae, Id = Morning_cloak
 g39 Family = Danaidae, Id = Monarch
 g40 Family = Noctuidae, Id = Cutworms_and_relatives
 g41 Family = Pyralidae, Id = Webworms_and_relatives
 g42 Generic_id = Butterflies_and_moths
 g43 Family = Anthocoridae, Id = Minnie_pirate_bug
 g44 Family = Miridae, Id = Plant_bugs
 g45 Family = Reduviidae, Id = Assassin_bug
 g46 Family = Nabidae, Id = Damself_bugs
 g47 Family = Lygaeidae
 g48 Family = Berytidae, Id = Stilt_bugs

g49 Family = Corixidae, Id = Leaf_footed_bugs
 g50 Family = Pentatomidae, Id = Stink_bugs
 g51 Generic_id = True_bugs
 g52 Family = Tetrigidae, Id = Pygmy_grasshopper
 g53 Family = Acrididae, or Family = Gryllidae
 g54 Family = Gryllidae, Id = Field_cricket
 g55 Family = Acrididae
 g56 Id = Alfalfa_weevil
 g57 Id = A_weevil
 g58 Id = B_weevil
 g59 Id = Blue_metallic_weevil
 g60 Id = Grey_blistar_beetle
 g61 Id = Striped_blistar_beetle
 g62 Generic_id = Generic_id = Blister_beetles
 g63 Id = Potatoe_leaf_hopper
 g64 Id = Aster_leaf_hopper
 g65 Id = Leaf_hopper
 g66 Id = Big_spined_bugs
 g67 Id = Seed_bugs
 g68 Id = Seed_bugs_and_relatives
 g69 Id = differential_grasshopper
 g70 Id = Red_jagged_grasshopper
 g71 Id = Two_striped_grasshopper
 g72 Id = Migratory_grasshopper
 g73 Generic_id = Grasshoppers_and_relatives

The parser was modified as follows: given a rules block of the form -

```

ruleblkname RULES
VARS = varsblkname ;
PRECOND = lhsnode1;
TERMINATION = lhsnode2;
PROP = CONTROL
{UTILITY,BACKWARD,FORWARD,INORDER}
%%
lhs-rule :> rhs-rule ;
.
.
.
END

```

then the following tuples are produced -

```

(rulesblkname MKEXEC MKCS MKREF1 lhsnode1)
(rulesblkname MKEXEC MKCS MKREF2 lhsnode2)
(rulesblkname MKPROP textnode)

```

where the lhsnode1 and lhsnode2 refer to items which are identical to left hand sides of rules and are directly evaluable by the RULE EVALUATOR. Under our proposed control scheme, preconditions are applied for each rule group and if satisfied the rule group is entered (initiated), and rules will start to be executed using the type of control scheme specified by the text in the CONTROL property of the rule block.

Termination conditions are evaluated after each rule in the rule group to test whether the rule group should be exited. This continues until there are no more rules in the case of the sequential scheme or until the termination condition is satisfied.

14.3. Multigoals

PLANT/ds only allows a single variable on the RHS to determine the goal. Certain complications of goals might be imagined:

- a) structured values of goal variable,
- b) dynamically changing sets or structure of values for goal variables,
- c) dynamically changing "goal" variable,
- d) sets of goal variables,
- e) dynamically changing sets of goal variables.

As a minimum, an array of goals which will be determined in one of 2 ways is desirable.

Option 1 If there is a GOALS specification in the rules block, the goals will be taken to be that of a goals tuple. That is, given a rules block -

```

ruleblkname RULES
VARS = varsblkname ;
GOALS = (goal1, goal2, goal3, ...);

```

lhs-rule :> rhs-rule ;

.

.

END

produce a tuple of the form -

(rulesblkname MKEXEC MKCS MKREF0 goal1 goal2 goal3 ...)

with the idea that goals occur in order of their priority.

Option 2 If this GOALS designation is not present for a rule group, then all right hand side variables will be used as candidate goals.

A special tuple modifies each rule group as it is read in to update the information about which rule groups have been defined. This tuple will appear under the special node GLOBALS where the global variables are already hung. Hence the GLOBALS tuples form is:

(GLOBALS MKVARS var1 var2 ...)

and

(GLOBALS MKRULES rulegroup1 rulegroup2 ...) .

14.4. Disjunction on the Right Hand Side of Rules

Several cases arise in the entomology domain where it is desired to pursue one classification option but if it does not work out (leads to a contradiction or low confidence of decision), then it should be abandoned and another option should be selected to pursue. These points of backtracking can easily be recorded in rules and weights can indicate which options should be considered in what order.

For example, the rule :

```
[Mouth Parts = Chewing] [Hind Legs = Jumping]
→
[Order = Orthoptera : 0.9]
  V
[Order = Coleoptera : 0.2] [Insect = Flea Beetle : 0.1]
```

has the interpretation "pursue Orthoptera first, but if that does not work out, then try Coleoptera given there are no more promising options." This style of parallel reasoning assumes that the values of goals are mutually exclusive and is most easily effected by a priority queue.

Another flavor of the disjunction on the right hand side of rules is for independent values of a variable each having the possibility of each satisfying LHS selectors simultaneously. This latter option was not undertaken in our system. We distinguish the 2 types of disjunction by allowing a special symbol for each type. No changes to the Rule Evaluator are presently proposed for this distinction.

14.5. Data Acquisition

A special data acquisition method was proposed to operate as a expert system apart from the inference engine. The control scheme was to pass a list of ranked variables that would be of most use. The data acquisition system makes considerations based upon the following criterion as to what questions to ask and in what order (possibly on an automatically built form):

- (1) hierarchies of variables - certain variable should be asked before other variables, that is, it does not make sense to ask confirmatory data before primary data (even though the confirmatory data has more apparent usefulness),
- (2) flow of consciousness - certain questions seem too jumpy if asked out of sequence, semantic closeness should be taken into account,
- (3) certain variables values imply other values - restriction rules which assign defaults and does-not-apply values should be used, a graph structure will be used for this,
- (4) structured variables will be implemented so as to propagate uncertainty in a reasonable manner, more use of the mutual exclusive and independent values will be made,
- (5) some distinction will be made between different type of unknown - that which is uncertain, that which does not apply, that which may not be asked for again, etc.,
- (6) some values of variables need never be asked at some point in a consultation and this can be inferred from the remaining viable rules,
- (7) restriction rules need not achieve threshold to fire but propagate uncertainty.

14.6. Additional Research Goals

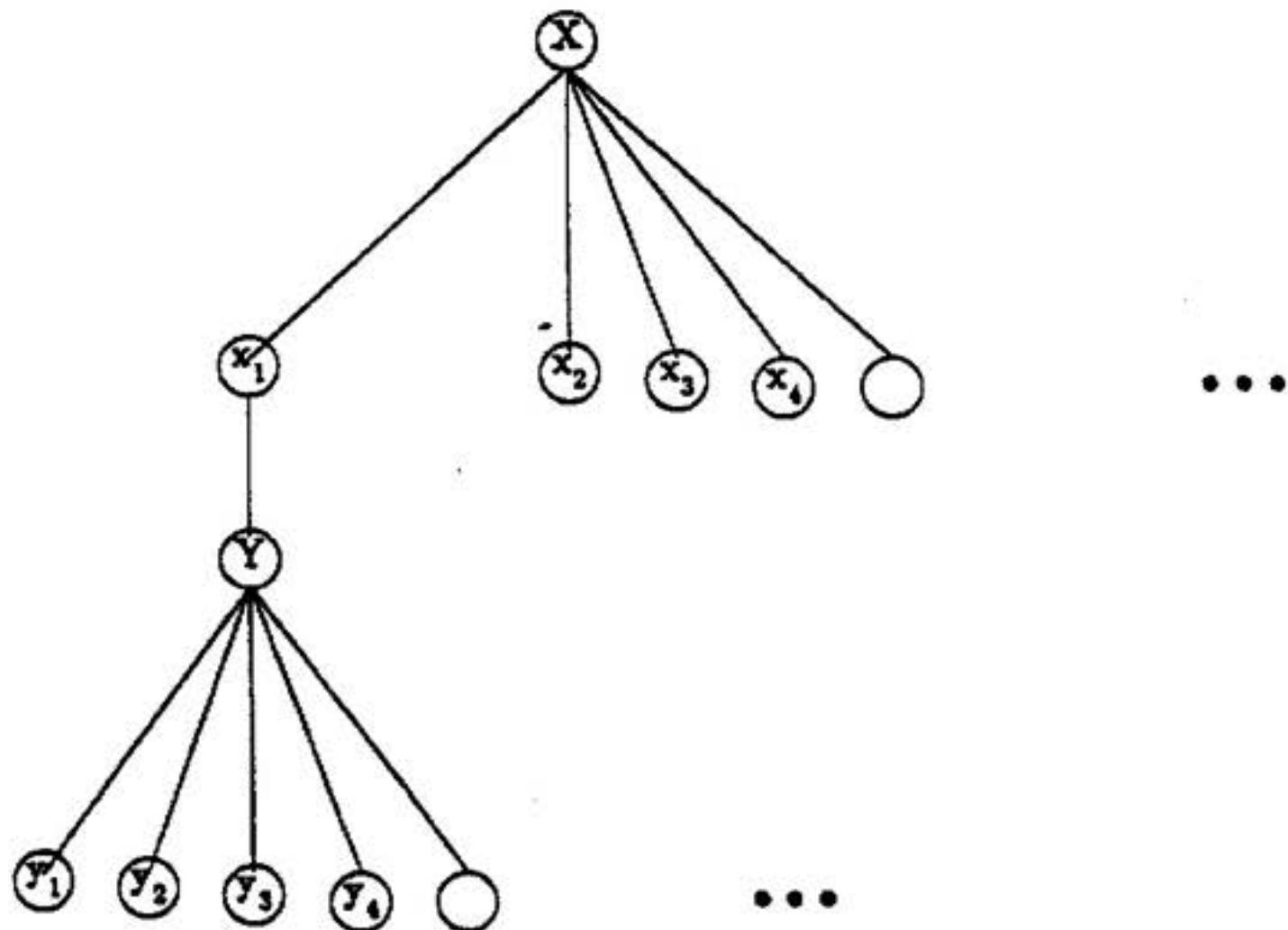
In the course of this project, several additional interesting areas of research were uncovered and pursued. These are described here.

14.6.1. Describing Variable-Value Relations with Semantic Nets

In PLANT/ds, the intrinsic relationships among various variables and their values were necessarily represented by a series of restriction rules which dictated when values of some variable prohibited other variables from having semantically inappropriate values. For example, there were a number of rules of the form:

```
[CONDITION_OF_LEAVES = NORMAL] :>
[CONDITION_OF_LEAVES_BELOW_AFFECTED_LEAVES = DOES_NOT_APPLY]
[POSITION_OF_AFFECTED_LEAVES = DOES_NOT_APPLY]
[LEAF_DISCOLORATION = NONE][SHOT_HOLING = ABSENT]
[LEAF_WITHERING_AND_WILTING = ABSENT]
[LEAF_MALFORMATION = ABSENT][LEAF_SPOTS = ABSENT]
[LEAF_MILDEW_GROWTH = ABSENT][SHREDDING = ABSENT];
[ROOTS = NORMAL] :> [ROOT_SCLEROTIA = ABSENT]
[ROOT_ROT = ABSENT]
[ROOT_GALLS_OR_CYSTS = ABSENT];
```

Such information indicates only one sort of behavior: whether or not the "normal" or "default" value of one variables value should preclude other variables from being asked



Derived Rule :

$$[X=x_2] \vee [X=x_3] \vee [X=x_4] \vee \dots \vee [X=x_n] \vee [X = \text{does_not_apply}] \rightarrow [Y = \text{does_not_apply}]$$

Figure 43 : Abstract Outline of Algorithm Converting Variable/Value Graph to Rules.

[Michalski and Chilausky80]. This information is more naturally expressed as a semantic network specifying connectivity between variables and values. Such a representation does not preclude the use of rules, and an interface algorithm was designed which converts such semantic nets into rules. Figure 43 illustrates the algorithm for an abstract graph. This is a simple matter of a tree traversal from top to bottom if there are no common subtrees.

Complications arise in the case of a DAG where there is a common subtree. This is currently taken to mean that two variables simultaneously must be considered by an AND condition of the rule as shown in Figure 44. An option exists to specify an "independence" annotation for the common node allows the node to have the interpretation that either variable may influence the d.n.a. assignment to it.

A partial ordering is being produced in both the simple and latter case, and a consistent total ordering must be derived for the rules if they are to be fired sequentially.

The rules can be made order independent by use of "OR" operands as in figure 45. Depending on the needs of the particular system, the graph traversal algorithm could be modified appropriately.

14.6.2. Dynamically Changing Value Sets for Variable

Much to the chagrin of the consultation user, the prototype system once queried the user as to which of 30 different values the variable COLOR should be assigned. This was extremely annoying, apart from the inability of the display routines to fit so many values on a single screen, because it was obvious at the point of query that only a few of the defined values were either relevant or possible.

There are various mechanisms for achieving the dynamically varying value sets for variables. One of these would dictate that after each query, a computation might be effected to find all relevant domain values for any plausible conclusions supportable by the current state of the system. Another strategy might dictate that this only be done for particularly

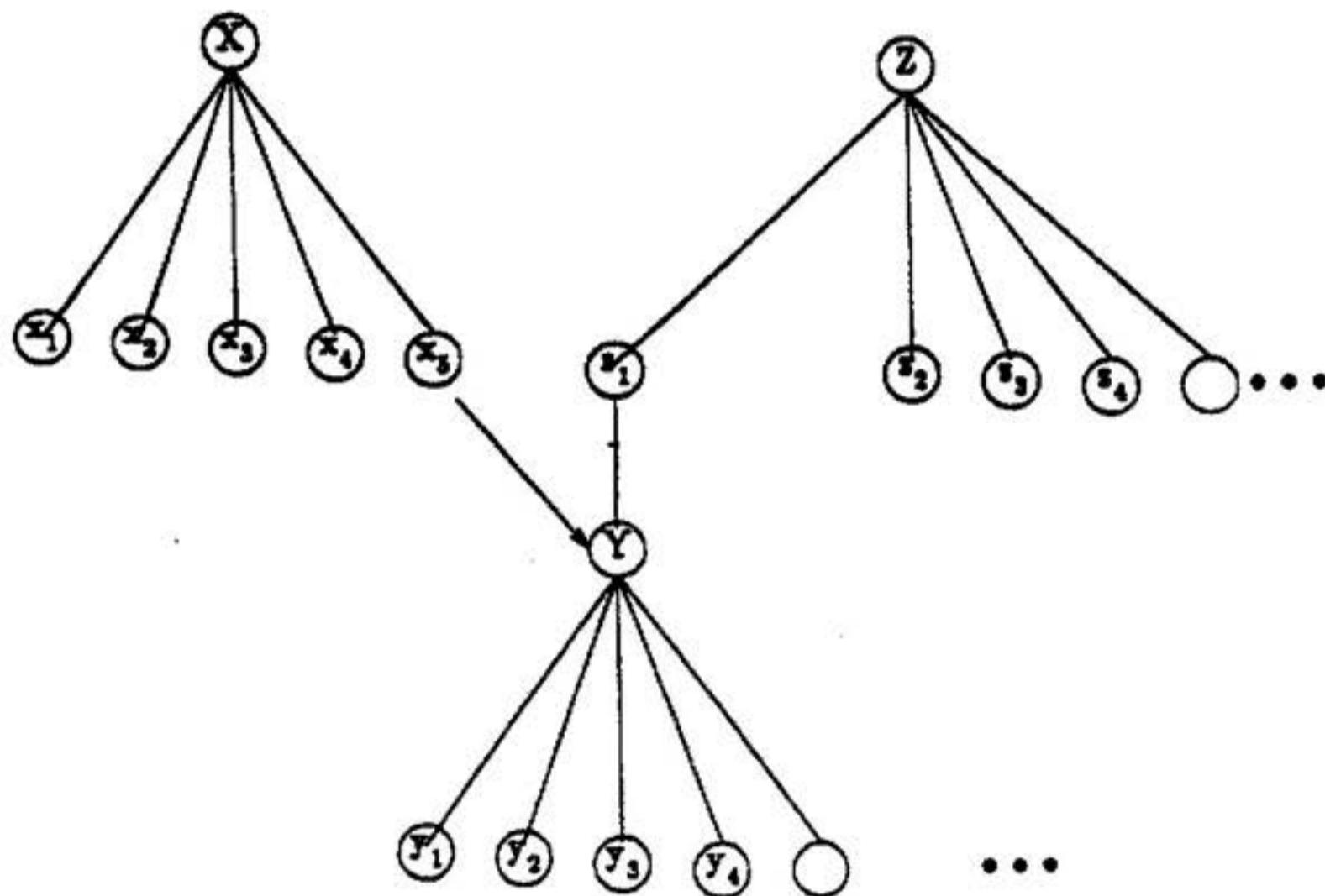
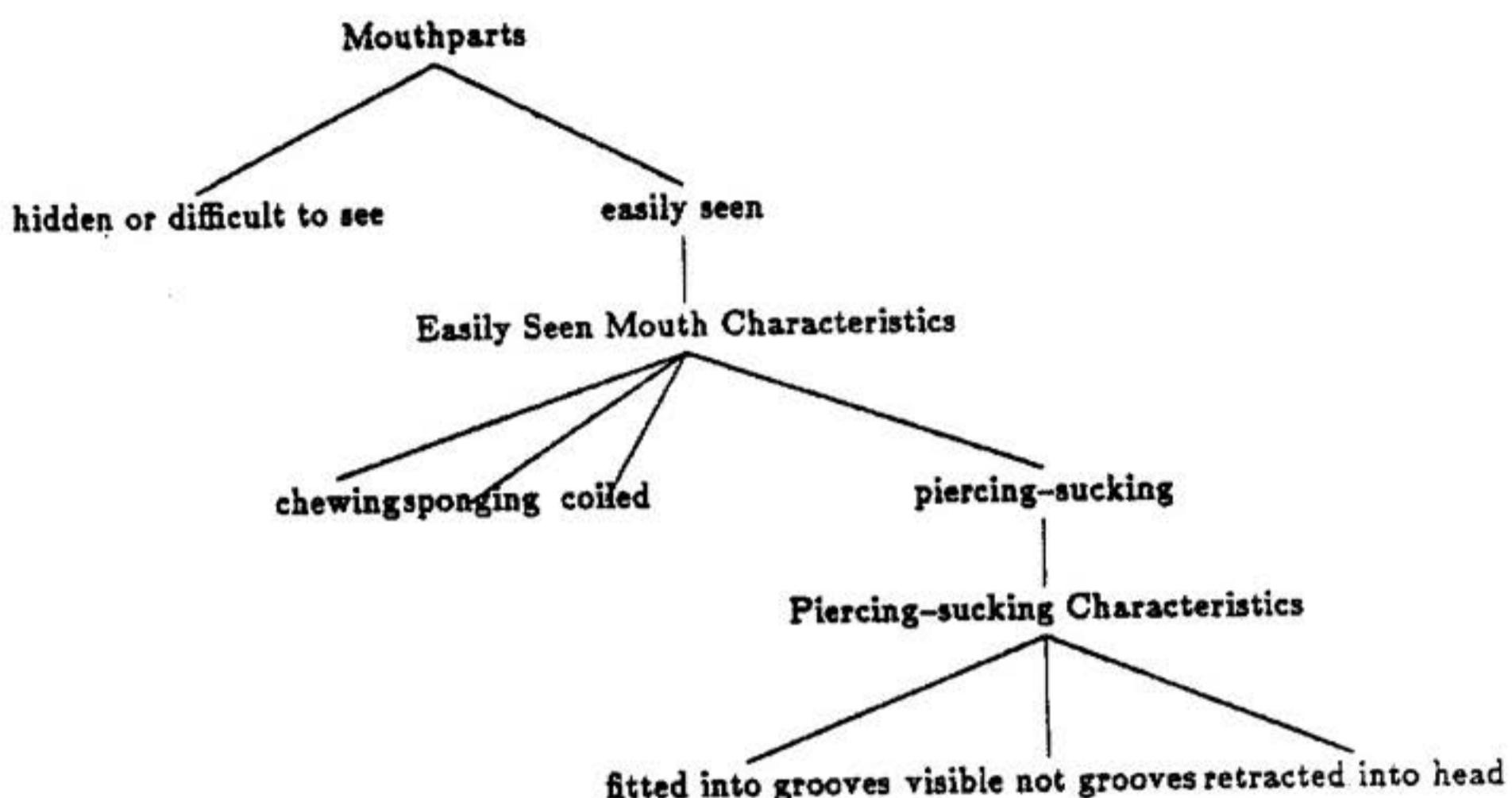


Figure 44 : The Case of a Common Subtree in Variable/Value Graphs.



Rules :

[Mouthparts = hidden or difficult to see]
 v[Mouthparts = does not apply]
 → [Easily Seen Mouthpart Characteristics = does not apply]

[Easily Seen Mouthpart Characteristics = chewing]
 v[Easily Seen Mouthpart Characteristics = coiled]
 v[Easily Seen Mouthpart Characteristics = sponging]
 v[Easily Seen Mouthpart Characteristics = does not apply]
 v[Mouthparts = hidden or difficult to see]
 v[Mouthparts = does not apply]
 → [Piercing-sucking Characteristics = does not apply]

Figure 45 : Typical Order Independent Rules Derived from a Variable/Value Graph.

annotated variables, and then only when the variable is asked for (no sense in computing the value set until it is actually needed for a query). Another strategy involves defining a series of variables which "inherit" values from previous variables as a rule group is started up, or defining rules that regulate the value of a single *structured* variable based upon an

auxiliary variable encapsulating state information.

Referring to Figure 42, the reader will be convinced that is an appropriate level of granularity for such computations. It is here that it is most relevant to talk about the possible current and subsequently-possible values of a variable based on the current state of the system. The dynamic nature of a consultation, with possible modifications to the state of variables necessitates undoing the conclusions drawn up to a particular point, and possibly revising the "possible-values" of such variables. This is currently accomplished by a complete traversal of all rule groups for which the precondition does not explicitly fail, and recording all values for the variable sought which explicitly occur in such rule groups. This is only an approximate solution (and costly even so) since some rules may be judged to be below threshold based on current state.

A better solution is possible though. A pretraversal of the rule group parse tree by a special compiler could generate a graph structure that indicates the relative correspondence between GOAL state and a given variable. This would ultimately be available by a simple table lookup or annotation to the rule group parse tree.

14.7. Future Research Extension

Enhancements to the current system might include the following.

- inverse operation to converting semantic net to rules (rules to s.n.)
for verifying human rule knowledge
- visual entry of the restriction information on vars
- more general relations over var-val semantic nets
 - value of one var sets another
 - probabilistic
- modifying the algorithm for traversing the vars graph to get
restriction values dynamically
- visual display of the tree of possibilities – allowing to select the
most probable failure point or use a ranking of reliability over vars
- if a hi freq var occurs as RHS in restriction rule
 - then all the LHS vars should be elevated to at least its value
 - plus the value it has from other rules
 - what if it is in more than one restriction rule
 - what if more than one RHS var applies
 - what about redundant effects
- if a rule group fails
 - should run again
 - there should be a way of forcing a conclusion
 - there should be a way of cycling or back tracking to modify
suspect values – eg., freq. mistyped or misperceived vals
- only relevant values should be asked
- restriction rules should be in a separate rule group with a lower
threshold than other rules - change parser + RE to allow LHS fwt
to be RHS conf - global RE option / syntactic in rule
 - a THRESHOLD prop on rule group
- straighten affairs among UNDEF, UNKN, ALL, NA

The biosphere of the alfalfa field is a complex structure. Once all the vectors present in the field are identified, it is no simple task to decide on the proper management procedures to

be applied. For instance, there may be beneficial insects as well as detrimental ones. Insecticide may be inappropriate for any of a number of reasons, prompting the introduction or enhancement of environment for beneficial insects. It may not even be worth the cost of applying insecticide if the alfalfa can be cut in reasonable time to ensure a good yield. These considerations make it clear that the management of alfalfa fields could be an expert system in its own right.

CHAPTER 15

Summary and Future Work

We have described the ADVISE meta-expert system at the conceptual and technical level. The system can be used for constructing and experimenting with expert systems in specific domains by:

- (1) assembling knowledge about the problem,
- (2) encoding the knowledge in some formal manner,
- (3) possibly using the knowledge for computer inductive inference, and
- (4) picking the inference mechanism(s) to be used on the encoded knowledge.

At present, we have experimentally implemented four expert consultation systems using the ADVISE meta-expert system, PLANT/ds for diagnosing soybean diseases, PLANT/cd for predicting extent of Black Cutworm damage to corn, BABY for monitoring infants in a neo-natal intensive care unit, and ALFALFA for identification and classification of insect pests.

15.1. The Language of the Future

In our system development efforts thus far, we have been evaluating a general purpose language such as PASCAL on a UNIX system for use in developing advanced expert system techniques. We did this because of the wider audience available for expert system tools that use a standard programming language and a common operating system environment. We have demonstrated that such a system is definitely possible. Indeed, there is an ongoing effort by an ITT research lab continuing the PASCAL work. This development effort indicates that the continued evolution of the PASCAL effort has ceased to be basic research and has become applied engineering which is best carried out in such an industry setting.

Although we do not intend to abandon PASCAL, the increasing use of LISP for learning programs developed in our immediate research environment and the extensive developments in LISP machine-based program development tools has prompted us to move all future ADVISE developments to common areas of interest. The major new areas of research activity are summarized below:

1. Handle temporal data and processes — knowledge representations, learning, and expert system performance;
2. Explore the development of very large knowledge bases which evolve over time rather than being rebuilt;
3. Use our meta-expert system to develop an expert system that can help a knowledge engineer tailor ADVISE options and build a knowledge base using learning programs;
4. Develop methods for trading time and computational cost against precision of the result;
5. Redefine our architecture to view an expert system as a collection of cooperating experts: dialogue manager, planner, rule evaluator, constraint propagator, etc;
6. Explore a multiprocessing/multiprocessor approach to distributing problem solving among multiple cooperating experts.

The use of LISP as the basis for the new developments in ADVISE will allow more run-time binding of system components together in different combinations. Although we will develop our new tools in LISP, we will remain mindful of the possibility of transferring at least a subsystem to smaller machines either using subsets of LISP or more traditional programming environments.

15.2. Extension of Present Efforts

In addition to the new areas of work outlined above, we intend to continue the work in a number of presently active areas. We plan to build on our present experience to extend the ADVISE paradigm in the following areas:

1. We presently support multiple knowledge representations, but the representation (which may be a hybrid) must now be chosen before knowledge base building begins; we intend to allow a choice for subsets of the knowledge base;
2. We presently support multiple control schemes but only one may be used for each different knowledge base; we plan to allow the control scheme to be specified as part of the knowledge base and varied as required within a single knowledge base;
3. We have multiple uncertainty propagation mechanisms available and we are conducting studies of the various methods in order to better guide a knowledge engineer in matching uncertainty mechanisms to real world problems;
4. We plan to support tighter coupling to learning programs and to develop techniques for learning during a consultation;
5. We are developing methods for representing knowledge about conducting a dialog with an expert system user as an identified part of the knowledge base over which inference can be conducted;
6. We have developed methods for handling reasoning with both positive and negative evidence, and plan to incorporate these methods into a working prototype.

Acknowledgements

A system of the size and complexity of ADVISE could not have been developed without collaboration and contributions from many individuals. The authors wish to acknowledge in particular Stephen Borodkin, Albert Boulanger, Bob Reinke, Lance Rodewald, Kent Spackman, and Dr. Robert E. Stepp, who made significant contributions to ADVISE at various stages of development and testing. Much of their work is described in various chapters of this document.

This research was supported in part by: the Office of Naval Research (grants No. N00014-83-G-0030 and No. N00014-82-K-0186), the Defense Advanced Research Project Agency (grant No. N00014-85-K-0878), and the National Science Foundation (grant No. NSF DCR 84-06801).

REFERENCES

- Bain, P.W., "An Algorithm to Perform Feature Selection on Nominal and Ordinal Features Using Non-Statistical Criteria", Report No. 1078, Department of Computer Science, University of Illinois, Urbana, Illinois, 1982.
- Baskin, A.B. and Levy, A.H., "MEDIKAS - An Interactive Knowledge Acquisition System", *Proc. of the 2nd Symposium on Computer Applications in Medical Care,* Washington, D.C., IEEE Publications, Nov. 5-7, 1978.
- Baskin, A.B., "A Methodology for Machine Representation of Medical Knowledge," PhD Dissertation, Department of Computer Science, University of Illinois, Urbana, Illinois, 1979.
- Baskin, A.B., "LOGIC NETS: Variable-valued Logic plus Semantic Networks," in *Policy Analysis and Information Systems*, No. 3, 1980.
- Bo, Ketil, "Human-Computer Interaction," *Computer*, Vol. 15, No. 11, 1982.
- Boulanger, Albert, "The Expert System PLANT/ed: A Case Study In Applying The General Purpose Inference System ADVISE To Predicting Black Cutworm Damage In Corn," Masters Thesis, Department of Computer Science, University of Illinois, Urbana, Illinois, 1983.
- Buchanan, B., "Expert Systems: Working Systems and the Research Literature," Knowledge Systems Laboratory, Report No. KSL-85-37, December, 1985.
- Buchanan, B.G. and R.O. Duda., "Principles of Rule-base Expert Systems", *Advances in Computers*, 22, Academic Press, 1983.
- Channic, T., "ADVISECORE: A Screen Package for Expert Systems", Report No. UIUCDCS-F-84-919, Department of computer Science, University of Illinois, Urbana, Illinois, 1984.
- Channic, T., "Editing Network-Structured Knowledge Bases in the ADVISE System", Report No. UIUCDCS-F-85-934, Department of computer Science, University of Illinois, Urbana, Illinois, 1985.

Date, C.J., *An Introduction to Database Systems*, Addison Wesley, 1977.

Davis, R. and King, J., "An Overview of Production Systems" in *Machine Intelligence*, S. Elcock and Michie (eds.), 1978.

Davis, J.H., "CONVART: A Program for Constructive Induction on Time Dependent Data", Thesis, Department of Computer Science, University of Illinois, Urbana, Illinois, 1981.

Duda, Richard O., Hart, Peter E., and Nilsson, Nils J., "Subjective Bayesian Methods for Rule-Based Inference Systems", *Technical Note 124*, SRI International, Melno Park, California, January 1976.

Duda, Richard O., et. al., "Development of the PROSPECTOR Consultation System for Mineral Exploration." Final Report for SRI Projects 5821 and 6415, SRI International, Melno Park, California, October 1978a.

Duda, R.O., P.E. Hart, N.J. Nilsson and G.L. Sutherland, "Semantic Network Representation in Rule-Based Inference Systems," in *Pattern Directed Inference Systems*, D.A. Waterman and F. Hayes-Roth (eds.), Academic Press, 1978b.

Eddy, David M. and Charles H. Clanton, "The Art of Diagnosis." *The New England Journal of Medicine*, Vol. 306, No. 21, 1263-1268, 1982.

Foley, J.D. and A. van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, 1982.

Frayer, William W., "Patient Data Management in Neonatal Intensive Care", *Clinics in Perinatology*, Vol. 7, No. 1, 1980.

Freedman, A.M., O.P. Buneman, G. Peckham, and A. Trattner, "Automatic Recognition of Significant Events in the Vital Signs of Neonatal Infants", *Computers and Biomedical Research*, Vol. 12, 141-148, 1979.

Galen, Robert S., "Predictive Value and Efficiency of Laboratory Testing", *Pediatric Clinics of North America*, Vol. 27, No. 4, 861-869, 1980.

Gevarter, W., "Expert Systems" Report No. NBSIR 82-2505. Washington D.C. : National Bureau of Standards, 1982.

Harmon, P. (ed.), "Inventory and Analysis of Existing Expert Systems," *Expert Systems Strategies*,

August, 1986.

Hayes-Roth, F., D.A. Waterman, and D.B. Lenat, eds., *Building Expert Systems*, Addison-Wesley, Reading, Mass., 1983.

Johnson, S., "Yacc: Yet Another Compiler-Compiler," Bell Laboratories, Murray Hill, New Jersey, 1978.

Human-Computer Interaction -- INTERACT '84. Proceedings of the IFIP Conference, organized by the Task Group of the Human-Computer Interaction (Human85), London, Sept. 4-7, 1984. North-Holland/Elsevier, 1985.

Larson, J., "INDUCE-1: An Interactive Inductive Inference Program in VL₂₁ Logic System," Report No. UTUCDCS-R-77-878, Department of Computer Science, University of Illinois, Urbana, Illinois, 1977.

La Gamma, Edmund F., "Concepts in Critical Data Evaluation and Neonatal Monitoring", *Clinics in Perinatology*, Vol. 7, No. 1, 93-106, 1980.

McMillan, R.S., "NEWVAR: Constructing New Variables," Internal Report, Department of Computer Science, University of Illinois, Urbana, Illinois, 1982.

Michalski, R.S., "AQVAL/1—Computer Implementation of a Variable-valued Logic System and the Application to Pattern Recognition," in *Proc. of the First International Joint Conference on Pattern Recognition*, Washington, D.C., October 30-November 1, 1973.

Michalski, R.S., "Variable-valued logic: System VL1", *Proceedings of the Fourth International Symposium on Multiple-Valued Logic*, Morgantown, West Virginia, 1974.

Michalski, R.S., "Designing Extended-Entry Decision Tables and Optimal Decision Trees Using Decision Diagrams," Report No. 898, Department of Computer Science, University of Illinois, Urbana, Illinois, 1978a.

Michalski, R.S., "Pattern Recognition as Knowledge-Guided Computer Induction." Report No. 927. Department of Computer Science, University of Illinois, Urbana, Illinois, 1978b.

Michalski, R.S. and J.B. Larson, "Selection of Most Representative Training Examples and Incremental Generation of VL₁ Hypothesis: The Underlying Methodology and the Descriptions of Programs ESEL and AQ11," Report No. 877, Department of Computer Science, University of Illinois, Urbana, Illinois, 1978.

Michalski, R.S. and R.L. Chilausky, "An Experimental Comparison of Several Many-valued Logic Inference Techniques in the Context of Computer Diagnosis of Soybean Diseases," *International Journal of Man Machine Studies*, 1980.

Michalski, R.S., "Knowledge Acquisition Through Conceptual Clustering: A Theoretical Framework and an algorithm for Partitioning Data into Conjunctive Concepts," *International Journal of Policy Analysis and Information Systems*, Vol. 4, No. 3, pp. 219-244, 1980.

Michalski, R.S., A.B. Baskin, and K.A. Spackman, "A Logic-based Approach to Conceptual Database Analysis," *Sixth Annual Symposium on computer Applications in Medical Care*, Washington, D.C., 1982a.

Michalski, R.S., Davis, V.S. Bisht and J.B. Sinclair, "PLANT/ds: An Expert System for the Diagnosis of Soybean Diseases," European Conference on Artificial Intelligence, July 12-14, 1982b.

Michie, D. (ed.), *Expert Systems in the Micro Electronic Age*, Edinburgh University Press, 1979.

Moran, T.P., "An Applied Psychology of the User," in *ACM Computing Surveys*, Vol. 13, No. 1, March 1981.

Nilsson, N.J., *Principles of Artificial Intelligence*, Tioga Publishing Co., Palo Alto, California, 1980.

Pople, Harry E., "Heuristic Methods for Imposing Structure on Ill-Structured Problems: The Structuring of Medical Diagnostics", *Artificial Intelligence in Medicine*, Peter Szolovits (ed.), Westview Press, Boulder, Colorado, pp. 119-190, 1982.

Riedesel, J., M. Jha, B. Rao and R. Reinke, "A Brief Review of Meta-Expert Systems," ISG Internal Report, Department of Computer Science, University of Illinois, Urbana, Illinois, 1986.

Schubert, Richard, "The VL Relational Data Sublanguage for an Inferential Computer Consultant," Masters Thesis, Report No. 846, Department of Computer Science, University of Illinois, Urbana, Illinois, 1977.

Spackman, Kent, "Integration of Inferential Operators with a Relational Database in an Expert System," Masters Thesis, Department of Computer Science, University of Illinois, Urbana, Illinois, 1982.

The Spang Robinson Report, "Evaluating Expert Systems Tools," vol. 2, No. 10, October 1986.

Stauffer, M., "GEM User's Manual," Internal Report, Department of Computer Science, University of Illinois, Urbana, Illinois, 1982.

Stefik, M., J. Aikins, R. Balzer, J. Benoit, L. Birnbaum, F. Hayes-Roth and E. Sacerdoti, "The Organization of Expert Systems, A Tutorial," *Artificial Intelligence*, 18, 1982.

Stepp, R., "Learning From Observation: Experiments in Conceptual Clustering," in *Proc. Machine Learning Workshop-Symposium*, Carnegie-Mellon University, Computer Science Department, July 16-18, 1980.

Thursh, D. and Mabry, F., "An Interactive Hyper-text of Pathology," in *Proc. of the 4th Annual Symp. on Computer Applications in Medical Care*, IEEE, Nov., 1980.

Troester, S., "Damage and Yield Reduction in Field Corn Due to Black Cutworm Feeding: Results of a Computer Simulation Study." *Journal of Economic Entomology* Vol. 75, pp.1125-1131, 1982a.

Troester, S., Clement, S., Showers, and A.J. Keaster, "Determining Yield Loss by Black Cutworms on Corn." *ASAE paper No. 82-5026*. St. Joseph, Mi., 1982b.

Troester, S., Ruesink W., and Rings R., "A Model of Black Cutworm (*Agrotis Ipsilon*) Development: Description, Uses, and Implications." *Illinois Agricultural Experiment Station Bulletin*, No. 774. pp. 1-33, 1982c.

Troester, S., C.D. Bremer, K.L. Steffey, D.E. Kuhlman, and R.H. Meyer, "Avoiding Black Cutworm Losses: An Educational Manual for a Computer Model that Demonstrates Effects of Corn Production Practices on Black Cutworms," Illinois Natural History Survey, Champaign, Illinois, 1982d.

van Melle, W., "A Domain Independent Production Rule System for Consultation Programs," *Proc. Sixth IJCAI*, 1979.

Winston, P. H., *Artificial Intelligence*, pp. 254-257, Addison-Wesley, 1984.

Woods, William A., "What's Important About Knowledge Representation", *Computer*, Vol. 16, No. 10, 22-29, 1983.

APPENDIX A

Rule Parser Details

This appendix contains additional information regarding the rule parser. Section A.1 describes some basic lexical notions concerning input to the parser. In section A.2 are details of error handling by the parser. Section A.3 defines the "mark" types currently handled by the parser.

A.1. Basic Lexical Notions

Format: All input is free-format. Lines can be as long as desired.

Comments: Comments may appear anywhere in the input, and are denoted by a leading !. The comment continues for the remainder of the line (i.e. terminated only by EOLN).

Identifiers: Identifiers must begin with a letter, and then may contain up to 39 other non-punctuation symbols including digits, @ _ and '. Note that apostrophes are NOT used to enclose strings of otherwise-illegal characters.

Variables: A variable is (to the parser) an identifier which has been previously defined as a VL variable name.

Integer: A sequence of digits without a decimal point.

Real number: A sequence of digits with a decimal point. NOTE THAT A REAL MUST BEGIN WITH A DIGIT. ".9" IS NOT A REAL NUMBER.

The parser can process ASCII input files containing lower-case text, however because of the escape-code nature of lowercase material, the identifiers may be only 20 lower-case letters long. All parser keywords (i.e. RULES, VARS, etc.) are recognized only as upper-case characters.

A.2. Error Information

When a syntax error is detected the parser does four things:

- (1) A ^ is written under the offending symbol.
- (2) The message **** SYNTAX ERROR is printed.
- (3) The list of symbols which would be acceptable is output.
- (4) The parser skips to the next unit of information, e.g. skips to the next rule, next variable definition or next block.

The error return code given by the parser may be NOERROR when no errors have occurred, or one of the following:

ERRPRNOPARTAB	:	The PARTAB file could not be read.
ERRPROSYNTAX	:	One or more syntax errors were detected.
ERRPROVERFLOW	:	Internal parser stack overflow.
ERRPRINTERNAL	:	The parser made a logic error which was fatal.
ERRPRUNSUPPORTED	:	An unsupported language feature was used.
ERR<other>	:	Any other fatal error code from supporting system (e.g. the tuplemanager) from which there is no recovery.

The parser writes additional messages to the MSG file to help diagnose some of these error types.

The parser writes additional messages to the MSG file to help diagnose some of these error types.

A simple parser driver program exists which utilizes two files of knowledge. Before parsing, a preexisting network is loaded from the file BACKUP. The parser is invoked on input file INPUT and giving output file OUTPUT. Subsequent to parsing, BACKUP is rewritten to contain the updated network (BACKUP is processed with READBACKUP and WRITEBACKUP directives). Also the file NETWORK is written via the WRITETEXT directive and gives a readable representation of the network.

A.3. Mark Type Definition

The parser generates a network of tuples for each block (i.e. for each RULES block and each VARS block). The name of the block (the group name) is the only handle for the network of tuples, whose head node is located via the directory, given the group name. From this node on in the block network, all tuples have a "parser mark" node in them located in the first node position, tuple[2]. The parser mark (or just "mark") indicates the functionality of the tuple for interpretation or decoding. The marks which are used this way are as follows:

A.3.1. Within a VARS Block

These mark types are generated from the VARS block definitions:

MKVARS: a list of variable-defn nodes in this tuple

MKDOMAIN: the nodes in this tuple define the domain

[2]MKDOMAIN

[3] domain type:

MKNOMINAL: additional nodes give either

- one integer: number of levels or
- list of values

MKINTERVAL: additional nodes give either

- two integers: interval range or
- ordered list of values

MKSTRUCTURE: additional nodes give values for leaves and internal nodes in the structure tree

MKREFINE: gives the refinement structure of a structured variable tree

[2]MKREFINE
 [3] internal node name
 [4]... subordinate nodes to [3]

MKUNITS: gives the units of measurement for the variable
 MKVALCON: gives the value of an identifier or rule component. The tuple contains pairs of numbers (value, degree of confidence), where value may be either real or integer and where deg. of conf. is always real. When multiple alternative values are to be represented, many (value, degree of confidence) pairs may be used. When space in one tuple is exhausted, another MKVALCON tuple is incorporated to hold additional data.

A.3.2. Within a RULES Block

These mark types are generated within the RULES block:

MKNULL: a "no-operation" indicator
 MKVARDCL: for a group of rules, next node indicates variable definitions which are to be used
 MKBEHAVIOR: the node in position [3] denotes the behavior
 MKINCR: increasing ()
 MKDECR: decreasing ()
 MKRMIN: has a relative minimum ()
 MKRMAX: has a relative maximum ()
 MKPAREN: indicates parentheses used around this unit
 MKTVARS: indicates "target" variables used in the rule. A target variable is one which occurs to the left of the relation symbol in a selector.
 MKRVARS: indicates "reference" variables used in the rule. A reference variable is one which occurs to the right of the relation symbol in a selector. In either the MKTVARS or MKRVARS tuple the following node (tuple[3]) contains either MKRHSV or MKLHSV to indicate whether the variables listed occur in the rule's right- or left-hand side, respectively. Tuple positions from tuple[4] hold variable node data. Each variable cited occurs one or more times in the right or left hand side (according to tuple[3]) as a target or reference (according to tuple[2]).
 MKLHSV: indicates rules in a rule group which involve a specific variable in the left hand side part.
 MKRHSV: indicates rules in a rule group which involve a specific variable in the right hand side part. Tuple[3] contains a variable node and tuple[4] on to the end contain rule nodes which incorporate the variable within the left- or right-hand side (as indicated by tuple[2]).
 MKEXEC: indicates that the following node (tuple[3]) contains an "executable" code which can be one of the following:
 MKFALSE: denotes the constant value false
 MKTRUE: denotes the constant value true
 MKRULES: a list of rule nodes in this tuple
 MKRULE: rule lhs node and rule rhs node in this tuple
 MKLM: the following node is a linear module

MKCS: pairs of (lm-coeff, lm-node) in this tuple
 MKOR1: nodes are lin-modules to be or-ed together
 MKAND1: nodes are condition units to be and-ed together
 MKOR2: nodes are condition units to be or-ed together
 MKEXCPT: two nodes follow: node-a node-b
 MKEQUTV: two nodes follow: node-a $\leqslant \geqslant$ node-b
 MKIMP: two nodes follow: node-a \Rightarrow node-b
 MKAND2: the nodes are selectors to be and-ed together
 MKVAR: start of selector. there are five nodes plus optional nodes:
 [2]MKVAR
 [3] node for the variable
 [4] node for the relation (MKEQ,MKGT,MKLT,MKNE,MKGE,MKLE)
 [5] node for reference type (MKREF0,MKREF1,MKREF2)
 if MKREF0: one additional node follows in position [6]
 MKALL: entire domain (*)
 MKUNKN: unknown (?)
 MKUNDEF: undefined (\$)
 MKNA: not-applicable (NA)
 if MKREF1: one or more nodes follow denoting a list of nominal values
 if MKREF2: one or more sets of node triples of the form
 (low-value, high-value, weight) used for interval values and
 weighted values of both interval and nominal type
 if MKREF3: one or more sets of node quadruples of the form
 (low-value, high-value, weight₁, weight₂) used for interval values and
 weighted values of both interval and nominal type

MKFUNC1: the nodes in this tuple indicate a function reference similar to MKVAR above, except that the node for the name of the function is "floating", i.e. it contains no parse tree.
 To evaluate such a node, the dictionary must be consulted.
 The relation and reference nodes may both be omitted. If either is required, both are present.

MKFUNC2: the node in this tuple indicates the general function
 MKTRAP: the node in this tuple indicates a trap function
 MKARGS: the nodes are arguments to MKFUNC1 above

APPENDIX B

GVL₁ Grammar

This appendix lists the GVL₁ grammar at the time of this writing. Its format is as input to the YACC compiler-compiler available on Berkeley UNIX.

```
%token LP RP LSB RSB LBR RBR MOD IMPASGN IMP RMIN NE TO
/*      ( ) [ ] { } % ::> => / <> .. */
%token OR PLUS MINUS TIMES DIV EQUIV LE GE RMAX EQ COLON
/*      v + - * / <=> <= >= / > : */
%token COMMA DOT EXCPT GT LT SEMI TRUE FALSE UNKN UNDEF
/*      . , > < ; T F ? $ */
%token ID VAR INT REAL RULES VARS FUNCS END IN NA
%token NOMINAL INTERVAL STRUCTURE UNITS PROP TRAP
%%

file    → block
       | file block

block   → blk

blk     → idnode RULES vardefl funcdefl optprops rulebody END
       | idnode VARS optprops varsbody END
       | idnode FUNCS vardefl optprops funcbody END
       | error

vardefl → VARS EQ ID SEMI

funcdefl → FUNCS EQ ID SEMI
```

| c

rulebody → rule
| rulebody SEMI rule

varsbody → vardefn
| varsbody SEMI vardefn

funcbody → funcdefn
| funcbody SEMI funcdefn

rule → optidnode lmsnode IMPASGN strength lmsnode optprops
| error
| ε

optidnode → ID
| ε

optprops → prop
| optprops prop
| ε

prop → PROP EQ ID END

lmsnode → lms

lms → linearmodule
| lms OR linearmodule

linearmodule → quantifier LP lmsnode RP

```
| Imparts  
  
quantifier → LT optint idnode inset GT  
| ε  
  
optint → INT  
| ε  
  
inset → IN ref  
| ε  
  
Imparts → Import  
| Imparts PLUS Import  
  
Import → LP Imparts RP  
| optreal optreal condstmt  
  
optreal → realnode  
| ε  
  
condstmt → condstmt condition  
| condstmt OR condition  
| condstmt EXCPT condition  
| condition  
  
condition → term EQUIV term  
| term IMP term  
| term  
  
term → LP condstmt RP  
| selectors  
| TRUE
```

```
| FALSE

selectors -> selector
| selectors selector

selector -> LSB exprnode rel refgroup behavior wfunc RSB
| exprnode

wfunc -> exprnode
| ε

optrelref -> rel refgroup
| ε

rel -> EQ
| GT
| LT
| NE
| GE
| LE

refgroup -> ref

ref -> valunit
| ref COMMA valunit
| set weight
| ref COMMA set weight
| TIMES
| UNKN
| UNDEF
| NA

valunit -> exprnode      ddval weight
```

```
val  -> intnode  
| realnode  
| ID  
| varnode optargs  
| TRAP optargs
```

```
args  -> arg  
| args  
COMMA  
arg  
| ε
```

```
arg  -> exprnode
```

```
optargs  -> LP args RP  
| ε
```

```
ddval  -> TO exprnode  
| ε
```

```
weight  -> COLON exprnode optexpr  
| ε
```

```
optexpr  -> exprnode  
| ε
```

```
set  -> LBR vals RBR
```

```
behavior  -> DIV  
| EXCPT  
| RMAX  
| RMIN  
| ε
```

```
vals → val
| vals COMMA val
| ε
```

```
exprnode → expr
```

```
expr → aterm
| expr addop aterm
```

```
aterm → afactor
| aterm mulop afactor
```

```
afactor → MINUS afactor
| LP expr RP
| val
```

```
addop → PLUS
| MINUS
```

```
mulop → TIMES
| DIV
| MOD
```

```
strength → LP intrealnode COMMA intrealnode RP
| LP intrealnode RP
| ε
```

```
vardefn → idnode dfields optprops
| ε
```

```
dfields → dfield
| dfields dfield
```

```
dfield → dfieldpart

dfieldpart → NOMINAL EQ LP idsunique RP
           | NOMINAL EQ inttuple
           | INTERVAL EQ intrealtuple TO intrealtuple
           | INTERVAL EQ LP idsunique RP
           | STRUCTURE EQ LP idsunique RP LP refines RP
           | UNITS EQ idtuple

funcdefn → idnode var LP functab RP

functab → funcentry
          | functab funcentry

funcentry → idnode intrealnode intrealnode

var → varnode

refines → refinement
          | refines refinement

refinement → idtuple EQ LP ids RP

idsunique → idsu

idsu → idtu
      | idsu idtu

idtu → idintnode
```

idnode → ID

varnode → VAR

intnode → INT

realnode → REAL

intrealnode → intnode
| realnode

idintnode → idnode
| intnode

ids → idtuple
| ids idtuple

idtuple → idnode

inttuple → intnode

intrealtuple → intrealnode

APPENDIX C

Internal Representation of a BABY Network

```
(NODES_TO_PROPAGATE ())
(DATA_SET ())
(PATTERNS (
    (DOWNWARD NEONATE_PATTERNS )))
(ASSERTIONLIST ())
(NEONATE_PATTERNS (
    (MKVALTUPLE BAAND 0.1e1 0.0 0.163184079 0.099950124e1 )
    (DOWNWARD SIADH EVIDENCE 0.7968e-8 0.1631 0.09995e1 )
    (DOWNWARD HYPONATREMIC EVIDENCE 0.0 0.8 0.1e1 )
    (DOWNWARD GLYCOSURIA EVIDENCE 0.0 0.8 0.1e1 )
    (DOWNWARD NORMAL_ADRENAL_FUNCTION EVIDENCE 0.0 0.8 0.1e1 )
    (DOWNWARD NORMAL_RENAL_FUNCTION EVIDENCE 0.0 0.8 0.1e1 )
    (ASSERT 0.85 )))
(NORMAL_RENAL_FUNCTION (
    (MKEXEC MKRULE u002 MKNULL 0.1e1 0.1e1 )
    (MKTVARS MKLHSV NRF )
    (UPWARD SIADH EVIDENCE )
    (UPWARD NEONATE_PATTERNS EVIDENCE )
    (MKVALTUPLE BARULE 0.8 0.0 0.8 0.1e1 )
    (ASSERT 0.95 )))
(NORMAL_ADRENAL_FUNCTION (
    (MKEXEC MKRULE u004 MKNULL 0.1e1 0.1e1 )
    (MKTVARS MKLHSV NAF )
    (UPWARD SIADH EVIDENCE )
    (UPWARD NEONATE_PATTERNS EVIDENCE )
    (MKVALTUPLE BARULE 0.8 0.0 0.8 0.1e1 )
    (ASSERT 0.99 )))
(GLYCOSURIA (
    (MKEXEC MKRULE u006 MKNULL 0.1e1 0.1e1 )
    (MKTVARS MKLHSV GLY )
    (UPWARD ba1 EVIDENCE )
    (UPWARD NEONATE_PATTERNS EVIDENCE )
    (MKVALTUPLE BARULE 0.8 0.0 0.8 0.1e1 )
    (ASSERT 0.1 )))
(HYPONATREMIC (
    (MKEXEC MKRULE u008 MKNULL 0.1e1 0.1e1 )
    (MKTVARS MKLHSV HYP )
    (UPWARD SIADH CONTEXT 0.75 )
    (UPWARD NEONATE_PATTERNS EVIDENCE )
    (MKVALTUPLE BARULE 0.8 0.0 0.8 0.1e1 ))
```

```

(ASSERT 0.5e-1 )))

(SIADH (
  (UPWARD NEONATE_PATTERNS EVIDENCE )
  (MKVALTUPLE BAYESIAN 0.8 0.79680e-8 0.16318 0.09995e1 )
  (DOWNWARD ba0 EVIDENCE 0.1e1 0.0999e-2 0.1e1 0.4875e-1 0.1e1 )
  (DOWNWARD NORMAL_ADRENAL_FUNCTION EVIDENCE 0.1e1 0.0999e-2
   0.0999e-2 0.1e1 0.1e1 )
  (DOWNWARD NORMAL_RENAL_FUNCTION EVIDENCE 0.1e1 0.0999e-2
   0.0999e-2 0.1e1 0.1e1 )
  (DOWNWARD HYponatremic CONTEXT 0.501e3 0.19960e-2 0.19960e-2
   0.1e1 0.500999999e3 )
  (ASSERT 0.5e-1 )))

(ba0 (
  (UPWARD SIADH EVIDENCE )
  (MKVALTUPLE BAOR 0.1e1 0.1e1 0.19999999 0.1e1 )
  (DOWNWARD ba4 EVIDENCE 0.0 0.1 0.1e1 )
  (DOWNWARD ba3 EVIDENCE 0.0 0.1 0.1e1 )
  (DOWNWARD ba2 EVIDENCE 0.0 0.1 0.1e1 )
  (DOWNWARD ba1 CONTEXT 0.1e1 0.19999999 0.0 ))) 

(ba1 (
  (UPWARD ba0 CONTEXT 0.75 )
  (MKVALTUPLE BANOT 0.1e1 0.1e1 0.19999999 0.0 )
  (DOWNWARD GLYCOSURIA EVIDENCE 0.1e1 0.1e1 0.1e1 ))) 

(ba2 (
  (MKEXEC MKRULE u010 MKNULL 0.1e1 0.1e1 )
  (MKTVARS MKLHSV URINE_OSM )
  (UPWARD ba0 EVIDENCE )
  (MKVALTUPLE BARULE 0.1 0.0 0.1 0.1e1 0.8 ))) 

(ba3 (
  (MKEXEC MKRULE u012 MKNULL 0.1e1 0.1e1 )
  (MKTVARS MKLHSV URINE_OSM )
  (UPWARD ba0 EVIDENCE )
  (MKVALTUPLE BARULE 0.1 0.0 0.1 0.1e1 0.8 ))) 

(ba4 (
  (MKEXEC MKRULE u014 MKNULL 0.1e1 0.1e1 )
  (MKTVARS MKLHSV URINE_SG )
  (UPWARD ba0 EVIDENCE )
  (MKVALTUPLE BARULE 0.1 0.0 0.1 0.1e1 0.8 ))) 

(GLOBALS (
  (MKTVARS NRF NAF HYP GLY SERUM_OSM URINE_OSM URINE_SG ))) 

(SERUM_OSM (
  (MKDOMAIN MKINTERVAL 1 1000 ))) 

(URINE_OSM (
  (MKDOMAIN MKINTERVAL 1 1000 ))) 

(URINE_SG (
  (MKDOMAIN MKINTERVAL 0.1e1 0.2e1 ))) 

(RULESET (
  (MKLHSV URINE_SG ba4 )
  (MKLHSV URINE_OSM ba2 ba3 )
  (MKEXEC MKRULES NORMAL_RENAL_FUNCTION NORMAL_ADRENAL_FUNCTION
   GLYCOSURIA HYponatremic ba2 ba3 ba4 ))) 

```

```
(MKVARDCL GLOBALS )))  
(u010 ( MKEXEC MKVAR URINE_OSM MKGT MKREF2  
        SERUM_OSM SERUM_OSM 0.1 ))  
(u012 ( MKEXEC MKVAR URINE_OSM MKGT MKREF2 300 300 0.9 ))  
(u014 ( MKEXEC MKVAR URINE_SG MKGT MKREF2 0.101e1 0.101e1 0.8 )))
```