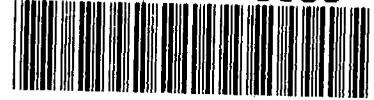
LINEAR LIBRARY
C01 0068 1386



The Development of an Expert System Shell With A Mixed Knowledge Representation, Explicit Control of Reasoning and a Truth Maintenance System.

By : Guy Jacobson

Submitted in fulfillment of requirements for the M.SC degree in Computer Science at the University of Cape Town.

Supervisor : Professor K.J. MacGregor

The University of Cape Town has been given the right to reproduce this thesis in whole or in part. Copyright is held by the author.

The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

University of Cape Town

PREFACE.

I would like to thank my thesis supervisor Professor K.J. MacGregor for all his assistance and guidance.

I would also like to thank Dr Basil Payne, not only for the many fruitful discussions concerning the WISE truth maintenance system and reasoning features, but also for all the time off work that has enabled me to complete this thesis. His cooperation and support were of great value to me.

A special word of thanks to my fiance Dr Sarah Friedman for proof reading the thesis and for all the moral support she has given me.

NOTE :

- In order to make the meaning of certain sentences unambiguous, some of the terms and names used have been enclosed in double quotes (").
- The spelling in this thesis is according to english as written in North America.
- This thesis was printed on a HP Rugged Writer and written using MS-word. The diagrams were done using graphics gallery.

PART1.

Chapter 1 Introduction. 1

Chapter 2 Knowledge representation.

2.1 Introduction 3

2.2 Procedural and descriptive knowledge 4

2.3 A declaritive representation of knowledge 7

2.4 Rules : A procedural representation 16

2.5 Frames : a synthesis 21

2.6 The co-existence approach 26

Chapter 3 Organizing Knowledge for control of reasoning

3.1 Introduction 29

3.2 Backwards and Forward chaining 29

3.3 Control of reasoning 35

3.4 Control of retrieval 37

3.5 Explicit control of refinement 41

3.6 Agendas 44

3.7 Meta-rules 46

3.8 Explicit rule clustering 49

3.8 Conclusions 50

Chapter 4 Reasoning with non-monotonic logic

4.1 Introduction. 52

4.2 Reasoning about actions 55

4.3 Inheritance, defaults and exceptions 60

4.4 The closed world assumption 63

4.5 Circumscription 66

4.6 Uses of non-monotonic logic 67

4.7 Conclusions 68

Chapter 5 Truth Maintenance

5.1 Introduction 70

5.2 Data Dependencies 71

5.3 Justification based truth maintenance 73

5.4 Assumptions define contexts 81

5.5 Handling Contradictions 82

5.6 Other truth maintenance systems 86

5.7 Conclusion 90

PART2.

Chapter 6 Introduction for Part2 93

Chapter 7 Knowledge representation in the WISE system

7.1 Introduction 95

7.2 The Wise knowledge base 95

7.3 Inter frame relations 96

7.4 Own and Inherited Slots 99

7.5 Inheritance 105

7.6 Entry-conditions and Exit-actions 106

7.7 Rules 108

7.8 Conclusions 113

Chapter 8 Control of reasoning in WISE

8.1 Introduction 114

8.2 Forward and Backward chaining 114

8.3 Frame level control 119

8.4 A WISE consultation 123

8.5 Validity flags and Indices 139

8.6 Control Actions 140

Chapter 9 The Wise truth maintenance system

9.1 Introduction 141

9.2 The theoretical foundations for the WISE TMS 142

9.3 Choice and Assumptions 148

9.4 System defined choices 157

9.5 The justification network 158

9.6 Contexts as possible worlds 167

9.7 Resolving Contradictions 172

9.8 Inference engine-TMS interface 177

9.9 TMS-pointer-list and circularity 180

9.10 Conclusions 182

Chapter 10 Components of the WISE shell

10.1 Introduction 183

10.2 The Knowledge engineering environment 184

10.3 The compiler 192

10.4 The User interface 195

10.5 Conclusions 197

Chapter 11 Implementation Details

11.1 Introduction 198

11.2 The LISP programming language 198

11.3 Data Structures 200

11.4 Conclusions 208

Chapter 12 Use of the WISE system

12.1 Applications built on the WISE system 209

12.2 The chemical processing application 210

12.3 Conclusions 212

PART3.

Chapter 13 Related Work

13.1 Introduction 213

13.2 Nexpert Object 213

13.3 Expert System Environment 218

13.4 KEE 222

Bibliography 227

Appendix 1 Wise language and compiler output 237

Appendix 2 Specification diagrams 248

PART ONE

CHAPTER1 : INTRODUCTION.

An expert system is a program that represents domain specific and common sense knowledge; this knowledge is used to make decisions usually made by a human expert. Bruce Buchanan [Buchanan 86] , one of the leading researchers in the field of expert systems defines expert systems along 4 criteria :-

1) Artificial intelligence (AI) methodology : An expert system is a program that uses AI techniques such as: reasoning with symbolic objects and heuristic inference.

2) Expert level performance : An expert system should provide a performance equivalent to that of an expert. The expert system is, however, usually working over a much smaller domain.

3) Flexibility : AI programs should be flexible, both at design time and runtime.

4) Understandability : an expert system must be able to explain how it reaches a decision.

Research and development into the commercial possibilities of expert systems has accelerated dramatically in the last few years. After two decades of development and research, Artificial Intelligence (AI) technology is rapidly reaching

commercial thresholds [Williams 86]. Systems such as DEC's XCON configuration advisor for the VAX, and Westinghouse Elevator Company's system used for the deign of their elevators [Marcus et al 88], are commercial successes that have led to renewed interest in the field of expert systems.

In order to make the development of expert systems easier and thus less costly, researchers have been devising tools to assist the knowledge engineer in this task. These tools are called expert system shells. The research contained in this thesis was directed towards the development of a commercially viable expert system shell. This shell has been developed under the name WISE (Wolf Intelligent System).

This thesis concentrates on several important issues in expert system research, namely :

- representation of knowledge
- control of reasoning
- implementation of non-monotonic logics via truth maintenance systems.

There are three parts to this thesis. PART1 covers the background research in the above mentioned topics. PART2 discusses the WISE system and the way in which research from PART1 was applied to the development of the WISE shell. PART3 considers the features of other expert system shells.

CHAPTER2: SYMBOLIC STRUCTURES FOR KNOWLEDGE REPRESENTATION.

2.1 Introduction to Knowledge Representation.

A computer system's solution to a problem depends very much on the manner in which the problem is represented. A good representation of knowledge can greatly simplify the reasoning process on that knowledge eg. the representation of numbers as arabic rather than Roman numerals greatly simplified the tasks of multiplication and division.

Any knowledge representation system provides information about some perceivable reality . The process of perception consists of distinguishing objects and their properties (attributes) [Orlawska & Pawlak 84]. Perceiving attributes consists of assigning values to them.

A measure of the power of a representation is the distinctions it can make and the distinctions it can leave unspecified to express partial knowledge [Woods 1983]. A natural language is expressive if it can make distinctions eg. in Zulu each pattern of markings on cattle has its own word. Thus Zulu can make this distinction while English requires the distinction to be made in terms of other descriptors such as color or pattern eg brown speckles.

Another important measure of the effectiveness of a representational scheme is how the structure of the representation affects the computational operations of the system, and how easy it is to modify [Woods 1983].

Most expert systems are very domain specific and it has been observed in AI that expertise in a task domain requires substantial knowledge about that domain. One of the crucial factors in the design of an expert system is the effective representation of the task domain. As previously inferred the representation for such information must have expressive power, understandability and accessibility [Fikes and Keller 85].

This chapter will consider several representations that address these issues. However, before looking at particular representations, an important distinction must be made between the types of knowledge we wish to represent.

2.2 Procedural and Declaritive knowledge.

Their are 2 types of knowledge that must be expressed:

- The declaritive knowledge is the description of the application area ie what is known.
- The procedural knowledge tells the system how to use the declaritive knowledge ie how to solve the problem.

Different representations emphasize the different types of knowledge, although all systems must obviously have both. Their is wide spread disagreement on whether the emphasis in knowledge representation should be on declaritive or procedural knowledge.

Proceduralists claim that the core of our knowledge is "knowing how". In other words the human inference process uses knowledge expressed in programs. All the information needed for a particular task (such as speaking english) co-exists as part of the knowledge on how to use it [Winograd 85].

The declaravists on the other hand say that knowledge can, and indeed should, be separated from how it is used. They believe the core of our knowledge is represented by sets of facts describing particular domains, and very general procedures for using this knowledge.

Both approaches have merits. The declarative approach allows a piece of knowledge to be specified without explicitly specifying how it will be used. Declaritive knowledge also tends to be easy to understand and modify.

Procedural knowledge, on the other hand, is often the natural way in which we think of processes. It is often easy to describe a particular problem that involves domain specific actions, such as getting a robot to manipulate objects in a simple world, in a procedural form. Procedural knowledge also allows one to express knowledge about knowledge. The most

important advantage of procedural over declarative representations is that specific and general heuristic and control procedures can be easily specified procedurally [diagram 2.1]. This aspect of reasoning will be discussed in detail in the chapter3.

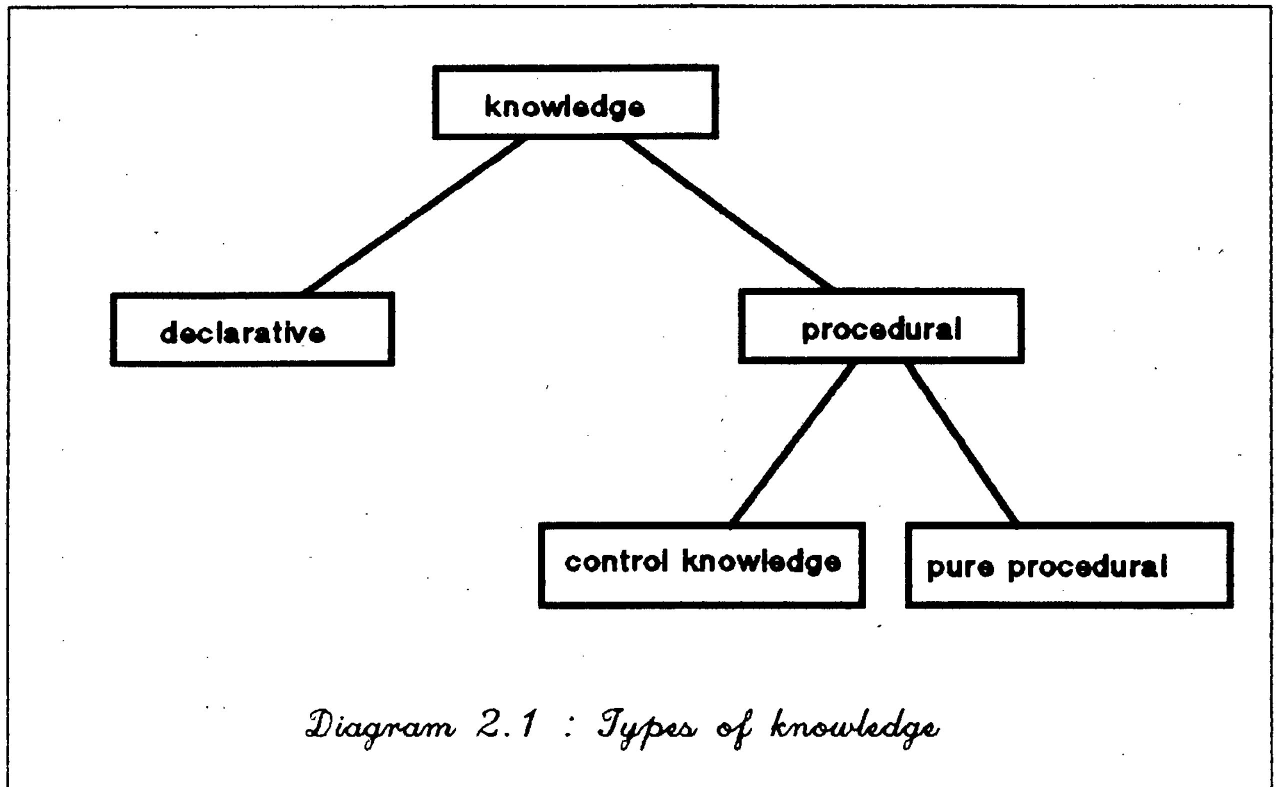


Diagram 2.1 : Types of knowledge

From a very basic point of view there is no difference between these representations [Winograd 85]. Any program can be viewed as data that is used by the program hardwired into chips of the computer. In this extreme view everything is declaritive. On the other hand any fact could be viewed as a program that responds to inputs like "are you true" or "assume you are true" [Hewitt, Bishop & Steiger 73].

The point is not whether a piece of knowledge is a program or a statement but on how this knowledge is viewed by the system

using it. This chapter will consider several representations which are viewed primarily as declaritive (logic and nets), or procedural (rules). It will then consider a representation that attempts to capture the best of both views (frames). Finally it will consider a co-existing approach to knowledge representation.

Many of the early AI systems, such as GPS [Simon and Newell 63], used general procedures with domain specific declaritive knowledge to solve problems. However it soon became clear that in order to solve real world problems the system needs a high level of domain specific control (procedural knowledge) [Kunz, Shortliffe et al 84]. This domain specific knowledge is expressed procedurally in the knowledge base and is separated from the inference engine that uses it.

2.3 A Declaritive representation of knowledge.

2.3.1 Logic Programming

A simple and powerful way of representing declaritive knowledge is using formal logic. The simplest form of logic is the propositional calculus. This logic consists of statements, called propositions, which can be either true or false. More complex propositions can be built up from simple

ones by using the operators : and, or, not, implies and equivalent [Jackson 83].

When variables are added to the propositional calculus it becomes the predicate calculus. This logic system consists of predicates which are functions that return either true or false. These functions may have any number of arguments.

[Diagram 2.2a] is an example of a logic program. The first 6 statements are facts about the domain. The last statement is an implication which means :
x is the teacher of y is implied by x is a teacher and x is qualified in z and z is the subject of y.

```
is_a(fred,teacher)
is_a(bob,teacher)
qualified(bob,db)
qualified(fred,ke)
subject(course-1,db)
subject(course-2,ke)

teacher(X,Y):-is_a(X,teacher),
              qualified(X,Z), subject(Y,Z)
```

Diagram 2.2a : Logic Program.

```
Is fred a teacher? (yes)
Is sally a teacher? (no)
Who are all the teachers? (fred,bob)
Who teaches course-1? (bob)
Does Bob teach course-2? (no)
etc etc
```

Diagram 2.2b: possible inferences.

These seven facts imply a wide range of facts as shown in diagram [2.2b.]

If this knowledge was represented procedurally every use of it would have to be separately specified. This is because descriptive formalism do not specify how the knowledge will be used. The knowledge need only be described and it can be used in multiple ways.

PROLOG is a language based on a subset of predicate calculus known as the Horn Clause subset. It allows the programmer to express facts and implications.

The PROLOG interpreter provides the operations needed to make the Horn Clause subset complete. Completeness means that if some proposition "P" follow from some set of propositions, S , then S=>P can be proved [Genesereth and Ginsburg 85].

The key idea in logic programming is programming by description. Instead of supplying an algorithm to solve a problem, the application area is described and a general purpose algorithm uses these facts to solve the problem. In PROLOG this general purpose algorithm is called the interpreter. The assumptions (truths) are explicit (stated), but the choice of operations performed by the interpreter, is implicit.

The interpreter uses the facts in its knowledge base (description) to solve problems. The logic of the problem is encompassed in the knowledge base, thus a good description will lead to a good program. As new knowledge about the

problem domain is encountered it can be added without necessitating a change to the other knowledge or the inference procedure.

PROLOG programs can explain their reasoning. This is done by saving the steps of the inference process. By presenting this information to the user, he can see how the problem was solved. These explanations (or traces) are used to debug logic programs [Genesereth & Ginsburg 85].

The main reason people use PROLOG to develop expert systems is the well defined mathematical foundations of logic. Logic programming also allows for incremental definitions. Facts can be easily stored and updated.

Knowledge bases using pure logic to represent their domain knowledge tend to be very difficult to read, and thus difficult to maintain. Because the representation is declaritive it is difficult to represent knowledge that guides the reasoner to a solution.

2.3.2 Semantic Networks.

Another popular descriptive representation of knowledge is the semantic network formalism.

Semantic nets [Quinlan 66] are a general associative mechanism for representing words and their meanings. This representation is rich enough to express natural language or human "semantic memory" yet general enough to be acted on by general procedures [Brachman 77]. Although originally designed for natural language processing the associative network idea was quickly applied in other areas of AI.

The semantics of a representation specifies how meaning is embodied in the symbols, and the symbol arrangements, allowed by the syntax. The syntax of a semantic net consists of objects, and the relationships between pairs of objects.

The semantic net representation can be thought of as a way of representing predicate logic. Terms are replaced with nodes and relations are replaced with labeled directed arcs [Charniak and Mcdermott 85]. The semantic net only represents binary and unary relationships. However this is not a fundamental restriction as any logical statement can be represented in that way.

In its simplest form a semantic network has nodes which represent objects and arcs between two nodes that represent relations. The relations most commonly depicted are of the taxonomical type such as "IS A" but theoretically any binary relation can be depicted by an arc.

Nodes are places at which knowledge is stored about particular things in the world [Brachman 85]. Typically there are:

- nodes for objects eg the person John would be depicted by a node
- nodes for factual assertions
- nodes for events.

Nodes are also used to represent a grouping of other nodes into classes. These nodes are called CONCEPT NODES. They allow information about objects to be 'clustered' together. The result is a hierarchical structure which allows for inheritance of properties. Concept nodes express the general nature of a class of individuals. They do this by extracting the common attributes from a group of objects. Concepts are linked together by a generality relation so that a concept more general than a given concept can be accessed from it.

Two types of statements that AI knowledge representation systems wish to express are:

- universal quantification (that one type is a subtype of another)
- predication (expressing that an individual is of a given type [Brachman 83].

The IS_A link was devised to express both of these concepts. This use of the same link to express two very different statements has led to confusion as to exactly what an IS_A link is [diagram 2.3].

The attributes of concepts are stored using the same mechanism that is used to store attributes of an object. An attribute can be thought of as a named link or the name of a field in a

record. Such a link can be thought of as a concept in its own right.

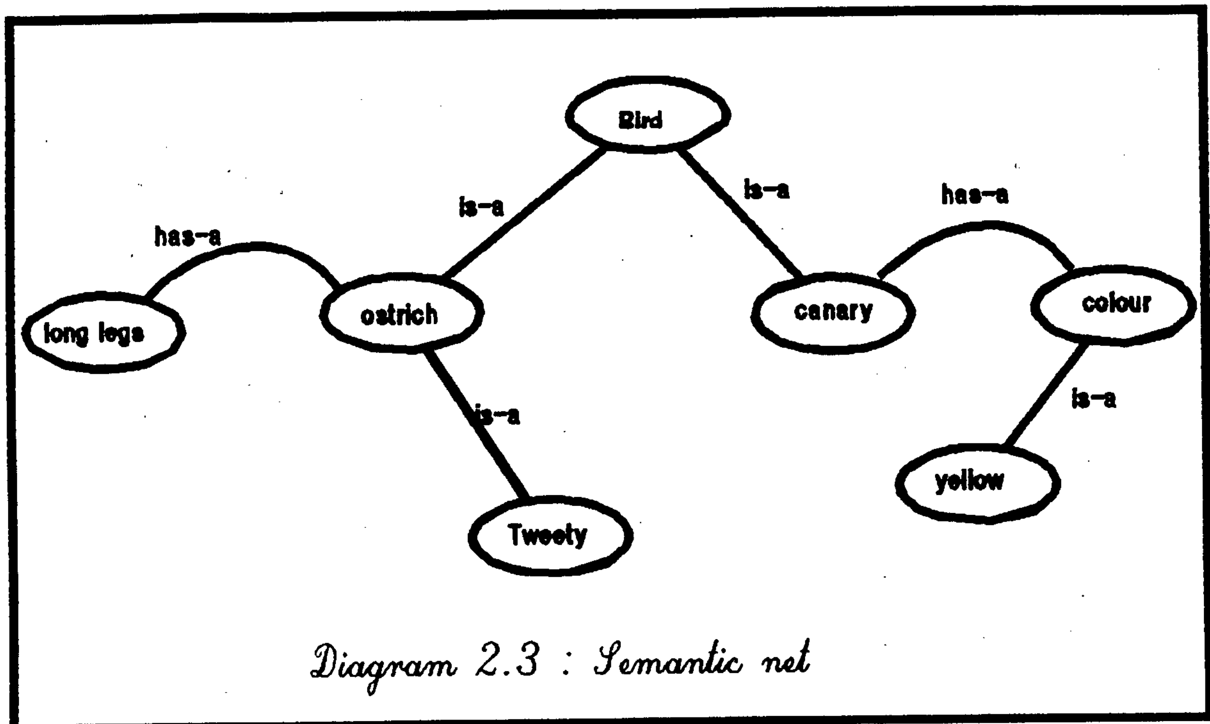


Diagram 2.3 : Semantic net

A problem arises because the link between a particular individual and the value of its attribute is indistinguishable from the link between a concept and its attribute's value. However one explicitly asserts something about a single object and the other something about a group of objects [Brachman 77]. This indistinguishability makes the writing of general procedures for using this knowledge more difficult.

By joining concepts and objects to each other with the IS_A link, a taxonomy is formed. Whether or not an item is a member of given set or type is of central relevance in question answering and fact retrieval. The taxonomical structure thus provides a natural and concise expression of a large part of the information about the domain [Hendrix 79]. A further advantage of the hierarchical structure is that information can be stored at its most general level of

applicability and indirectly accessed by more specific concepts said to inherit that information. The fact that members of sets often have common attributes means that these attributes can be stored at the set level and inherited by each member [Woods 86].

The knowledge contained in the semantic net is purely descriptive. The knowledge needed to use it is contained in general access routines. The procedural knowledge is implicit while the descriptive knowledge is explicit [Hendrix 79].

2.3.3 Problems with descriptive representations.

Both logic programming and semantic nets suffer from some problems which have limited their use as a general representation for expert systems. These problems stem from the fact that these representations have no way of expressing explicit domain specific procedural knowledge.

A knowledge representation scheme should be able to represent knowledge pertaining to how other knowledge will be used. Furthermore this knowledge should be expressed in a programmatic way as this is the natural way to describe such a process.

Semantic nets represent descriptions of a domain. This description is used by general (not domain specific) procedures [Rich 83]. A common accessing method is spreading activation or intersection search. This involves following lines out of two nodes and seeing where, or if, they meet. Semantic nets are heuristically inadequate [Jackson 83] in that they have no knowledge about the manner in which a search must take place. This lack of control is a major issue that we will discuss in the next chapter.

Logic programming also has no facilities for expressing local heuristics for directing the search. PROLOG uses an unintelligent exhaustive depth first search to find the solution to a question.

Another problem with both semantic nets and logic programming is the representation of negation. Negation is treated as failure to prove [Winograd 80]. This means that these systems are really negationless calculi ie. when asked for instance whether "Tweety is-a dog" they do a complete search of all the facts they know. If this fact is not there then they assume it is false. This operation is expensive in large domains.

Very few expert system builders would use PROLOG to build a large expert system however, as a programming language, PROLOG is very useful for developing systems that use better representations. Indeed many of the expert system development techniques discussed in this thesis could be implemented in PROLOG.

2.4 RULES : A Procedural representation.

Procedural knowledge is most commonly expressed in productions or rules. The rule based representation has its origin in production systems [Post 43]. Production systems represent knowledge procedurally. By this we mean that a piece of knowledge is specified in terms of its use.

Rules have been used to represent the knowledge of Expert systems with great success. They are easy to update, modular and uniform [Davis et al 85]. They are also a natural way of expressing procedural and control knowledge. In rule based systems the domain knowledge is embedded in procedures (rules).

Because rules are uniform they can be understood by different parts of the system. This allows for automatic modification of the rules by the system. Rules have proved to be a natural way of expressing an expert's knowledge. Because of the English-like structure of the representation, the expert can input his knowledge directly into the system.

The three components of simple (non learning) RBSs are the rule base, fact base and the inference engine. A rule consists of a set of antecedents and a set of consequents. The

antecedents are usually object-attribute-value triples. These triples are true if the attribute of the object is equal to the value. The inference engine executes a rule by first checking if all the antecedents are true and if they it then executes the consequents. If a consequent defines an action then the inference engine performs it.

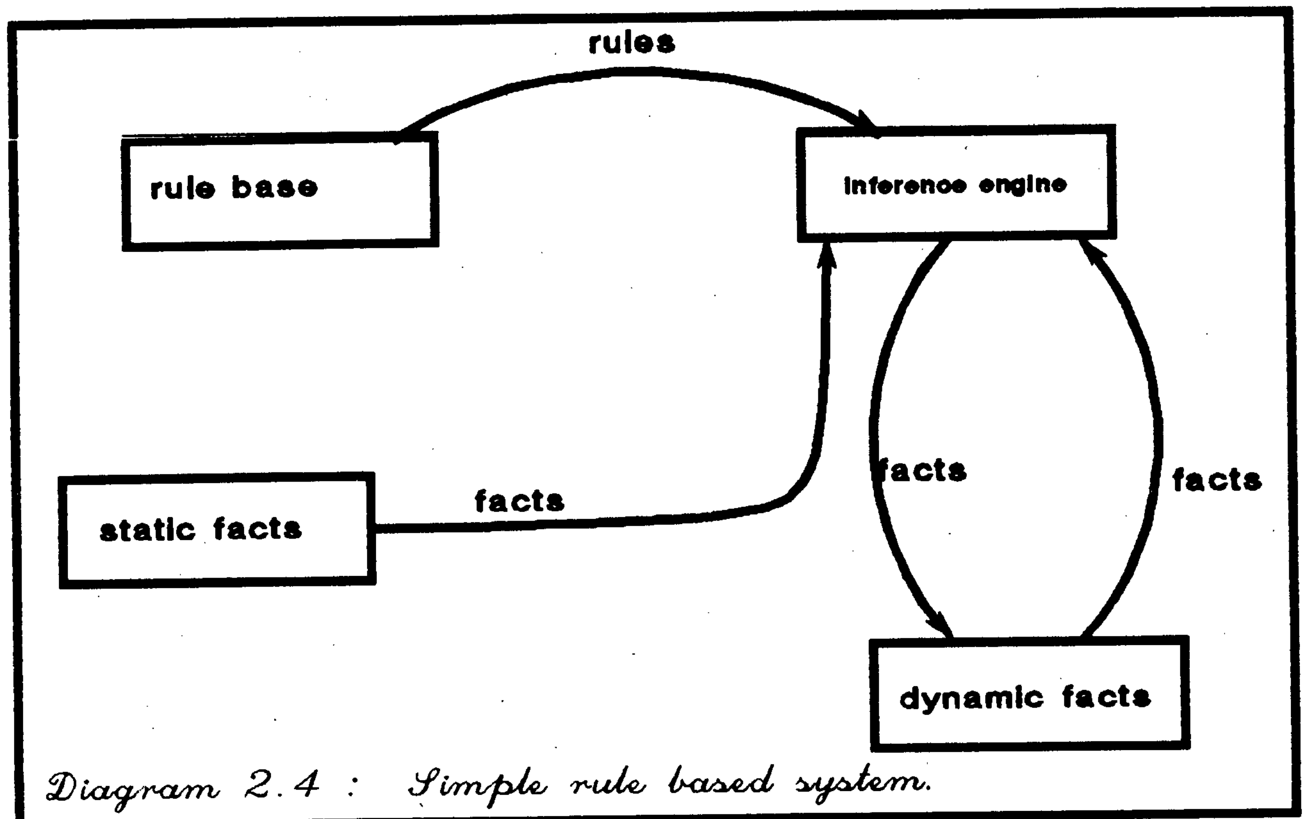


Diagram 2.4 : Simple rule based system.

Facts express assertions about properties, relations, propositions etc. They too are in the form of object-attribute-value triples (O.A.V). Apart from the original static information (set up by the knowledge engineer) there is also dynamic information that is gleaned by the system as the inference process is executed. This is stored in the WORKING MEMORY. This working memory is often referred to as problem-

solving state information [Hayes-Roth 85] and is in the form of facts. Thus the environment that the inference engine runs in consists of the rule-base and static facts plus the dynamic facts, all these constitute the context for interpreting a rule [Diagram 2.4].

An important part of an inference engine is the rule interpreter, or pattern matcher. This matches a rule component with the problem-solving state information. In its simplest form it will take the O.A.V.s that make up the antecedents and see if these O.A.V.s are in the working memory. However nearly all systems allow variables which increase the complexity of the pattern matcher. Different systems use different methods to simplify the pattern matching task. If all the premises are true the inference engine executes the consequent by either changing the working memory or performing some control action.

The ability to easily describe local heuristics is probably the most important advantage of procedural representations. These heuristics guide the search for a solution. The next chapter will consider the control of search in reasoning.

Good explanation features are one of the most important criterion for a successful expert system. Rules allow a very simple explanation facility to be easily implemented. By showing the user the rules that have fired he can follow how a conclusion was reached. However this is the least satisfactory type of explanation a system can generate [Hayes-Roth 83]. By

building simple natural language interfaces rule based systems can offer better explanations of their reasoning.

MYCIN [Kunz, Shortliffe et al 1984], one of the first large successful expert systems, was built using rules as a representation. The knowledge of the system is incorporated in conditional if-then rules and facts which state what is true. Mycin's basic task is to act as a consultant in the identification of organisms causing infection, and choose the appropriate drugs for treatment. MYCIN's design had certain constraints, these were :

- the system needed to deal with a large and constantly changing body of technical knowledge which is task specific
- the system had to handle interactive dialog in real-time
- the system had to be useful. In order for it be useful it had to give expert like advice in its domain.

2.4.1 Problems with rules as a representation.

The features of modularity and uniformity have problems associated with them. Because modularity is a necessary feature, for the incremental improvement of rule-based

systems, each rule must be independent of all the other rules. This independence means that the entire context of the rule must be stated explicitly in the premises, often leading to long, unwieldy and difficult to understand premises.

A further problem is that because the rules are independent of each other, there can be no groupings of rules that apply in certain situations. Without these groupings, adding or modifying rules could have an effect on the other rules which are difficult to predict [Aikins 83]. This problem has been addressed by meta-rules and other control strategies which will be discussed in the next chapter.

Uniformity of syntax often forces different types of knowledge, such as control knowledge, to be expressed in the same structures as normal procedural knowledge. Rules can contain many different types of knowledge, these include: specific inferences, categorization of given data, the necessary or sufficient conditions for achieving some goal as well as control strategies. Thus information pertaining to both the inferences and control knowledge can be expressed as rules; the function of the knowledge is not immediately clear and this can lead to extra complexity in the inference engine and explanation generator.

2.5 FRAMES: synthesis of procedural and declaritive knowledge.

It should be clear that both procedural and descriptive representations are needed to fully describe an expert's domain. Frames are an attempt to synthesize the best of both worlds. Frames supply a semantic net type hierarchy for declaritive knowledge. The nodes in the hierarchy have slots which are like fields in a record. Procedural knowledge on how to use the slots is attached to them. Thus they provide a generalized hierarchy as well as procedural knowledge for directing how this descriptive knowledge will be used.

The frame approach [Minsky 81] attempts to organize knowledge in collections of simple and separate pieces. Many of the ideas upon which frames are based came from psychology which has long hypothesized that knowledge is stored in chunks in the brain.

Frames were originally developed specifically for computer vision or perception, however they have since been used in other fields of AI. Minsky believed that when one encounters a new situation, one selects from memory a substantial structure or piece of information called a frame. A frame is a record like structure with slots and fillers. The slots are used to store the various attributes of the object. Slots are filled with values which are called fillers. Slots express binary relations between the frame and the values.

Frames are remembered situation that can be adapted to fit reality by changing details as necessary. The basic philosophy is that reasoning is dominated by a process of recognition in which new objects and events are compared to a stored set of expected prototypes [Bobrow & Winograd 85]. Frames incorporate the basic tools of semantic nets and add the idea of standard stereotypes.

Frames that represent classes or concepts have the common attributes of the members of the class ie the ones that hold good for the majority of cases. Each member of the class is stored beneath the frame representing the class, these instantiated frames deform the stereotype frame to capture the complexities of the real world including exceptions [Minsky 81]. Thus a frame representing a class of objects will have all the attributes that are common to the members. The members will have all the ways that they differ from the standard.

A Frames hierarchy can be thought of as having different levels. The top level is fixed and represents knowledge that is invariant for all instances of that typical situation. At a lower level are the slots which are filled by specific instantiations of data that relate to a particular instance of the prototypical situation. Lower frames inherit attributes from frames above them in much the same way as semantic nets. The inherited values can be overridden in the lower frame. A generic frame has 2 types of slots; some that apply only to

the generic group, and some that are inherited by the lower frames.

One interpretation of the frame structure is that they are bundles of properties [Hayes 85]. These properties are stored in slots. Much of the power of the frame representations hinges on the inclusion of expectations. The fact that a slot occurs in the frame at all is in itself useful knowledge eg. a frame for earthquake has slots people_killed, Homeless etc. This means that should an earthquake occur people are expected to be killed or left homeless [Winston 84]. Thus what we expect to know is explicit. Since the system knows what attributes an object should have, it can judge to what extent a solution is complete [Aikins 83].

Slots can be linked to a default value set by the knowledge engineer. These defaults can be overridden during the course of a knowledge base consultation. The defaults allow the system to reason with incomplete knowledge. Default assignment to slots is a common feature of frame systems. It allows them to say in the absence of information to the contrary conclude [Reiter 85]. These types of inferences are called non-monotonic inferences and I will discuss their uses will be discussed in chapter 4. In general the default is assigned only as a last resort ie. if the other options to determine the value should fail.

A default reasoning system has to take into account that in different situations an object could have different

interpretations. Frames can have multiple parents and it is not always clear which default value must be inherited.

A default system should allow for inheritance with exceptions. Exceptions are essential for representing real world knowledge such as "almost all". Common sense knowledge relies on the ability to make general statements with exceptions [Nado and Fikes 87]. These features of default reasoning are discussed in the chapter 4.

It was realized that semantic nets were lacking in two main areas. These were :

- they were logically inadequate in that there was no distinction between a node representing, for example, a generic truck and a node representing a specific truck.

- They were heuristically inadequate because their inference techniques were not knowledge based ie there was no procedural knowledge in the system directing the search for the goal.

Thus a new representation was needed that allowed the procedural and descriptive knowledge to be stored as one entity.

Frames do not only contain a description of an object but also knowledge on how the knowledge is to be used. Knowledge about knowledge is called control knowledge. In order to fill the slots with information, procedures can be attached to each slot telling it how the information can be received and what values to expect. Because the control knowledge is in the

frame one has context-sensitive control information [Aikins 84].

Procedural information is attached to the slots. This knowledge is executed when a value is accessed. This includes:

- a method for finding the value of the slot
- an action to perform when a value is added
- an action to perform when a value is removed

Context specific control greatly increases the efficiency of the inference procedures used to access knowledge. The inference process can be made to focus its attention by looking only at relevant information. This method of control also allows better explanation facilities.

By storing the knowledge in a frame network it is effectively partitioned . At each level the frames represent a partial ordering of knowledge. When a consultation is in progress, the inference process need not worry about the entire knowledge base but can concentrate on the relevant section.

Internist [Miller 84] is a expert system that uses frames to represent its domain knowledge. Internist is the widest ranging of the medical expert systems to date. Because it covers such a wide domain Internist uses frames to partition the knowledge. The system attempts to assist with diagnosis and selection of relevant investigations for diseases in internal medicine (non surgical).

The program uses a hierarchical disease categorization to store the different diseases. The knowledge base enumerates relationships between diagnoses and manifestations via disease profiles ; which is a list of findings which have occurred in given patients with a certain disease [Miller 84]. Internist has about 600 of these disease profiles and the system can recognize up to 4000 manifestations of disease.

Internist uses 'clever heuristics' [Kunz, Shortliffe et al 84] to focus attention, decide between competing possibilities and diagnose multiple diseases. In tests INTERNIST has shown to be about as good as an average physician but not as good as an expert physician.

2.6 The co-existence approach.

In the preceding sections the different knowledge representation paradigms, used in expert systems, have been described. Each representation has advantages and disadvantages. It is the domain of the expert system being developed that will influence the knowledge engineer in his choice of representation.

The representations used in expert systems have simplified the task of capturing knowledge in a certain domain rather

than addressing the problem of finding a universally applicable form of knowledge [O'hare & Bells 85]. It has been recognized that generality will not be achieved by one all encompassing knowledge representation, but rather through the co-existence of the currently used formalisms [Takenouchi & Iwashita 87] and [O'Hare and Bells 85].

Because of the wide range of knowledge domains researches are looking at the possibility of using a synthesis of two or more of these representational paradigms to model domains. The idea is not only to supply tools with a choice of representations, but to allow the representations to be mixed together.

The idea of using an amalgam of two or more representational paradigms was implemented in an expert system called CENTAUR [Aikins 83]. Centaur is a consultation system that uses knowledge about prototypical situations to guide its performance and explanation facility. The task domain is pulmonary physiology.

CENTAUR uses frames called prototypes. These frames contain procedural knowledge in the form of production rules. It was found that this representation was useful in allowing the system to control a consultation. Because a frame is only accessed in a particular context, the expert can specify an action (via rules) in a given context. The idea of combining representations is fairly new and researchers believe that it will increase the representational power of these systems.

The designers of CENTAUR believe that their chosen representation allows them to express all the knowledge necessary to perform pulmonary consultations. The rules are clustered (in frames) according to their function in a consultation; this is done by attaching them to slots in the frames. The designers also found that this representation allowed for much better explanation facilities than just plain rule-based or frame-based systems. This is because the prototype explanation in CENTAUR provides a broad context in which to view the rule explanation. This principle is found in all parts of the system, where the broader issues are dealt with at the level of frames, while the more fine reasoning is done at the rule level.

The CENTAUR system has demonstrated that using a co-existence approach to knowledge representation can increase the power of an expert system; over one built using a single representational paradigm.

CHAPTER3 : ORGANIZING KNOWLEDGE FOR CONTROL OF REASONING.

3.1 Introduction

The previous chapter looked at how knowledge is represented in an expert system. Reasoning is the process of using this represented knowledge to solve problems.

Knowledge organization is the manner in which the knowledge is stored, or viewed, by the problem solver. Reasoning can be viewed as search of the problem space ie the space of possible answers. This search must be guided by global and local heuristics, this is known as control of reasoning. This chapter will consider methods that have been used to control the order of invocation of procedural knowledge especially rules.

3.2 Backwards and Forwards chaining

The two most common control strategies used by expert systems are forward and backward chaining. Forward chaining is working from our known facts to try and find a goal, and is thus called data driven reasoning. Backward chaining is

working back from the goal to see if our premises would imply it, and is thus known as a goal-driven strategy.

In terms of rule-based systems, forward chaining consists of firing a rule when its antecedents' components match the problem-solving state information (set of known facts). In the backward-chaining, or goal directed inference technique, the inference engine starts with the goal and attempts to fire each rule with matching consequent components. The unmet conditions of the rule are treated as subgoals. In this way the system attempts to backwardly chain through the rules.

The execution of a set of rules can be viewed as a search of an and/or tree [Winston 1984]. Consider the set of rules in [diagram 3.1a].

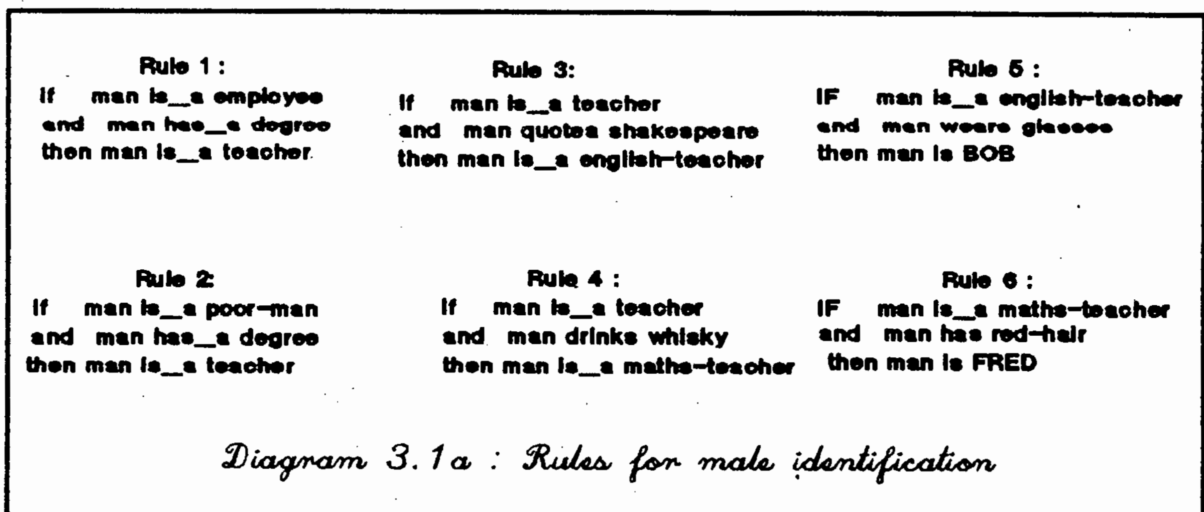


Diagram 3.1a : Rules for male identification

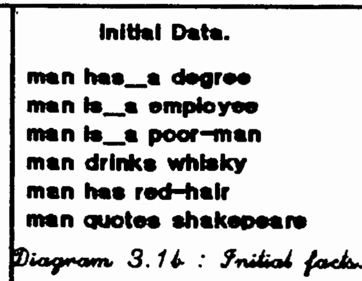


Diagram 3.1b : Initial facts.

In forward chaining systems a search tree is built by taking the initial configuration of data [diagram 3.1b] and finding all the rules whose LEFT HAND SIDE matches this, in this case rule-1 and rule-2. The RIGHT HAND SIDE of these rules (man is_a teacher) is then used to generate new configurations. These configurations form the next level of the and/or tree [diagram 3.2]. The subsequent levels are generated from the previous one by taking each node and finding all the rules whose LEFT HAND SIDES match that node's configuration. Continue generating nodes until the GOAL STATE is generated [Rich 1983]. In this example the goal is a positive identification of the man.

In backward chaining systems a search tree is built by making the goal the root. The next level is generated by finding all the rules whose RIGHT HAND SIDE matches the goal, in this case rule-5 and rule-6. The LEFT HAND SIDES of these rules make up the next level. Each subsequent level is generated by taking the nodes of the previous level and finding all the rules whose RIGHT HANDS' SIDES match them and using these rules LEFT HANDS' SIDES to generate the next level. Continue until a node that matches the INITIAL state is generated, in this case when we reach rule-1 or rule-2.

These six rules are turned into the and/or tree represented in [diagram 3.2]. In this diagram the solid dots represent inferred facts and the open dots represent the given facts. Facts that are used in more than one rule are represented

separately. The dotted line joining the two B dots (which represent man has-a-degree) signifies a choice of rule to fire. When more than one line comes into a solid dot these facts all have to be true for the dot to be inferred.

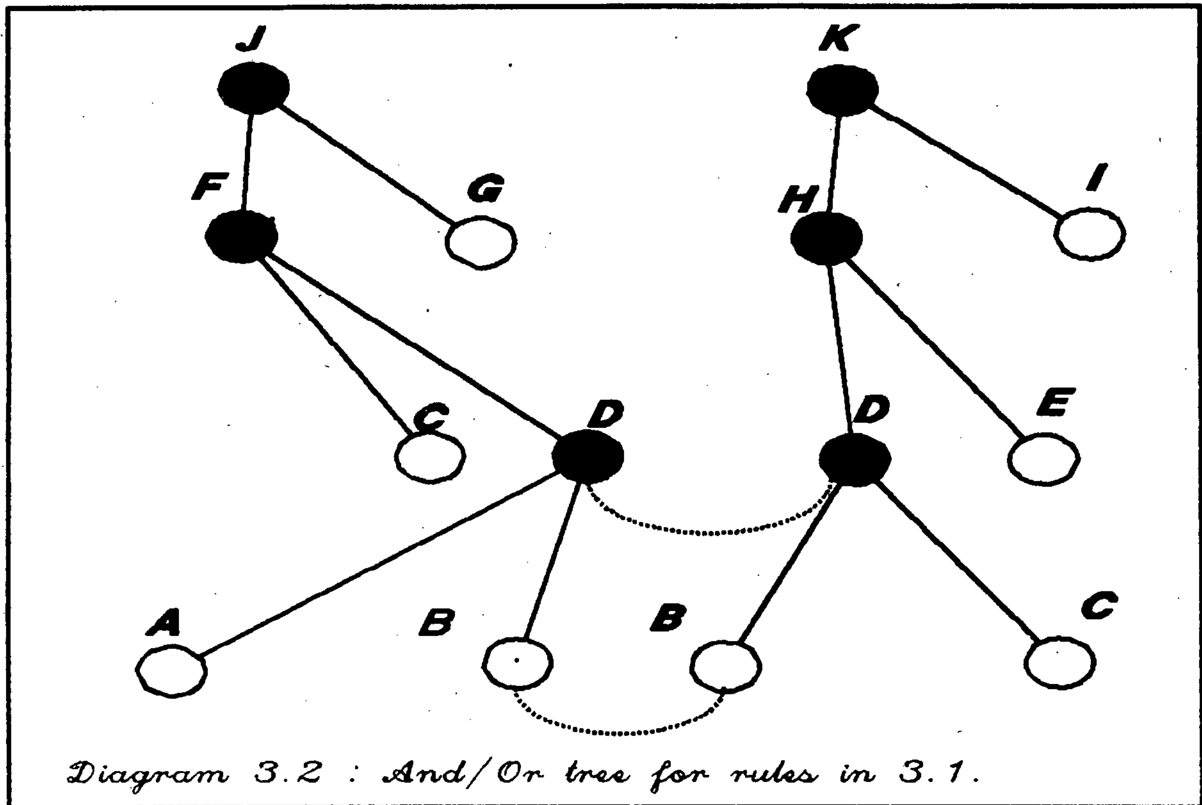


Diagram 3.2 : And/Or tree for rules in 3.1.

A = man is-a employee

B= man has_a degree

C= man is_a poor-man

D= man is_a teacher

etc.

Since the inference procedure can be viewed as a search of the above tree, the simplest way to describe the different methods such processes employ, is in terms of search

strategies. The 2 basic strategies are depth-first or breadth-first.

In forward chaining systems with a breadth first strategy each rule that is initially true is fired and all the new configurations of working memory are stored. For each of these new configurations the rules that are true are fired resulting in the next level. This continues until one of the configurations contains the goal state. This is very expensive if all the configurations are stored at each level therefore only the deltas would normally be kept. In the example where rule-1 and rule-2 are both valid, the system would execute rule-1 and store the new configuration. Then using the initial configuration it would execute rule-2, generating another configuration (in this case the two second level configurations are the same). Each of these configuration is used to generate the next level and so on.

A forward chaining system using depth first search will start at the initial working memory and fire **one** rule that is true, which rule to fire is decided by using a conflict resolution strategy (discussed below), either rule-1 or rule-2. The new configuration of working memory is now used to find a new set of valid rules (rule-3 and rule-4), one of which is fired. This continues until either the goal is reached or there are no rules to fire in a configuration. Should the situation arise where there are no fireable rules, the system must backtrack to a previous configuration and fire another possible rule.

If rule-1 is fired then there is a possibility of firing rule-3 or rule-4. If rule-3 is fired and it turns out that man does not wear glasses represented by node G in [diagram 3.2], then we cannot reach a goal, so we backtrack to where we made our last choice (node D), and fire rule-4, which infers H (man is_a maths-teacher) , rule-6 is then fired and the solution is found ie man is FRED (node K).

In a backward chaining system using the breadth first strategy all the rules whose RHS contains the goal are tried. At each level all the rules are tried and each rule's antecedents are set as subgoals. This strategy is not commonly used.

A backward chaining system using a depth first search strategy starts at the goal and chooses **one** rule with the goal on the RHS treating this rule's LHS as subgoals. It then finds one rule with each of these subgoals on the RHS and treat these rules' LHS as subgoals. It will continue doing this until all the subgoals are in the working memory or it fails. If it fails it backtracks ONE level and chooses another rule.

If the goal is find out if the facts indicate that the "man is FRED". Then the system attempt to fire rule-6, which causes the fact man is_a maths-teacher to become a subgoal. This causes us to attempt to fire rule-4 etc until we get to rule-1 or rule-2 for which the facts are available.

The advantage of a breadth first strategy is that it is complete and it finds the 'closest' solution. The disadvantages are the considerable intermediate storage needed to store each level's nodes. It is also potentially expensive on broad trees [Hayes-roth et al 1983].

The advantages of depth first search are that it is easy to implement and uses minimal space. The disadvantages is that it does not find the closest solution and can be expensive on deep trees.

3.3 Control of reasoning

Control over the way in which the knowledge pieces (rules) are used is an important feature that any expert system shell should provide. In the example above the tree search is blind and exhaustive. What is needed are some heuristics to control the search ie enable the system to find a goal without doing an exhaustive search.

Control mechanism could be global (applying to the entire tree eg. forward or backwards chaining) or they could be local. Local control mechanism or heuristics can be stored at a point in the search tree and used to give advice about which path to take. Thus instead of merely having clever

search algorithms one can have a smart tree. Each point of the search tree can be viewed as a point in which we have to decide what rule to use next (which possible path to take).

Using a piece of knowledge such as a rule involves three steps [Davis 80]: the first step is **retrieval.** Some subset of the knowledge base is chosen as a set of plausibly useful rules. Next this set of rules (called the conflict resolution set) is **refined.** This involves the set being pruned or re-ordered to give control over the order of execution. Finally the rule is **executed** and the knowledge base is updated or control is passed elsewhere.

All the plausible rules can be retrieved and tried if the rule base is small enough. However, as the knowledge base grows, it reaches a point in size when this stops being feasible. Most AI problems are often faced with many plausible inferences to perform and to exhaustively consider each in turn is not possible. The problem of being faced with too many plausible inferences is known as **saturation** [Davis 80]. The system needs strategies about how to focus quickly on the relevant categories of rules.

An important criterion for a successful control mechanism is that the control knowledge must be understood by the same inference engine as the object level knowledge. Reasoning about control is viewed as a problem solving task to which we can apply AI techniques. The control knowledge should

not be implicit in the interpreter, as in PROLOG, but explicitly represented and accessible to the program itself.

The knowledge representational paradigm must supply a way of encoding a higher level knowledge about knowledge. Knowledge that is concerned with the manner in which other knowledge is used is called control or meta-knowledge.

3.4 Control of retrieval.

A possible solution to saturation (being faced with many possible rules) is to supply increasingly specific pre-conditions or if-parts. Thus only a few rules (with all if-clauses true) will be plausibly useful in any situation. The problem with this is that, in a large knowledge base with N rules, when adding the $N+1^{th}$ rule the knowledge engineer has to compare it in detail with all the other N rules to ensure its preconditions were specific enough to facilitate retrieval of a small rule set.

A better approach is to use mechanisms that limit the amount of rules considered relevant at each point in the search. The rules are indexed according to when they might be relevant. This approach is followed in GPS's [Newell and Simon 1963] means end analysis and KRL's [Bobrow and

Winograd 1985] procedural attachment. These mechanisms try to ensure that retrieval is sharply focused.

3.4.1 Means-ends analysis:

One way of controlling the search is to control the rules that are retrieved. In a system using backward chaining rules that would solve some goal are retrieved. In a forward chaining system some rule property such as name, or pattern is used to select a set of plausibly useful rules from the knowledge base. What is needed is the ability to install a more precise method of retrieval ie the system should have some control over the retrieval.

One of the earliest approaches to control of retrieval was means-ends analysis which was implemented in GPS and STRIPS [Fikes and Nillson 71], both of which were early search programs that greatly influenced AI.

Means end analysis involves establishing the difference between the current state and the goal state and then attempting to find operators (STRIPS and GPS rules were called operators) to span that difference. This was done using a domain based function to calculate the difference between any state and the goal state. GPS thus used the

rule goal as an index to select whether the rule was plausibly useful in a given situation.

Systems such as STRIPS and GPS had a single evaluation function embedded in the interpreter. This function, which is domain specific, is responsible for choosing which operator to use at any point in the search. The had to be specially written for each application. The rule retrieval function was global and didn't allow for local heuristics that would vary with the context of the search.

3.4.2 Procedural attachment.

In the section on the frame representation the concept of procedural attachments was discussed. Procedural attachments were implemented in KRL [Bobrow and Winograd 85]. The designers of KRL viewed each step in the problem solving task as the application of operations to data objects. They decided to attach these operations to the data items being operated on. Many of the ideas of associating procedures with objects, or classes of objects, came from object orientated languages such as SMALLTALK [Learning Research Group 76].

KRL associated procedures with slots in frames. It also allowed objects to inherit these procedures in the same way

that they inherited declaritive properties. The procedures are methods to be used in association with the descriptive knowledge encompassed in the slot. When the slot is accessed, the procedures are used to perform certain actions associated with this attribute.

Probably the most important aspect of procedural attachment is that they are a built in mechanisms for indexing. Unlike other systems which rely on some rule property such as name, pattern or goal achieved, to retrieve a set of plausible rules KRL has sharply focused retrieval of procedural information.

KRL permits the designers to organize memory into those chunks that are most important to the specific task at hand. Thus procedural attachments are retrieved when needed and there is no problem with saturation. Since the manner in which descriptions are grouped together in units explicitly gathers together information that is relevant to that unit, indexing is no longer required.

Procedural attachments seem to solve many problems however general heuristic knowledge that is not related to a particular data item, such as control knowledge is more easily expressed in control rules.

3.5 Explicit Control of Refinement.

In large rule based systems the existence of saturation has to be accepted and the system must be guided in spite of it. Once it is accepted that given any indexing scheme too many rules will eventually be retrieved the system must have mechanisms for either selecting the appropriate rule from a set of plausible ones or for applying some order on the set of plausible rules.

The set of plausible rules is called the conflict set. A control strategy, called the conflict resolution strategy, decides which rule out of this set to fire. This strategy varies from system to system. The conflict resolution strategy employed must ensure fairness ie ready rules must eventually fire. The conflict resolution set leads to a level of non-determinism which the application of the conflict resolution strategy makes deterministic.

Early attempts at refinement were incorporated in languages which have knowledge based refinement such as PLANNER [Hewitt 1971] and CONNIVER [Mcdermott and Sussman 74]. Most modern expert system shells such as Nexpert [Neuron Data 87], Gold Works [Henson 87] and KEE [Intellicorp 85] provide special conflict resolution strategies.

3.5.1 Ordering the Rule set

The AI languages PLANNER, QA4 [Rulifson et al 72] and CONNIVER used refinement strategies. The interpreter had a method of selecting which of the rules, whose pre-conditions for being fired have been met, must be fired.

PLANNER implemented a refinement strategy via the use of a data structure called a recommendation list. This offered a mechanism for encoding refinement information that can be associated with a particular goal in a specific context. This allowed for different advice for the same goal in different contexts.

PLANNER used a depth first search with backtracking. At each point in the search space, when a decision had to be made about which procedure to use next, a final judgment is passed on each potential theorem in turn. This did not allow PLANNER to make comparative judgments between potential procedures.

CONNIVER also offered explicit control of rule invocation. It had a similar control structure to PLANNER except that a pattern directed call yielded a possibility list containing all the operators that matched that pattern. This list is accessible as a data structure and can be modified with list operations. These operations reflect the relative utility

of any operator retrieved. This allowed for comparative assessment of which theorem to use next.

Applications of the QA3 language explored strategies that acted to prune the set of clauses which might be resolved. It kept an active list and a secondary storage list. It represented these strategies in the same language (predicate calculus) as facts about the domain.

3.5.2 Conflict resolution Strategies.

Most expert system shells offer strategies for selecting or ordering the valid rules. The most common method is to order the rules according to some predefined priority. There are a range of such functions.

Some of the common strategies used are :

1 Refractoriness: [Jackson 83] this strategy says that a rule should be allowed to fired only once in a given set of data. Once the rule has been fired, it is removed from the rule set and in the next recognize-act cycle, one the other fireable rules will be fired.

2 Recency : The conflicts are resolved by firing the rule whose premises match the newest facts in working memory. The

facts that have been accessed most recently are considered the newest.

3 Specificity : if two rules are fireable , and the premises of one are a superset of the other's, then the more specific one fires. The most specifically instantiated rules are assumed to be the relevant ones.

4 Priority : rules are given numeric priorities and when there is a conflict, the rule with the highest priority is fired.

These strategies or combinations of these strategies allow the knowledge engineer to have control over the order of invocation of the rules.

3.6 AGENDAS

A system can control the search for a solution by controlling which rules will be considered for firing. These rules can be further refined by deciding which of them will be fired. Many of the advantages of these concepts can be achieved using an **agenda**.

An agenda is a structure used to record the potential actions awaiting execution during a consultation. It is

thus a queue of competing processes. By using priorities it can control the order of invocation of these processes. A system employing an agenda should supply mechanisms that can dynamically change the order in which actions on the agenda will be performed [Davis and Lenat 80].

Problem solvers repeatedly decide what to do next. A good control strategy must be flexible so that a problem solver can take advantage of new information. Agendas supply a mechanism for implementing these features. Agendas have goals and a repertoire of possible actions.

Agendas have been used in planning systems such as MOLGEN [Stefik 1981a] which was used to plan genetic experiments. The MOLGEN project discovered that because many planning decisions are themselves about meta-problems, there is a combinatorial explosive number of interactions in single level control organizations such as agendas. [Stefik 1981b] extends the agenda idea to a multi-layered system : the layered idea reduces the apparent complexity of the system. It also allows for the inclusion of meta-level heuristics or advice about problem solving.

MOLGEN has 3 levels of control: the top layer controlled what steps to execute in the middle layer which in turn decided what operations needed to be performed.

KRL had an agenda which had a description of each process and a scheduler function which was run to decide what

process to execute next. Each process in turn can be a scheduler with its own agenda. This supplies hierarchical control structure which is geared towards multi-processing.

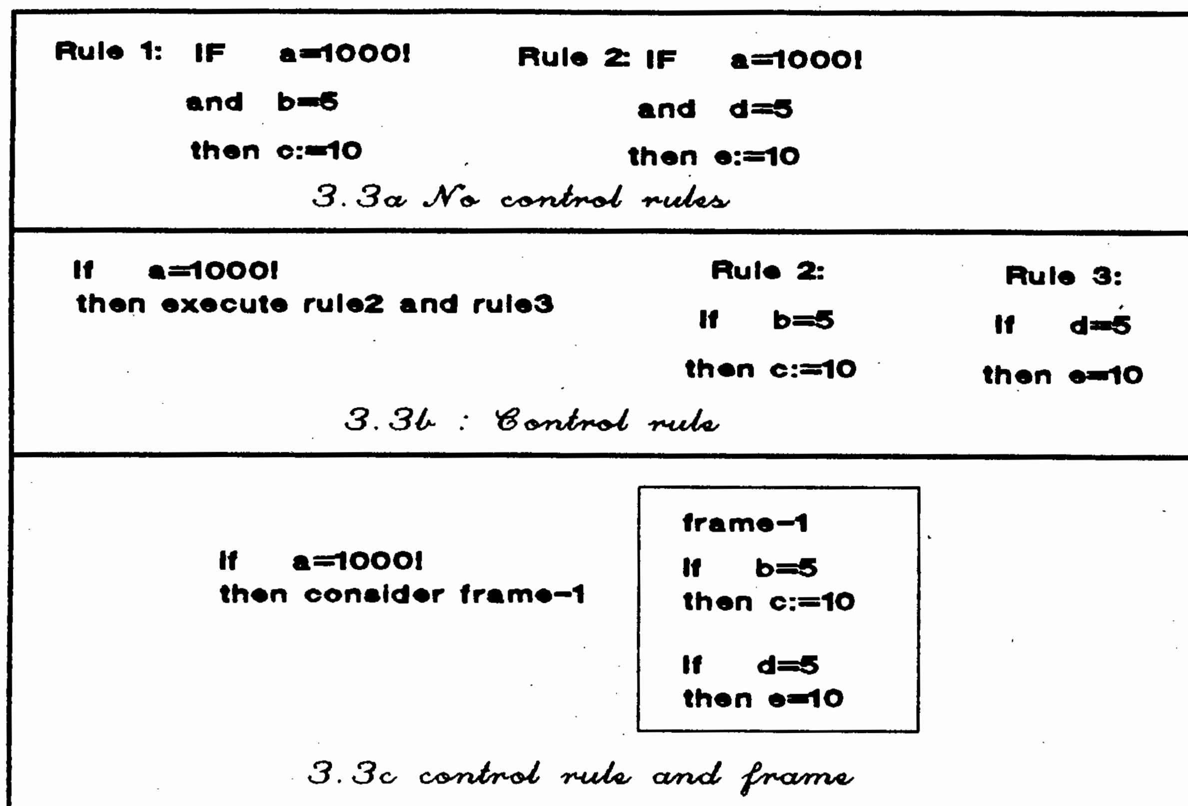
The main problem with agendas is that if the system has multiple interacting goals then interaction between the tasks on the queue becomes complex and cumbersome.

3.7 Meta-rules.

Meta-rules are used in the implementation of local or global strategies for rule retrieval or refinement. Meta-rules make conclusions about other rules. These conclusions concern the utility of some rule, not its validity. Meta-rules are differentiated from other rules in that they direct the reasoning process rather than actually performing the reasoning [Charniak and Mcdermott 85].

In their simplest form, meta-rules can partition the rule base into relevant and irrelevant information. In this case they can be used to retrieve only rules that are relevant due to some criteria. [Diagram 3.3] illustrates how the application of a meta-rule can save the system from doing extra work. In [diagram 3.3a] 1000! must be calculated twice in [diagram 3.3b] the context in which rule-2 and rule-3 are relevant is a=1000!, which is calculated only

once, after this has been calculated rule-2 and rule-3 are considered.



3.3a No control rules

3.3b : Control rule

3.3c control rule and frame

In Teiresias [Davis 1980], an extension to MYCIN, meta-rules were used to refine the set of relevant rules. Because it is a backward chaining system, only relevant rules are retrieved and put in the conflict set. This conflict set is then worked on by a meta-rule which imposes some order on the set of possible rules or does some pruning to the set. This is an improvement over Mycin's exhaustive strategy which tries each relevant rule and then continues with the one with the highest probability.

Meta-rules are usually, but not always, domain specific. However higher level domain independent meta-rules are also

possible eg one could have a meta-rule that advises the system to consider rules which generate small search spaces before rules that generate larger ones [Jackson 83].

Meta-rules allow a uniform encoding of control knowledge which makes the treatment of all levels the same; it is therefore easy to have multiple levels of control. Like agendas, meta-rules can be used to implement a layered control regime where there are meta-rules that refer to other meta-rules.

Meta-rules state the control explicitly. This means that the control of reasoning is conceptually clean and allows simple modification of existing strategies. They limit the depth and breadth of the implications drawn from any new fact or conclusion. The general advantages of the rules representation also apply to meta-rules.

The use of meta-rules has some problems associated with it. Using meta-rules means that the code for each rule is no longer self-contained. This undermines many of the representational advantages that rules have, such as modularity. Thus a major concern for the knowledge engineer using meta-rules is that of rule interaction when new rules are added to the existing set. The problems that could occur are illustrated in [diagram 2.3]. If a rule-4 is added to the system with an if clauses "a=1000! and e=5" then the meta-rule must also be changed to consider this rule.

Another problem is that the automatic explanation generation is complicated by the fact that some rules are used purely for control.

8 Explicit Rule Clustering.

Centaur [Aikins 1983] is concerned with the order of invocation of rules ie control. One of the main themes of this project is that the context in which knowledge is applied should be explicit. Rules are grouped together according to their function in the consultation. By doing this, the relevant rules for a particular subject or goal can be explicitly grouped together. The meta-level reasoning concerns which set of rules are relevant to a problem.

Systems that have a global strategy for deciding which rule to choose were discussed in the section on conflict resolution strategies. In CENTAUR the control knowledge is represented within each prototype. This provides context-specific control and separates the control knowledge from other knowledge in the system. Each set of rules, or goal finding functions, can have local control knowledge. The prototypes thus provide the explicit context in which the

finely grained reasoning done by the production rules takes place.

The WISE system uses a knowledge representation scheme that includes, as one of its features, rules in frames. By attaching rules to a particular frame, the system can have local conflict strategies, search strategies etc.

The representation and the control are closely related in a system such as CENTAUR and WISE. The main issue here is one of explicit control. The frames allow the rules to be executed in a certain context defined by the frame in which they are stored. Thus it is not necessary to state a rule's full context in its premises. [Diagram 2.3c] shows how meta-rules can be used to direct the reasoners attention to a frame with rules. If new rules are added to this set the meta-rule does not have to change.

3.9 Conclusions

Various strategies for controlling the order of invocation of knowledge in a knowledge base have been considered. There are certain criteria for deciding whether a control strategy is useful or not.

One of these, is the level of indexing. This relates to the place where the knowledge is selected . In STRIPS and GPS there is one global rule selection function which is implicit in the interpreter. In PLANNER there are many local selection functions generally associated with specific goals. Meta-rules also allow the user to specify where a particular strategy must be applied. Putting rules in frames provides control over which strategies are applied to which rules. This level of indexing has an impact on the power and efficiency of a strategy encoding mechanism.

The control information should be represented in the same way as object level information. This control information must also be accessible. This relates to whether it is encoded at the program level, as in local strategies of PLANNER, CONNIVER and QA4, or whether it is encoded as part of the interpreter such as in the global strategies of STRIPS.

CHAPTER4: REASONING WITH NON-MONOTONIC LOGIC.

4.1 Introduction.

The reasoning process is often faced with situations where there are many possible actions to perform. Assuming only one of these possible actions is the correct one, implies that the other possibilities are incorrect (ie. will not find a solution to the problem). This assumption (concerning which action to perform) might later be proven incorrect. Once it is accepted that the reasoner will make decisions that will later be proved incorrect, then the system must perform it's reasoning on a non-monotionic logic.

A monotonic logic is one where the addition of new axioms does not disprove any existing theorems. New axioms only add to the list of provable theorems and never cause any to be withdrawn hence the name monotonic logics [Winston 84]. Thus in a monotonic system :
(A -> x) -> ((A or B) -> x).
Since the addition of new knowledge cannot disprove previously derived facts all the derived facts must be based on complete knowledge. A conclusion cannot be accepted until it is proven. If a proposition, P is derived, and then later (not P) is derived ie there is a contradiction

then the entire system becomes inconsistent and none of its conclusions can be consistently believed.

Rarely, does a system have full knowledge of its domain. Often a reasoner is required to make assumptions or take defaults. The usual type of default taken is the most probable choice [Rich 83]. For example when reasoning about the "bird tweety" we wish to know if she flies, however the system do not know everything about tweety so in order to continue it assumes that she does fly [diagram 4.1]. Because this fact is an assumption it is a belief not a fact. New information can disprove this previously derived fact. For example it could find out tweety is an ostrich.

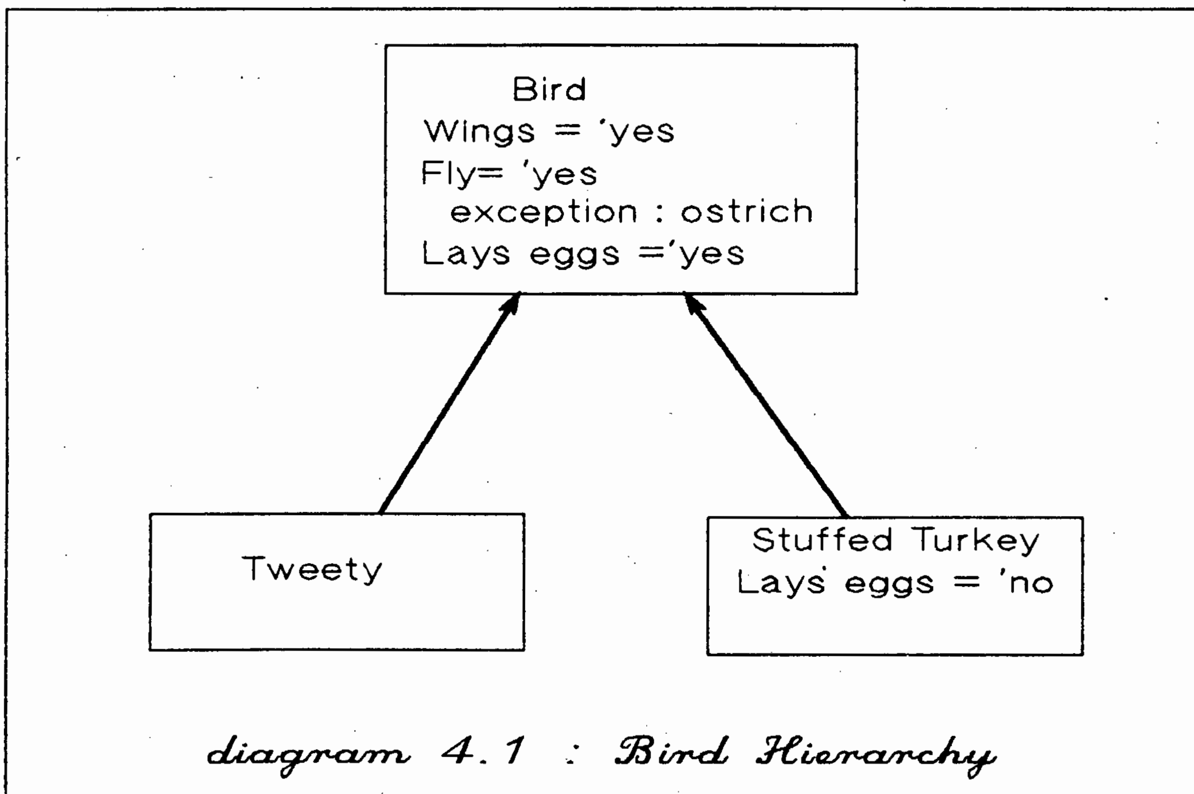


diagram 4.1 : Bird Hierarchy

Most expert system shells provide features for purely monotonic reasoning. However many problems need good guesses, or plausible inferences based on partial knowledge, to find a solution. These guesses usually rely on the fact that there is no contradictory evidence at present ie in the absence of anything to the contrary assume a fact to be true. As new facts are added, contradictions due to these guesses could come to light. In this case the guesses, and facts that are based on them, must be retracted.

When a human is faced with a contradiction he can readily revise his beliefs to incorporate the contradiction. For example a child knows that birds can fly yet when faced with an ostrich (or a dead bird) he can readily resolve the seeming contradiction. "We are able to discard old conclusions in favor of new evidence" [Doyle 79]. The ability to do this is the basis of common sense reasoning.

Non-monotonic logics are useful, not only for common sense reasoning, but also in domains such as design, diagnosing multiple faults, planning about actions and systems that employ causality [Mcdermott and Doyle 1980]. In the past decade there has been a lot of interest shown in logical systems that allow non-monotonic inferences [Shoham 1987]. These systems are able to perform speculative reasoning by jumping to conclusions based on incomplete knowledge.

The are several important types of speculative reasoning that will be discussed in this chapter including, default logic, circumscription, and the closed world assumption.

4.2 Reasoning about actions.

Non-monotonic logics grew out of attempts to model actions. Actions are defined by operators that act on certain situations to create new situations. [McCarthy and Hayes 1969] suggested a monotonic solution to problems encountered while reasoning about actions, namely the situation calculus. This approach involved describing situations and then explicitly stating axioms for inferring what propositions held following each possible action on theses situations.

A problem arises because systems, that reason about actions, are forced to state not only what changes occur to a situation when an action is applied, but also what has remained constant. In addition to the axioms describing how the properties of the current situation change we need a set of axioms, which characterize that which remains fixed while an action takes place. These were called the frame axioms. [Diagram 4.2a] shows a situation where:

blocks a, b and c are on the table

a is next to b, b is next to c

a is not next to c

c is not next to a

If the action "lift block B" is performed, the system must be told:

- what has changed: B on table, A is not next to C
- what has remained the same: A on table, C on table
- new situations such as: A next to C.

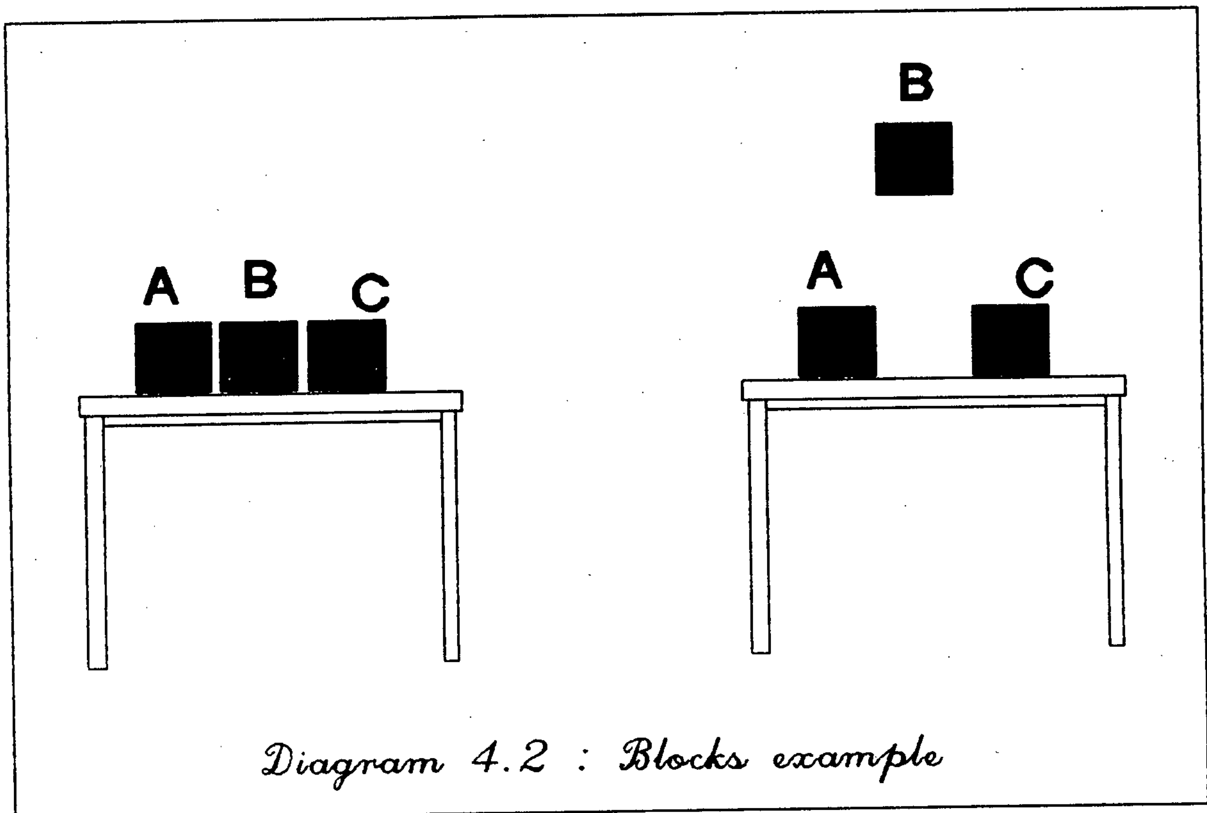


Diagram 4.2 : Blocks example

The frame axioms lead to the frame problem [Rapheal 1971]. To establish whether any proposition of the form P(obj,sit) is true, it is necessary to chain back through the entire set of operators that generate the situation, regardless of whether they had anything to do with "obj" or not. Expressed differently, the problem is that of which formulas

should stay the same and which should change. This is called the frame problem.

Plan solving is the generation of a series of actions or 'programs' that will achieve a certain goal (usually with constraints). The most common area of research into plans is in the field of robots but expert systems sometimes need to formulate plans too [Stefik et al 83].

The sequence of actions which achieve the goal is called the plan. A sequence of actions involves moving from one situation to another similar, yet slightly different situation. Thus a solution to the frame problem is required to enable successful reasoning about such a sequence of actions.

Another complexity that arises due to reasoning about actions is, whenever an action is performed there are many hidden reasons that could cause it to fail. Some of these are what is called anomalous conditions, which are conditions that are outside the model of the plan in normal operation eg in [diagram 4.2] the block might be stuck to the table. The negation of each of these conditions could be included in the antecedent of the rule but this is not practical as there may be many such conditions (possibly an infinite amount) and in the vast majority of cases none of these would hold. This is known as the qualification problem [McCarthy 80].

An expert system must be able to decide on an action without worrying about all the anomalous conditions that would cause this action to fail. However, it must also be able to handle finding out that an action previously performed was impossible due to some anomalous condition. It thus works without worrying about all these conditions, yet should it later discover that one was applicable it then revises its beliefs accordingly.

STRIPS was discussed as a system that used means end analysis. STRIPS attempted to get round the frame problem by keeping a single copy of the world description and modifying it to reflect the execution of a particular action. The basic observation in STRIPS is that the world does not change much from one instant to the next [Ginsberg & Smith 87]. STRIPS makes a non-monotonic assumption [Waldinger 1975] which says every action is assumed to leave every relation unaffected unless it is possible to deduce otherwise. STRIPS keeps a single model of the world which is updated as actions are performed on it.

STRIPS actions are defined in terms of pre-conditions, add-lists and delete-lists. The result of an action is to add the facts in the add list and remove the facts in the delete list. In the previous example of [diagram 4.2]
pre-condition : B on table and nothing on B
add-list : A next to C
delete-list : B on table, B next to C, A next to B.

This made it possible to avoid the explicit statement of frame axioms for all those things that remain unchanged. Furthermore, in conjunction with the implicit specification of situations, it provides an efficient way to maintain consistency [Winograd 80]. STRIPS thus offered a good solution to the frame problem however it provides no solution to the ramification problem. As the domain gets more complex, the consequences of actions get more complex. The onus is on the designer to explicitly list all the ramifications of an action. This leads to long, and complex, add and delete lists.

The designers of STRIPS realized this problem and thus a requirement of STRIPS is that the knowledge base contain only facts of a very specific nature [Ginsberg & Smith 87]. By doing this they could guarantee that the database remained constant. [Lifshitz 87] concluded that STRIPS is viable only when the facts about a world come from a pre-defined set. This restriction limits its use to simple domains.

What is needed is a non-monotonic solution to the frame problem and anomalous conditions as they relate to working with incomplete knowledge.

4.3 Inheritance, Defaults and exceptions:

The frame and semantic net based knowledge representational paradigms that were discussed in chapter2 have facilities for inheritance, defaults and exceptions. Hierarchies allow us to describe facts such as "all birds fly" [Diagram 4.3]. However this general rule has exceptions such as ostriches and dead birds. Using the general first order logic we would show this as :

?x (bird(x) ^ (not (Penguin(x) or ostrich(x)...) => fly(x)

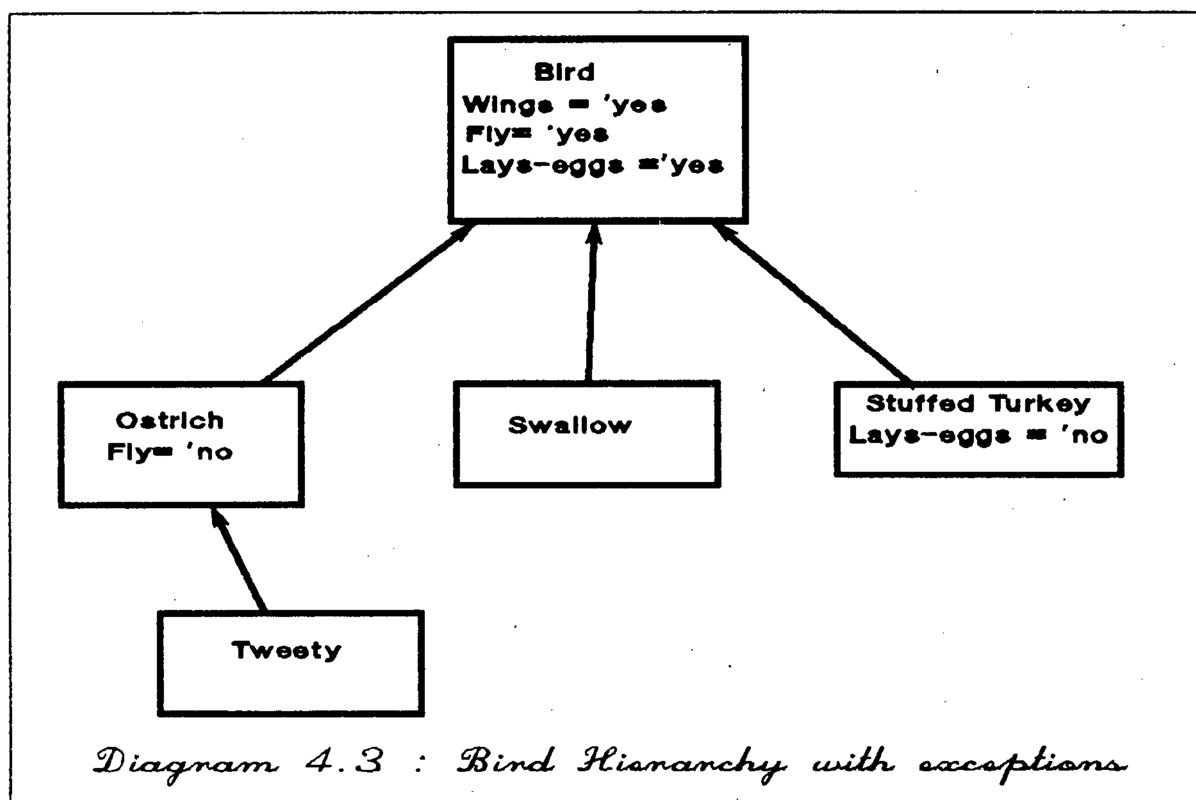


Diagram 4.3 : Bird Hierarchy with exceptions

This is cumbersome and does not allow the system to conclude whether the general bird (tweety) can fly without knowing whether tweety is an ostrich (ostrich(tweety)). In order to solve this problem [Minsky 81] defined the concept of prototypical situations. These expected descriptions or

defaults are inherited by the children or instantiations of a prototypical frame. In this case we can assume tweety is a prototypical bird and can thus fly.

Prototypical situations must be able to model exceptions [Brachman 1982, Etherington and Reiter 1983, Etherington 1987]. These exception allow for the cancellation of certain properties along the IS-A relation. To return to the above example initially the property "fly" is inherited from the general bird description. If it is later discovered that "tweety is an ostrich", then ostrich's attributes override bird's [diagram 4.3] and the belief that tweety can fly is retracted.

If FLY(tweety) is assumed by default then it has the status of belief and this belief might later be retracted (if infer that tweety is an ostrich) non-monoticity is therefore needed. Defaults are thus modeled using a non-monotonic logic. [Reiter 80] defined the consistency operator :M (to be read it is consistent to assume) and added it to the first order logic to make it non-monotonic.
The default rule:

BIRD(x) :MFLY(x)/FLY(x).

says if x is a bird and it is consistent to assume that x can fly then infer that x can fly :
The exceptions are then listed separately in the first order representation :

(x).penguin(x) ==> (not (fly(x)) etc.

The consistency operator also allows for a non-monotonic solution to the frame problem. The frame problem can be expressed as for all relations R which take a state variable as an argument, and for all state transition functions f :

$$R(\mathbf{x},s):MR(\mathbf{x},f(\mathbf{x},s))/R(\mathbf{x},f(\mathbf{x},s))$$

Intuitively this default assumption formalizes the so-called 'Strips assumption'. Every action is assumed to leave every relation unaffected unless it is possible to deduce otherwise.

The act of taking a default is the same as, accepting one of a number of possible views of the world. Reiter call each such possible view an *extension* to a logical theory. An extension contains all the default conclusion that can be consistently made. It provides one possible view of the world. Because we are working in an incompletely specified world, there can be many such coherent views, one for each extension of the default theory.

For example consider the situation :M(not p)/q and :M(not q)/p . This says if it is consistent to assume (not p) then infer q and if it is consistent to assume (not q) then infer p. This theory has 2 complete extensions {p,(not q)} and (q,(not p)} ie we either choose to believe the default (not p) which forces us to believe q or we we believe the default (not q) which forces us to believe p.

Each of these extension provides a possibly correct view of the world depending on the context. We are free to choose any of these extensions as our current view of the world. The particular set of defaults that is believed will usually be conditioned by some overall objective. The reasoner does not believe any arbitrary set of derived beliefs. All the beliefs must belong to some common extension. In order for the reasoner to use these features we need to be able to tell if a set of derived beliefs belongs to some extension.

The addition of new facts might lead to inconsistencies, to restore consistency the system must initiate a process of belief revision. This involves finding a consistent subset of the derived beliefs that makes the new set of default proofs consistent. Failing this reject some minimal subset of the derived belief set, this causes a change in extensions. This belief revision is one of the tasks carried out by a truth maintenance system [Doyle 79].

4.4 The Closed World Assumption.

The closed world assumption [Reiter 1978] is a very powerful and useful non-monotonic assumption for handling real-life situations. The assumption is that if a proposition cannot be proven true then it is false. Negative information is inferred by default. Reasoning under the closed world

assumption greatly simplifies the representation of that world as only the positive information about the world need be explicitly represented.

For example if we wish to find out if tweety is an ostrich and we have no information about what type of bird tweety is we assume not an ostrich until proven otherwise. Systems reasoning under the closed world assumption treat negation as failure to prove [Ginsberg 87]. This is the way negation is used in PROLOG.

PLANNER [Hewitt 1972] was a precursor to PROLOG. It has consequent theorems that correspond to STRIPS operators ie with GOALS to be matched and theorems to be deleted or asserted. It was the first system to treat failure to prove as negation. It did this by using the THNOT operator which allows for the closed world assumption and other forms of non-monotonic reasoning.

THNOT was a logical operator used in the formation of formulas in PLANNER. The procedure (THNOT A) means try to prove A and if you fail then (THNOT A) is proved. The success of THNOT A does not depend on proving (NOT A) but on not being able to prove A. For example a theorem for determining whether X is the brother of Y:

```
(CONSEQUENT (brother-of ?x ?y)
        (THNOT (exists (?Y) ((GOAL (sibling ?x ?y)
        and  (GOAL (Sisters ?x ?y))))))
```

This theorem will succeed if we fail to prove that: there is some object y that is a sibling of x and a sister of x.

Because we can never know everything about the domain we are forced to reason under the closed world assumption. THNOT is only reasonable in a system like PLANNER for which there is a finite set of inferences that will be tried for a given proof. Otherwise it becomes a very expensive operation. Only systems with explicit control of databases will thus be suitable for a THNOT type inference. But it can theoretically be applied to any system in which an attempted proof can terminate without succeeding.

One of the most useful applications of the closed world assumption (CWA) is that of handling the two types of knowledge that nearly all systems involved in reasoning about actions or planning lack. These are explicit anomalous conditions and frame axioms. Because no anomalous conditions or frame axioms are mentioned we assume there are none.

The CWA assumption can be modeled using the default operator [Reiter 81]. The CWA says that for any relation R, and any individuals, $x_1..x_n$ one can assume (not $R(x_1..x_n)$ whenever it is consistent to do so ie.

:M(not $(R(x_1..x_n))$/(not $(R(x_1..x_n))$.

4.5 Circumscription.

Another attempts at modeling non-monotonic reasoning was McCarthy's Circumscription; which is rule of conjecture used to 'jump to conclusions' namely : "the objects that can be shown to have a certain property P by reasoning from certain facts A are all the objects that satisfy P" [McCarthy 80]. It is used to solve the frame and qualification problems.

Circumscription formalizes certain common sense reasoning eg common sense tells us that a tool can be used for its intended purpose unless there is some explicit reason that it can not. Because ordinary logic systems are monotonic by nature circumscription cannot be achieved by adding axioms or rules to them [McCarthy & Hayes 69].

[McCarthy 1984] showed how circumscription could be used to formalize common sense reasoning and to do many of the non-monotonic inference done by Reiter. [Lifshitz 1987] showed that Reiter's closed world assumption is equivalent to circumscribing all the predicates in the database.

4.6 Uses for non-monotonic logic.

This chapter has shown how non-monoticity allows reasoning with incomplete knowledge. The concentration has been on situations that require a solution to the frame and qualification problems. However even many simple problems require non-monoticity.

The inclusion of non-monotonic features widens the range of problems which can be solved by expert systems. These new problems come from several domains some being:

Design systems create a set of specifications for a product. The final product has to satisfy certain constraints. A common approach to design is to use tentative reasoning [Stefik et al 83]. This process requires making a decision, often based on partial knowledge, exploring its consequences, and if it leads to a bad design or a broken constraint changing the decision.

Planning systems build a plan which consists of a method of achieving a set of goals without violating a set of constraints. As with design systems, tentative plans must be built and tried, and if they lead to the breaking of a constraint a slightly different plan will be tried.

Advanced Diagnostic systems are often faced with symptoms that could arise from a multitude of reasons or from several

reasons at once. A way to solve such a problem is to assume what is wrong and see if it explains all the symptoms, if all the symptoms cannot be explained change the assumption.

Any application that is required to reason in a domain where it is faced with more than one possibility of what action to perform next, or what to believe, must make a reasoned guess based on its partial knowledge [Marcus, Stout et al 88]. This guess involves making a choice between possible scenarios. If a contradiction, such as a broken constraint, is reached the system must backtrack to the point where a choice was made and make another choice. Backtracking is a very costly operation and must be carefully controlled [Waterman 85].

4.7 Conclusions.

A system using non-monotonic reasoning must be able to reject a belief in a uniform fashion by producing a new as yet unchallenged argument against the belief. In order to do this there must be an explicit way of choosing what to believe ie must have a way of selecting which of the possible revisions of beliefs we will take when faced with new information. Such systems require pragmatic belief revision rules.

The belief system must be additive ie nothing must be thrown away. In this way arguments can be accumulated and then

used whenever possible. Thus future debates are guided by keeping them from repeating past debates.

Systems that use non-monotonic logics can:

- make decisions based on incomplete knowledge
- make decisions which may later be revised
- provide a economical solution to the frame problem.
- explicitly control a consultation
- use common sense reasoning.

CHAPTER 5 : TRUTH MAINTENANCE.

5.1 Introduction

The non-monotonic paradigms that were discussed in the previous chapter are implemented using a truth maintenance system [Doyle 79] (TMS). A TMS is a system that maintains a consistent set of beliefs. In order to do this it supplies a mechanism for storing **beliefs** and the **reasons for beliefs.** Because a fact is associated with its reason for being believed, if the reason for believing a fact is later invalidated, then this belief can be retracted.

In systems that use a TMS the problem solver is separated into a part concerned about the rules of the domain and a part concerned about the current state of the search space. The TMS does not make any new inferences but it maintains consistency among the facts inferred by the inference engine. The idea is to increase the efficiency of the problem solver without losing coherence or exhaustivity [De Kleer 86].

The prime issue is still one of control ie what goal to try achieve next, which plausible inference to draw from incomplete data, which action to perform next, which possible world to explore next etc.

This chapter will look at some of the mechanism that have been used in truth maintenance. Part2 of this thesis will cover the WISE truth maintenance mechanism in detail.

5.2 Data Dependencies.

A knowledge base is not static; new facts are inferred and old facts are changed. Suppose we infer:

P and P=>Q

this will cause Q to be inferred.

Should P be retracted, there is no longer any reason to believe Q [Charniak & Mcdermott 85].

Every assertion, or fact, must have attached to it a record of why it is believed, this record is used to maintain consistent belief statuses for the facts. This record is called a data dependency or **justification.** Data and its dependencies can be schematically represented as in [diagram 5.1].

Definitions :

node : is a place where a fact is held. It is represented by a box in [diagram 5.1].

justification : reason for believing a node, consists of other nodes (antecedent nodes) and pointers to the node being justified (consequent node). This is represented in

[diagram 5.1] as a circle with the antecedent nodes pointing in and a pointer going from the circle to the consequent node.

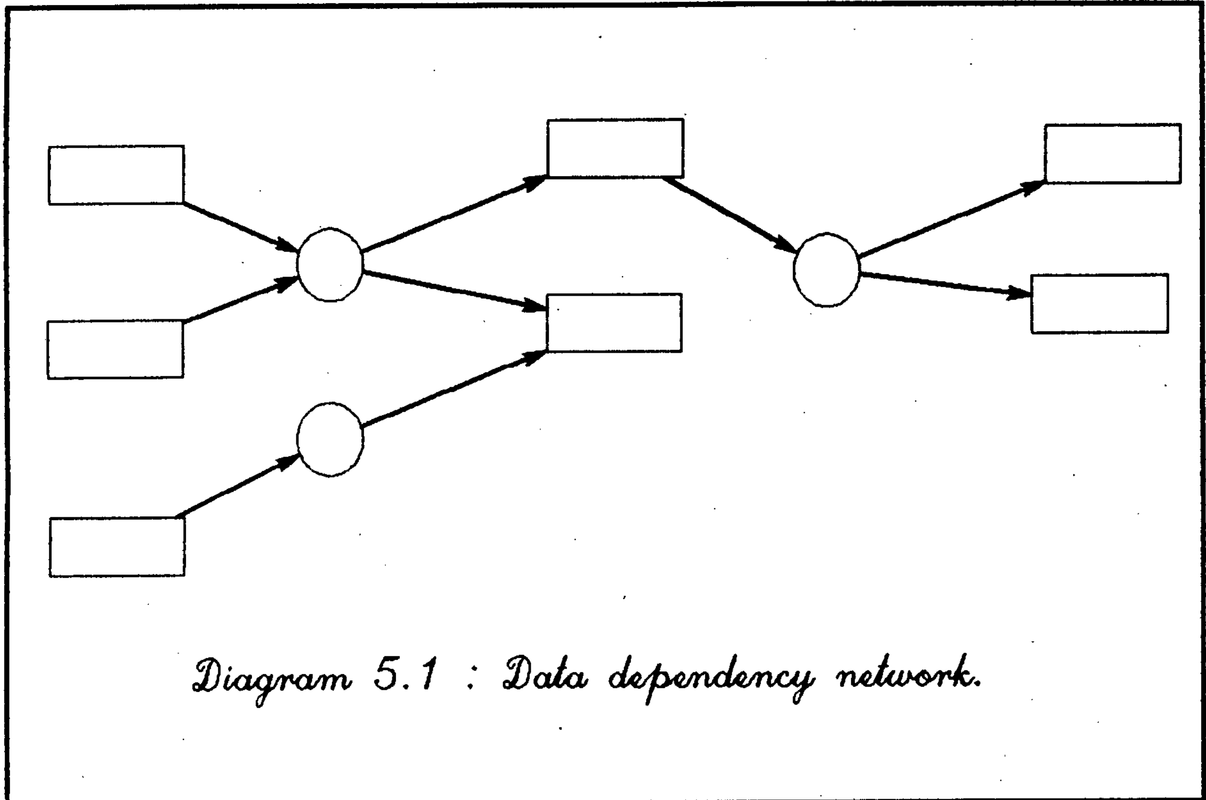


Diagram 5.1 : Data dependency network.

In order to keep the facts consistent with what is believed each fact is labelled with its current belief status. A fact is believed, or labelled IN, if it has **well founded support** from the atomic assumptions which are currently believed. An OUT label means that fact is not believed ie no well founded support.

"P is OUT" does not necessarily imply that one should believe (not P) ie there is a difference between P being OUT and (not P) being IN. The first statement says the system has no reason to believe P the second says the system has a

reason for believing the fact (not P). By applying these belief labels to facts the system is allowing for the possibility that a fact may later be revised ie the facts are just beliefs and are not permanent.

The object of a TMS is to label the nodes in a justification network as depicted in [diagram 5.1]. These labels represent the current state of the beliefs.

5.3 Justification based Truth Maintenance.

The justification based truth maintenance system (JTMS) was devised by [Doyle 79] . The fundamental actions of the JTMS are:

- to create a new *node* (to which the problem solver can attach a belief)
- add or retract a *justification* (reason for believing) for a node
- mark a node as a *contradiction*, this represents the inconsistencies of the current set of beliefs.

Let us consider the example TMS defined in [diagram 5.2.]. A simple TMS would have data structures for representing the beliefs (nodes) and the reasons (justification).

A NODE

Datum : corresponding structure in problem solver
Support : a justification that supports this node
Justifications : list of justifications that justify the belief in this node
Consequences : list of justifications in which this node is an antecedent
Label : belief status of this node

A JUSTIFICATION

Informant : name for use by the problem solver
Consequent : consequent node
Antecedents : list of antecedent nodes

Diagram 5.2 : Structures used by the TMS.

As the problem solver infers new facts the TMS stores these facts and their justifications. First the justification for the inferred fact is added to the node's justification set. The node which represents the newly inferred belief is placed in the consequent of the justification. Each node that plays a part in deciding the validity of the consequent node is placed in the antecedents of the justification and has this justification put in it's consequences.

The justifications place constraints on the truth values of the TMS nodes, if the antecedent nodes of a justification are IN then the consequent node must also be IN. By doing this the TMS can maintain a consistent set of beliefs.

The same fact may be deduced in more ways then one ie it may be believed for more than one reason. All the ways that a fact is deduced are kept. A node may have several reasons for being believed but only one of these justifications is its *support*. If a reason for believing a fact is invalidated the system checks if any of the other justifications for the fact are valid and if so it installs one as the new support for the fact.

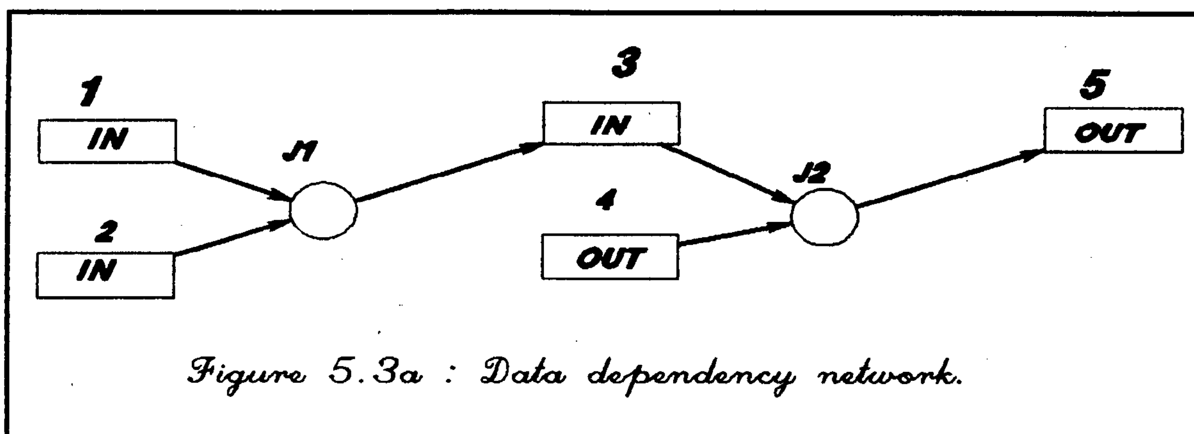


Figure 5.3a : Data dependency network.

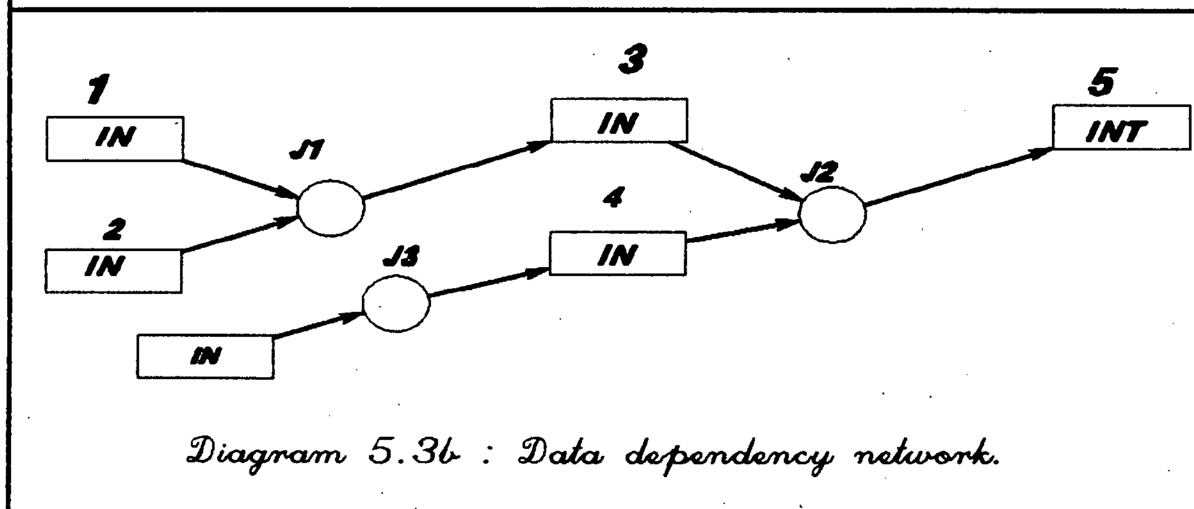


Diagram 5.3b : Data dependency network.

When adding a justification for a node, if the node is already IN (believed), then the beliefs are consistent and the system need do no more. If the node is OUT we start a process called **propagate-IN**. Propagate-IN propagates the effect of this belief coming IN through the justification network. The initial situation is represented in [diagram 5.3a]. When justification J3 is added, in [diagram 5.3b] node 5 becomes IN, and the propagate-IN process is started, this causes justification J2 to become valid, causing node 5 to become IN.

The reasons for believing a fact may be circular however there must be a non-circular support graph back to the atomic assumptions. This means the belief of a fact cannot be dependent on itself. The support must be *well founded* this means that the support has no circular arguments.

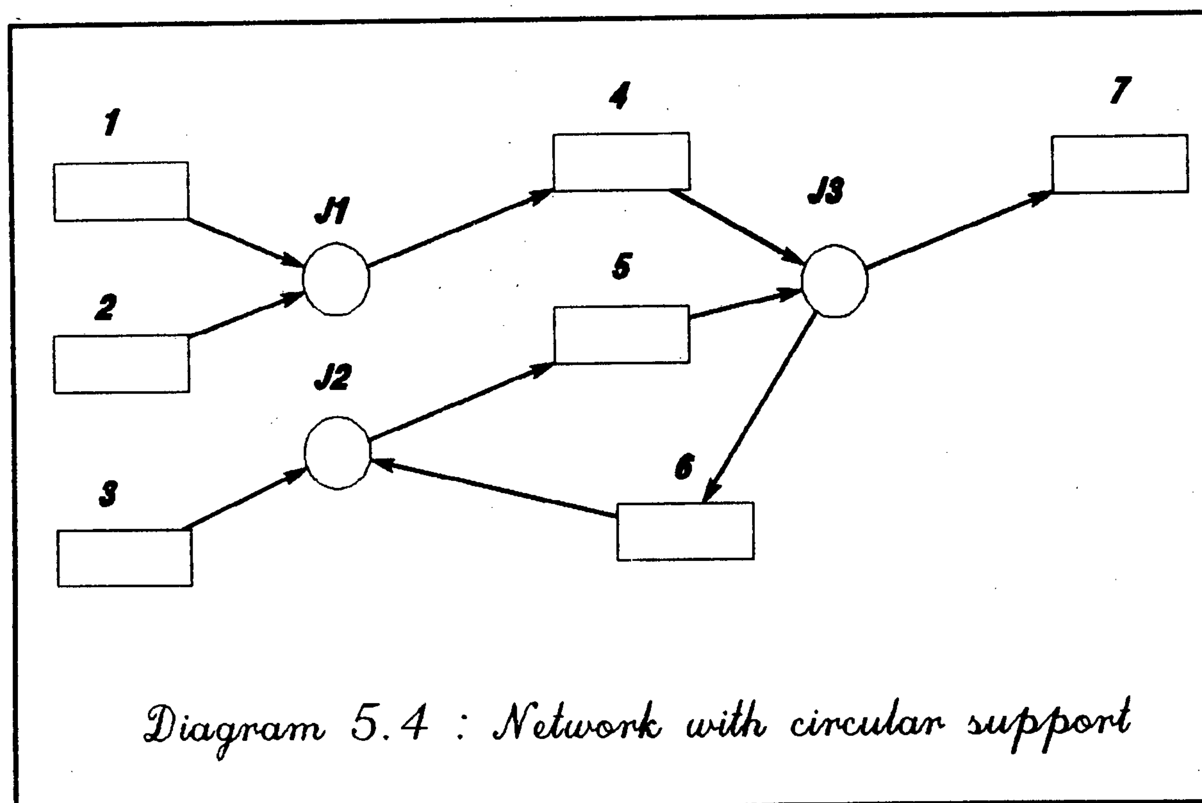


Diagram 5.4 : Network with circular support

In [figure 5.4]. support for node 5 is circular and thus not well founded.

[Doyle 79] defined two types of justifications namely support list and conditional proof. The two types of justifications represent the two ways in which the belief in a node can depend on the belief of other nodes.

The more common justification is the SL-justification (support list). It has the form (SL <INlist> <OUTlist>) and is valid if all the nodes in the INlist are IN and all the nodes in the OUTlist are OUT. It is used to justify or support a node.

The CP-justifications (Conditional Proof) : have the form: (CP <consequent> <INhypotheses> <OUThypotheses>). The OUThypotheses are usually nil. It is valid if the consequent node is IN whenever each node of the INhypotheses is in and each node of the OUT hypotheses is OUT. This allows hypothetical reasoning. CP-justifications are more difficult to handle than SL-justifications and the system treats them by converting them to equivalent SL-justifications.

A fact such as "fred is-a teacher" with support (SL () ()) is called a premise ie its belief is not dependent on the belief or non-belief of any other fact. Assumptions are facts whose supporting-justification has a non null OUTlist. These fact's beliefs are dependent on the system not

believing another fact, and are thus non-monotonic, for instance if one of the facts in the OUTlist becomes IN then the justification is no longer valid. This allows us to model default reasoning and non-monotonic assumptions. Typically belief revisions systems will explore an alternative based on a choice or assumption and if a contradiction occurs will revise the beliefs based on assumptions [Martins and Shapiro 88].

A non-monotonic assumption such as taking a default involves making a choice between alternative possible beliefs. If the system has to choose from a set of alternatives (A_1 to A_n) and G is the reason that the choice must be made, then (SL (G) ($A_1..A_{i-1},A_{i+1}..A_n$)) is used to support choosing A_i. This justification is non-monotonic as if one of $A_1..A_{i-1},A_{i+1}..A_n$ becomes true then A_i becomes invalidated ie. A_i becomes OUT.

In order to consider this concept in greater detail, a specific example will be reviewed; that of a system that is required to schedule some courses by allocating teachers to them. The statements which will follow below are represented in [diagram 5.5].

In this diagram Doyle's notation has been used. A support-list justification SL has an IN-list and a OUT-list and is valid if the nodes in the IN-list are in and the nodes in the OUT-list are OUT.

A small part of this scheduler allocates a teacher to course-1. To do this the system **assumes** some teacher is the teacher of course-1. It does this by making a choice between the possibilities, in this case BOB and FRED. Assume "Bob teaches course-1". The rules have a rule (rule-3) that states if a course is taught by BOB then the venue is TECH. Therefore the justification for Node-3 depends on Node-11 and some valid rule (rule-3).

Node-1	course1.teacher = BOB	(SL () (Node-2))
Node-2	course-1.teacher = FRED	
Node-3	course-1.venue = TECH	(SL (Rule-3 Node-1) ())

Initial Situation

Node-1	course1.teacher = BOB	(SL () (Node-2))
Node-2	course-1.teacher = FRED	(SL (Rule-4 Node-4 Node-5) ())
Node-3	course-1.venue = TECH	(SL (Rule-3 Node-1) ())

Subsequent Situation

Diagram 5.5 : Scheduling Example

Should the system now consider another rule, represented in node Rule-4, that causes us to believe that "fred is the teacher of course-1", the assumption that "Bob is the teacher of course-1" is rejected. The TMS will revise its beliefs so that Node-2 is IN and Node-1 and Node-3 are OUT. This occurs because after Rule-4 has been executed the

justification supporting Node-2 is valid and Node-2 is thus IN, this causes the justification supporting Node-1 to be invalid as Node-2 is in its OUTlist. Node-1 thus goes OUT.

When a node that was IN becomes OUT a process called propagate-OUT is started. Just as propagate-IN propagates the effect of a node coming IN through the network, so propagate-OUT propagates the effect of a node going OUT. Both of these processes involve checking the consequences of the affected node to see if the change in the node's belief has affected the consequences validity. If it has the consequent of the consequence justification is set to it's new truth value (IN or OUT) and the process is repeated on this just changed node.

Justifications can be used to implement a linearly ordered set of alternatives eg. the system could first try Bob as the teacher and if that fails then try Fred etc. This adds still more control knowledge. The WISE system employs a special mechanism for this type of reasoning.

If an assumption that has been OUTed is believed again, then the dependency information is used to unOUT all its implication thus saving the system from having to work them out again, this involves checking the other antecedents of the old consequences and if they are believed unOUTing them.

5.4 Assumptions define Contexts

Just as defaults define extensions, so assumptions define contexts. A context is defined as a set of beliefs held under a set of possible assumptions. Each set of believed assumptions defines a context. OUTing and unOUTing can be viewed as a context switch; the set of believed assumptions changes by either disbelieving a believed assumption (OUTing) or believing a disbelieved assumption (unOUTing).

When a new assumption is believed, a new subcontext is defined ie a more specific context. When an assumption is disbelieved a new supercontext is entered. Initially the entire set of contexts exist. As more knowledge is added It is automatically placed in the context(s) determined by the premises of each deduction.

A context in which a contradiction is believed, is ruled out as a possible solution. A fact may be known in more than one context for independent reasons. A fact is not installed in the current context, it goes into the highest super context that its derivation will work in and is merely inherited by the current context.

In the JTMS, the contexts under which a piece of data is believed, are implicit. The current context is defined by the set of IN data. This requires that the database be kept consistent and makes it impossible to refer explicitly to a

context. Because there is no explicit reference to contexts the system requires truth maintenance and dependency directed backtracking (DDB) to move to a different point in the search space. By having the assumptions of the current context at the node of the contradiction DDB is greatly speeded up in the JTMS. The system doesn't need to search for the assumptions.

5.5 Handling Contradictions.

Systems using a JTMS can employ tentative reasoning to solve their problems. They can make assumptions that may later be shown as incorrect ie lead to a contradiction. The contradictions would be detected by special equations. The TMS is not responsible for finding the contradiction a contradiction is discovered by the inference engine and the contradiction, plus the reason it arose, is passed to the TMS. The contradictions must be processed by making new assumptions.

In the section on forward and backward chaining it was mentioned that some systems backtrack when they reach a dead end or a contradiction. Backtracking decreases the size of the search space as when a failure is reached the system does not throw away the whole search ie every possible solution is not worked out from scratch.

A wrong decision is discovered when the system fails to find the goal or reaches some contradictory state. In systems that do not employ DDB the system will backtrack to the most recent assumption and try another alternative. This is known as chronological backtracking and is the central control mechanism of PROLOG. However, we need a way of intelligently processing contradictions. In order to achieve this, the system requires an ability to distinguish guilty assumptions (assumptions on which the contradiction depends) from innocent ones (assumptions not related to the contradiction). This is called Dependency Directed Backtracking (DDB).

Withdrawing statements based on the order in which they were generated by the search program rather than on responsibility for a failure, gives rise to the system performing wasteful search. When a contradiction is found, the system should backtrack to an assumption that led to the contradiction not just the last one. Chronological backtracking leads to contradictions being rediscovered. A set of assumptions that led to a contradiction should never be tried again. However a system that doesn't know the reasons for a contradiction can't help doing this.

Obviously non-monotonic systems which could have the basis of their beliefs proven untrue at any point need DDB. DDB was first implemented in EL [Stallman and Sussman 77]. The ideas from EL where used to build the first TMS [Doyle 1979] which supplied mechanisms for DDB.

When faced with a contradiction, the system examines the reasons that caused this contradiction. The DDB recognizes the assumptions (non-null OUTlist) upon which its belief is based. It does this by tracing backwards from the contradiction node following each path until it hits an assumption or a premise. If it finds an assumption it is put in a set (S). This set of assumptions, which represents a context, are put in a NOGOOD-set (S) for the contradiction.

The NOGOOD is put in a node, if this set appeared as a nogood earlier then that previously defined node is used. Justify the nogood node with:

(CP contradiction-node S ())

ie contradiction is IN when the nodes in S are IN. This remains IN even after one of the assumptions is OUTed since the CP-justification means that the NG does not depend on any of the assumptions.

Once the NOGOOD-set is formed a culprit assumption is selected from the set (S), and a node from this assumption's OUTlist is chosen. This new assumption is supported with (SL (NG $A_1..A_n$) (rest of choices ie OUTlist of other assumption minus this assumption)). The culprit is forced OUT by invalidating its supporting-justification with the new justification (by bringing a node from the OUT-list IN).

Thus a new assumption is made and a old one is retracted at the same time. It is all very well to change an assumption that led to a contradiction but all the beliefs based on this assumption must also be retracted. Similarly all the beliefs previously discovered that are dependent on the new assumption, must be believed again. This is known as belief revision and involves propagate-IN and propagate-OUT. If an assumption is first believed, then disbelieved, and then believed again, we must rebelieve the facts that are based on it.

Recent research [Petrie 87] has suggested more advanced methods for deciding on a culprit. This could include advice stored at the point of the contradiction, suggesting ways in which this contradiction could be resolved [Marcus et al 88].

When a contradiction is reached, all the assumptions upon which the contradiction is dependent, are put in a NOGOOD set. This combination of assumptions will never be tried again. Contradictions are remembered in two ways :

- by contradiction assertions which are placed in the structure at the point of the assertion
- by NOGOOD assertions which record the same information in a form easily used by the routine which tries alternate state-assumptions.

A NOGOOD assertion explicitly lists the choices and atomic facts that led to a contradiction. It shows that context

(set of assumptions) leads to a contradiction. This information resides in a global context and its information is thus available to all contexts. When the system needs to make a choice it must examine the NOGOODS to eliminate any choices that are already known to be incorrect.

As mentioned previously belief revision is done when new information contradicts previous beliefs. There are several ways to resolve a contradiction :

1) reject the belief that the action took place for example, if during planning, a contradiction is encountered due to thinking a series of actions, the planner should reject one of the actions and try another.

2) reject the previous belief and say the action made a change in the world, or a previous assumption was corrected by the observation. For instance in the above example a contradiction could be resolved by deciding that some assumption about the world was wrong rather than by believing that we didn't perform some action.

5.6 Other Truth Maintenance Systems.

In this chapter, the JTMS was looked at in detail. There are however, other truth maintenance systems. These include the logical truth maintenance system (LTMS) [Charniak et al

87] and the Assumption based truth maintenance system [De Kleer 86a].

The JTMS has certain disadvantages. The most relevant being

- it cannot discover contradictions that are not explicitly told to it.

- the system has no concept of negation and therefore cannot decide if a proposition P and its negation (not P) are believed at the same time.

The JTMS can be extended to a LTMS (logical TMS) by providing each node with two labels, namely positive and negative [Mcallister and McDermott 88b]. Each belief represented by the node can be believed positive, believed negative, unknown or believed positive and negative which leads to a contradiction.

The LTMS can in fact be viewed as a JTMS with each node representing two propositions one being the negation of the other, and a constraint saying both can not be believed at the same time. This allows the LTMS to automatically find contradictions. The LTMS has one drawback it is slow to switch assumption sets [Mcallister and McDermott88b] as this requires a propagate-out and then a propagate-in. The ATMS [de Kleer 86b] overcomes this drawback by labelling the nodes, not just with their truth values in the current assumption set, but with respect to all assumption sets that have been mentioned.

In the ATMS the positive and negative labels consist of expressions that describe which assumption sets they are believed in and which they are not. A context switch requires the system to recompute which nodes are valid in the new assumption set.

In the ATMS each belief is labelled with the set of assumptions (context) under which it is believed. This set of assumptions is worked out by the ATMS from justifications that it receives. The assumptions are the primitive data from which all facts are derived. The contexts, represented by the assumption sets, can be manipulated far more easily than the datum they represent. The justification network with all the labels represents many possible worlds and thus the overall database does not need to be consistent. It is easy to refer to contexts and moving from point to point in the search space requires very little work.

A contradiction is recorded in its most general way in order to rule out as much of the search space as possible. This is done by recording the most general (supercontext) set of assumptions that led to it.

[De Kleer 86c] defines a interface between the ATMS and the inference engine. This interface is based on what has become known as the consumer architecture. In this architecture the TMS notifies the inference engine about changes in the state of some TMS node. This triggers off inferences.

Inferences are triggered via an "establish-consumer" function that has a boolean condition and a function. After each justification or assumption is added to the system all the boolean conditions (of the establish consumer functions) are checked. When one is found to be true the function is executed. This function returns a set of nodes, which are justified with the boolean condition [McDermott & Mcallister 88c].

The consumer architecture for a rule-base system would need:

1) a function that, given an assertion, finds the TMS-node corresponding to it or creates one as necessary (n).

2) When this is done for each rule whose left hand side matches the assertion in this node (n) establish a consumer for the node (n). This consumer uses that node plus the rules that use it on their LHSs, to support a node representing the conclusion of the rule (n+1).

3) This process is then repeated with this new pattern.

In this way, the ATMS controls the inference process. The consumer architecture is not quite as simple as has been outlined. There is the need for some sort of scheduling routine to work out the order of execution of pending consumers and it is sometimes necessary to build dummy nodes to hold functions.

5.7 Conclusion.

In a TMS based system there are two components, a problem solver that draws inferences and a TMS that records these deductions.

The TMS has 3 main roles ;

1 - it serves as a cache for inferences so that an inference once made, need not be repeated and a contradiction once discovered need never be discovered again.

2 - it allows the problem solver to make non-monotonic inferences eg "unless there is evidence to the contrary assume A". The presence of these non-monotonic inferences requires that the TMS use a constraint satisfaction procedure, called truth maintenance, to determine what data is to be believed.

3 - ensures that the database is contradiction free. Contradictions are removed by finding justifications whose addition to the database will lead to their removal. The process of finding and adding these justifications is called dependency directed backtracking. These new justifications are added to the antecedents of the non-monotonic justifications.

The interface between TMS and inference engine has problems associated with it most of these problems concern control eg

when backtracking, making a new choice and continuing where does the system continue from, especially if the choice has been made before and it is unOUTing.

[Martins and Shapiro 88] categorized 5 problems that a belief revision system must tackle.

1 - Inference : belief revision systems must keep track of where the facts represented in the knowledge base come from. This dependency record should be done automatically by the system.

2 - Nonmonoticity : it is useful to be able to record if one belief depends on the absence of another. If this latter belief becomes believed then the former becomes disbelieved. This leads to the idea of making a choice between options.

3 - recording reasons for belief : there are two ways to do this. In JTMS the support for a proposition are the propositions that directly caused it. In the ATMS [De Kleer 86a] the support for a proposition is the hypothesis that produced it.

4 - disbelief propagation : concerns updating the knowledge base when some proposition is no longer believed. In the JTMS the propositions were marked as believed or not. In the ATMS the knowledge base retrieval function knows what

context it is working under and it thus decides dynamically what propositions should be considered.

5 - belief revision : this is the main task of the system and it uses the previously defined features. It involves choosing a culprit for a contradiction revising the beliefs accordingly and making a new assumption.

PART TWO

CHAPTER6 : INTRODUCTION FOR PART2.

Part2 of this thesis will demonstrate how the research discussed in part1 was applied to the development of the WISE shell. The WISE shell is a large product but this thesis will concentrate only on the three areas that have been researched, namely : representation, control and truth maintenance.

The representational structures that are used are described in chapter 7 and the WISE language is described in [appendix 1]. Control of a WISE consultation is discussed in chapter 8 and the WISE TMS in chapter 9. Chapter 10 deals with some of the other modules of the WISE system. Chapter 11 considers some of the interesting implementation details, such as the language used to develop the system and the internal data structures of the system. Chapter12 looks at some of the applications that have been built on the WISE shell.

The motivations for many of the features have been laid out in part1 and will not be repeated here. The theoretical foundations for the control and representational features are well documented as there is very little new in these sections. There is a small section in the TMS chapter which outlines the theoretical background for the WISE TMS.

The WISE shell attempts to supply the knowledge engineer with easy to use and flexible features that will enable the development of large expert system applications.

CHAPTER7 : KNOWLEDGE REPRESENTATION IN THE WISE SYSTEM.

7.1 Introduction.

The WISE system uses a mixed knowledge representation scheme. This allows the knowledge engineer to express procedural and descriptive knowledge in a simple manner.

A WISE knowledge base consists of a hierarchy of frames and an agenda. All the descriptive and procedural knowledge needed to execute a consultation is incorporated in the frames. The descriptive knowledge is described in the attributes and relations of the frame, while the procedural knowledge is described in procedural attachments, entrance conditions, exit actions and rules. The control knowledge is described both procedurally and descriptively.

7.2 The WISE Knowledge Base.

A knowledge base [diagram 7.1] is made up of interrelated frames. Every knowledge base has a name and at least one frame called "agenda".

A frame has several fields including : relations to other frames, attributes, control knowledge, entry conditions, exit actions and rules. The frame is the basic unit of knowledge in the system and it is used to represent an object or concept.

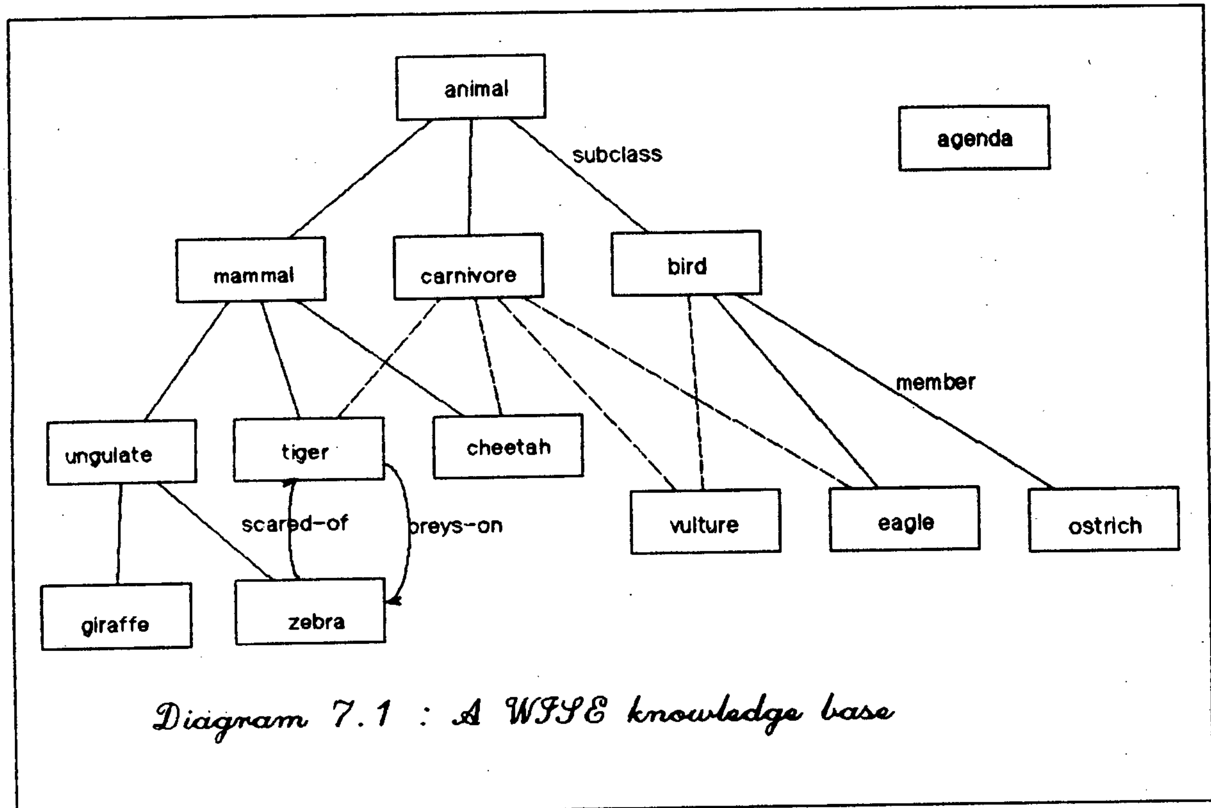


Diagram 7.1 : A WISE knowledge base

7.3 Inter frame relations.

The frames are related via system and user defined relations. The system relations are used to form the hierarchy along which inheritance of attributes occurs. There are four types of hierarchical relations: superclass, subclass, member and set. A frame has two type of slots,

own and **inherited**, only the slots of type inherited are inherited.

The **subclass** and **superclass** relations are between two generic or concept frames. They are symmetrical opposites ie if x is a superclass of y then y is a subclass of x. The "subclass" relation represents the sub set or sub concept relation for example, in [diagram 7.2], the relation between the frame "mammal" and the frame "ungulate". The subclass of a concept inherits the slots of type **inherited** from its superclass. Thus the "ungulate" concept inherits the slot called "markings" from "mammal" [diagram 7.2]. When inheritance is from one concept to another the slot is of type "inherited" in the child frame. The children of ungulate can thus in turn inherit the slot from it.

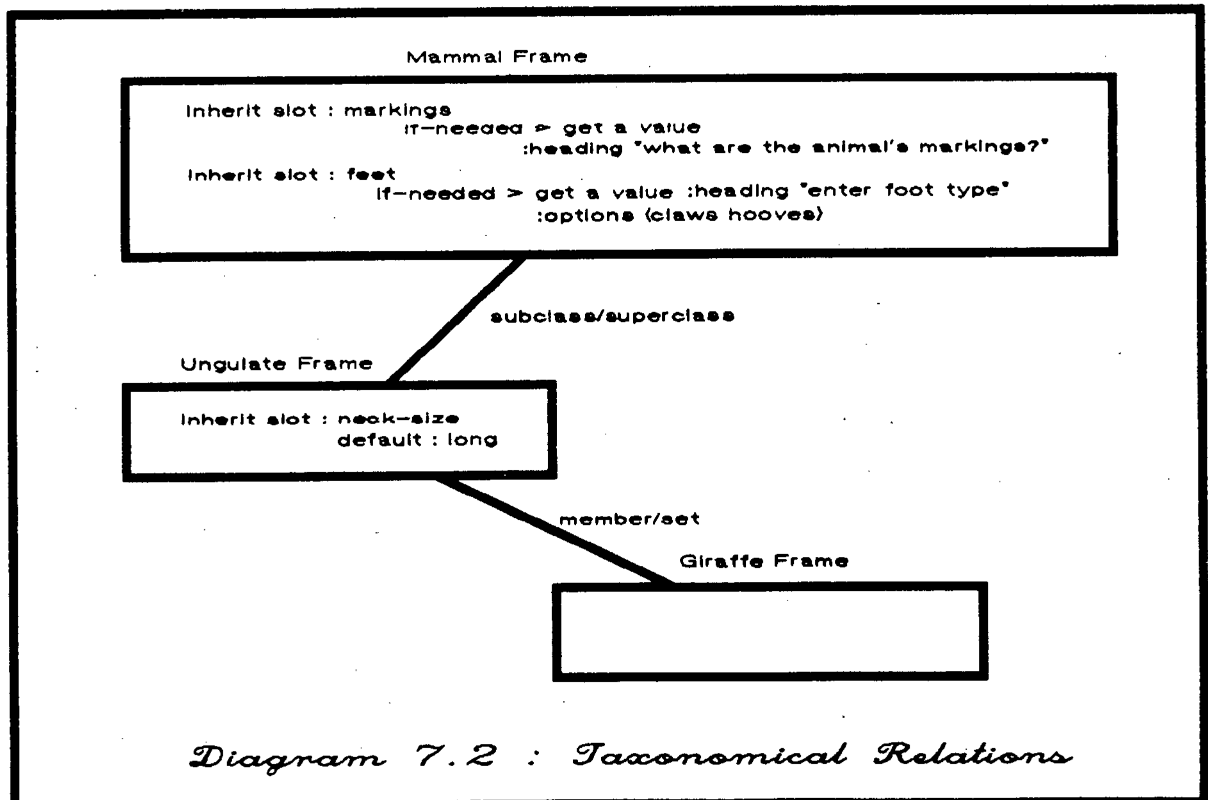


Diagram 7.2 : Taxonomical Relations

The **set** and **member** relations are between a generic frame (representing a class) and an instantiated frame (representing an individual). These two relations are opposites ie if "giraffe" is a member of "ungulate" then "ungulate" is a set of "giraffe". The member of a set inherits attributes from its set. When inheritance is from a concept to an instantiation the slot is of type own in the child frame. For example in [diagram 7.2] "neck-size" is inherited by "giraffe" from "ungulate".

Apart from the system defined hierarchical relations there is also a facility that allows the knowledge engineer to define his own relations. This allows for the representation of a semantic net like structure. In [diagram 7.1] there is a user defined relation called "preys-on" between tiger and zebra. These relations can be referenced in rules. For example a rule with the clause:

If : tiger preys-on zebra

will return true.

The semantic net structure is a neat, easy to use and powerful tool that allows the knowledge engineer to express arbitrary relations between objects.

All relations are set and changed dynamically during a consultation. If a new frame is created it is related to its generic frame via a member relation. As new relations between objects are discovered they are set. The structure of the knowledge base can thus change during the course of a

consultation. There are special commands for adding and removing the semantic net relations.

7.4 Own and Inherited slots.

Slots are used to describe the attributes of a frame. There are two types of slots, own and inherited. **Own** slots are used to describe an attribute that refers to the frame but not to any of its children. For example the "animal" frame has a slot called "possible-types" which refers to the "animal" frame but not to its children [diagram 7.3].

```
Inherit Slot : type
                retained : 'yes
                if-needed ()
                if-added execute-rule check-possible-types
                if-removed ()
             : teeth
               default : 'pointed
    own slot : possible-types
                value : '(giraffe zebra cheetah ... ostrich)
```

Diagram 7.3 : Own and Inherit slots in frame animal

An **inherited** attribute is used to describe an attribute that is inherited by the children of a frame. For example the

"type" slot in the "animal" frame is inherited by the children frames.

A slot is made up of a number of fields. A slot's field is called a facet. The own and inherited slots have similar facets namely value, retained facet, default and procedural attachments, the inherited slots also have an exception facet.

The value of an attribute or slot is kept in the **value facet.** A slot can take a single value or multiple values. A multi valued slot has a list of values as it value. The cardinality of a slot refers to the number of elements in a slot's value facet. In [diagram 7.3] a slot like "type" has a cardinality of one, whereas a slot like "possible-types" has a cardinality of 7. As values are added and removed form a multi value slot the cardinality changes.

The WISE shell has s procedural language [appendix 1] that provides commands to retrieve the value or the cardinality of a slot. There is also an ordinal retrieval that can access a value in a particular position in a multi-valued slot.

In [diagram 7.4] the first clause causes the retrieval of the value of the slot "coat" in the "animal" frame and compares it to the value "hair". The second clause is a retrieval of a multi-valued slot called "possible-values"

and a check to see if the value of the slot "type" is one of these possible values.

The value of a slot can be of any type ie a number, character or alphanumeric string. The type of a slot is not fixed and can change during a consultation, consistency checks are done using the procedural attachments which are discussed below.

```
If    the coat of the animal is 'hair
and   the type of animal is one of possible-types
Then enter the mammal frame
```

Diagram 7.4 : Slot Retrieval

The **retained facet** is shown in slot "type" in [diagram 7.3]. This facet gives the knowledge engineer a level of control over the way that an attribute's value is handled. The knowledge engineer often wants to set the value of an attribute and then later use that previously set value, in this case (which is the default case) the retained facet is set to 'YES. When a value is put in the value facet it remains there for use by other rules in the system. Sometimes the knowledge engineer wishes a value to be dependent on the **current state** of other values. In this case every time the value is accessed it is calculated according to some method. This method is stored in the if-

needed procedural attachment. In this case the retained facet is set to 'NO.

The retained facet is particularly useful in an inherited slot that is used by a number of different children. In each child frame the slot must have a value that is dependent on some value in the child frame. For example a knowledge base that uses a "profit" slot [diagram 7.5] would want the value of that slot to change as the value of "income" and "revenue" changes. If this slot were to be inherited its value would be dependent on the value of income and revenue in the **inheriting** frame *even if it had been set* in the parent frame.

```
Inherit Slot : Profit
               if-needed : income - cost
               retained : 'no
```

Diagram 7.5 : Slot with non retained value

The **default facet** is used to store a default value which can be a single value, a list of values, character, string or ordered list of alternatives. If the value of an attribute, needed by the system during a consultation, cannot be found then the default is taken.

The default can also treated as an assumption by the system. In this case it is the equivalent of: if there is no evidence to the contrary take this value and if we later prove this to be incorrect revise the beliefs accordingly.

The default can also be a list in which case the system treats it as an ordered list of alternatives, ie it tries the first and if this leads to a contradiction it tries the next. I will discuss the implications of this in the section on the TMS.

The **exception facet** only appears in inherited slots and is used to stop this slot being inherited by certain frames. It allows defaults with exception for example in [diagram 7.6] the "bird" frame has a slot called "transportation" which is inherited by its children. However there are certain birds that don't fly, such as ostriches, in order to stop these frames inheriting the "transportation" slot, they are listed in the exception facet.

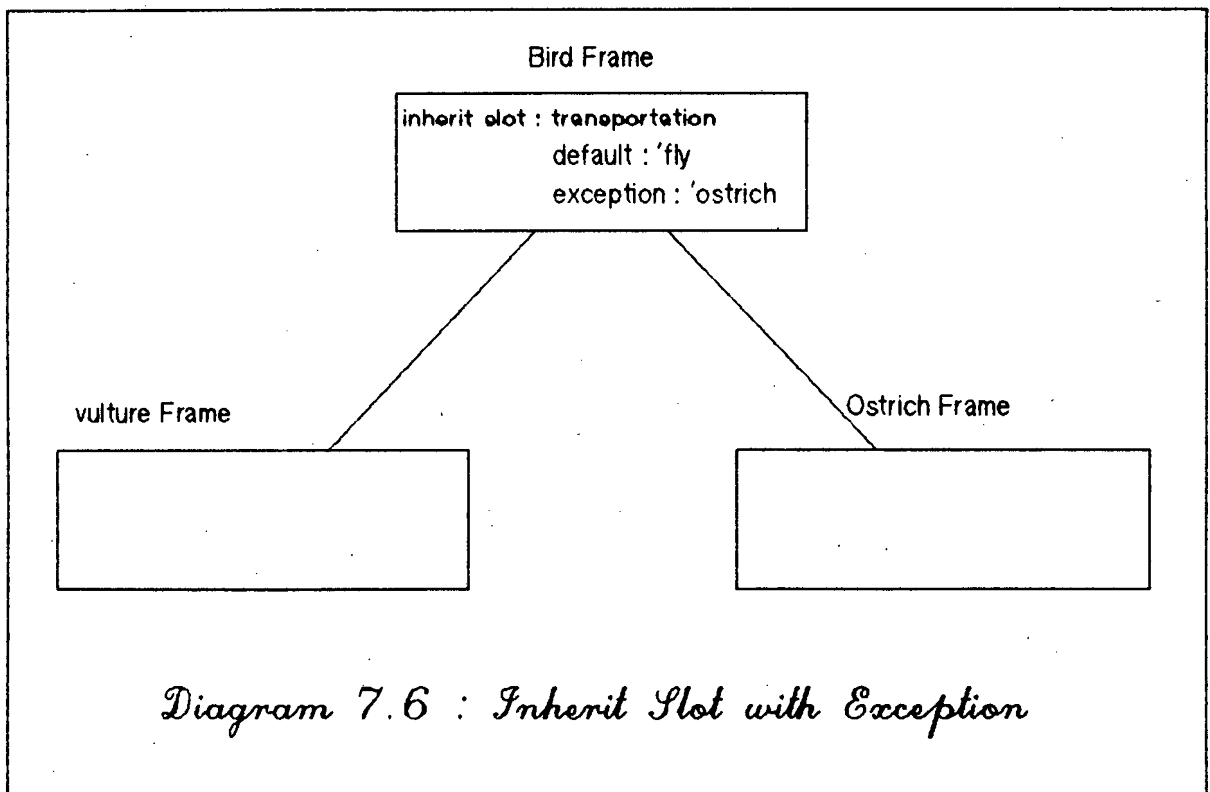


Diagram 7.6 : Inherit Slot with Exception

Procedural information is most efficiently used when it is in its correct context. **Procedural attachments** are methods

related to the slot. The WISE system has 3 types of procedural attachments :

1) The **if-added** procedure is a method that is executed when the value, of the value facet, of the slot, is changed. The procedural information here usually involves some type of consistency check. The WISE language has commands for checking the type of a value, the cardinality of a value etc.

2) If a slot's value is needed but the value facet has no value in it then the **if-needed** procedure is executed. The method in this facet returns a value. This procedural attachment will usually consist of a calculation or a request to the user to input a value. For example the "profit" slot in the "course" frame [diagram 7.5].

3) The if-removed facet is a method that is executed when the value of a slot is removed. Removing the value of a slot involves setting it to NULL. Which is a system defined value equivalent to the Lisp null or ().

Slots with procedural attachments are shown in [diagram 7.3] and [diagram 7.5].

7.5 Inheritance.

As previously mentioned the WISE shell supports inheritance of attributes along the frame hierarchy. When a slot is inherited by a child frame it is copied to this frame. This is necessary as each time the slot is inherited by a frame, the inheriting frame will most likely set it to some new value. The generic frames represent a generic occurrence with default values. Each instantiation inherits the slot and puts in its own values. For example, if the "food-source" slot where to be accessed by a rule in the "vulture" frame its value would be "scavenger" whereas in the "eagle" frame it would be "hunting".

In keeping with the concept of explicit control, the knowledge engineer is supplied with two types of inheritance search strategies, namely depth first and breadth first. Breadth first is the default strategy, this ensures that the frame will inherit from the closest relative with that slot. In certain circumstances the knowledge engineer set the parameter to depth first. This could speed up the process of finding where the slot will be inherited from.

A frame may have more than one parent with slots of the same name. In this case the first frame with a slot of that name, found by the inheritance search routine (depth first or breadth first), is copied into the inheriting frame. For example in [diagram 7.1] if "tiger" is inheriting a slot

with a "breadth first" strategy then the system will look in "mammal" then "carnivore" and then "animal", if the strategy is depth first it will look in "mammal", "animal" and then "carnivore".

When a frame inherits a slot it inherits not just the value, but the defaults and procedural attachments as well. This can lead to problems with inheritance. For instance when the frame "ostrich" inherits the default attribute "fly". In order to represent this type of knowledge inherited slots have a facet called an exception facet in which the knowledge engineer can list the frames that do not inherit this slot's default. This facet can be set dynamically during a consultation. A knowledge engineer can thus specify a generic concept such as "bird" with the attribute that it fly's. The "fly" attribute has an exception facet that explicitly lists frames that must not inherit the default value [diagram 7.6].

7.6 Entry-conditions and Exit-actions.

A frame is a module of knowledge that is considered in a certain context. The entry-conditions are a set of boolean clauses which are checked before a frame is entered. If they are all true then the frame can be entered and its

rules are considered. If one or more of the entry conditions are false then the frame is exited.

When a frame is entered the control knowledge for the frame is considered and the rules in the frame are fired. Entering a frame thus causes the inference engine to focus attention on the rules in that frame and consider weather they might be relevant for execution. When there are no more rules to fire the frame is exited. Before exiting a frame the exit actions are performed. The exit actions are a list of actions that are only executed when the frame is exited.

Entry conditions and exit actions fit into the idea of storing procedural knowledge in the place where it would be used. The procedural knowledge pertaining to weather we enter a frame or not ie the module :

```
If conditions
then enter-frame F1
```

is considered only in the context of trying to enter F1.

The procedural knowledge pertaining to what actions to perform when there are no more rules to fire in a frame ie

```
If   no more rules to fire in F1
then actions
```

is considered only when there are no rules to fire in F1.

7.7 Rules.

Procedural and control knowledge is captured in a high level language called the WISE language. This language is compiled to an internal form that is used by the inference engine. [Appendix 1] shows the compiled form of the clauses. One of the main motivations of the WISE knowledge representation is that procedural knowledge should be placed at the point where it is to be used. There are several places to position procedural knowledge these being procedural attachments, entry conditions, exit actions and rules.

Rules are stored as part of a frame, and are only considered for evaluation if the frame they are in is entered or considered. This point is important as at no point does the inference engine have to consider the entire rule set when deciding what rule to fire next.

By placing the rules in frames the knowledge base is effectively partitioned. The entire rule set is thus never considered at one time. This greatly simplifies the problem of saturation. For example [diagram 7.7] as soon as it is discovered that the "animal" is a "mammal", the "mammal" frame is entered and the rules that deal with the "bird"s are not looked at.

A rule is shown in [diagram 7.7] it consists of an **If part** and a **Then part**. The If part of the rule states a set of conditions in which the rule is applicable. The Then part is the appropriate action or conclusion to make when the conditions are satisfied.

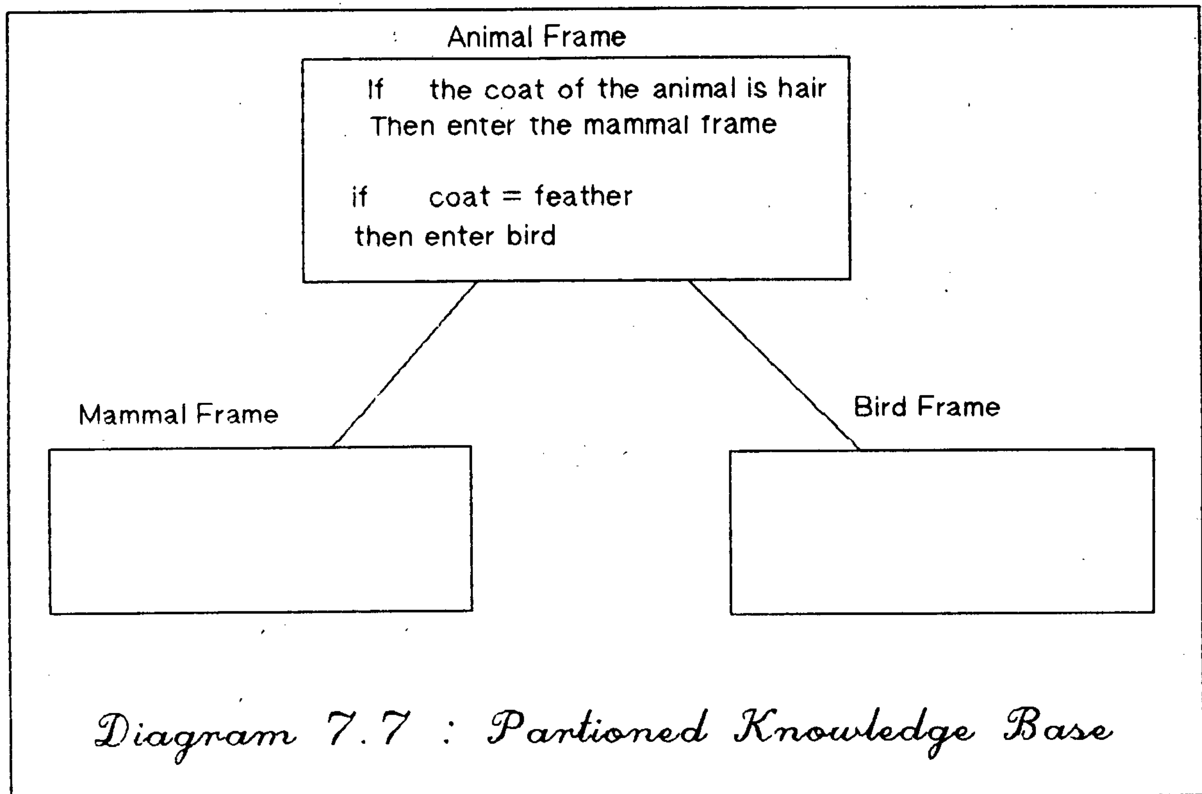


Diagram 7.7 : Partioned Knowledge Base

As previously mentioned there is a high level language in which the rule clauses are written. The clauses can in fact be written either in this pre-defined language, which is functionally complete, or in LISP. The **if-clauses** are boolean clauses ie they return true or false. A set of if-clauses must all be true for the rule to be valid. For example in [diagram 7.8] the first boolean clause states: if the value of the slot "coat" is "hair" then this clause is

true. This is known as a comparative clause. The second says if the "carnivore" frame is related to, the frame whose name is in the slot "type" of frame "animal", by a member or subclass relation then it is true. This is known as a **relational** clause.

The comparative clauses have a left hand side (LHS), a right hand side (RHS) and a comparative operator such as =, >, < etc. The LHS or RHS of such a function consists of an expression. These clauses can be written in shorthand as shown in [diagram 7.8].

the coat of the animal is hair
the (type of animal) is a carnivore

animal.coat = 'hair
animal.type is–a 'carnivore
f1.s1/f2.s2 = f1.s2 + 6

Diagram 7.8 : If clauses

The THEN-CLAUSES represent actions that the system must perform when the if-clauses are true. There are several types of actions, namely input actions, output actions, actions that change the knowledge base, control actions and generating a contradiction [diagram 7.9].

The input and output actions are used to communicate with the user or any external data source such as a database or unix file. The input actions are used to prompt the user for an input, there is a command for a simple prompt, or a prompt with a menu of options. For example the slot coat in the "animal" frame has an if-needed facet that asks the user for a value. The "teeth" slot prompts the user for a value from the list (pointed square) [diagram 7.10]

```
set the type of animal to 'carnivore
display animal.type
add cow to possible-types
remove relation 'preys-on 'zebra from 'tiger
```

Diagram 7.9 : Then clauses

```
Inherit Slot : teeth
        retained : 'yes
        if-needed : get the value
        :header "what type of teeth does the animal have"
        :options (pointed square)

Inherit Slot : coat
      retained : 'yes
      if-needed: get the value
            :header "What type of coat does the animal have?"
```

Diagram 7.10 : Input commands

The output commands are used to output information to the user in a structured and readable way. There are a series of such commands including commands that allow a frame, slot etc to be shown to the user [see appendix 1].

There are several commands that change the knowledge base including commands for changing a slot's value setting up global variables, changing relations between frames and creating instantiations of existing frames. Changing a slot's value is done using one of the set commands [see diagram 7.9]. These allow for the setting of an own or inherited slot's value as well as the adding or removing from multi-valued slots. The WISE language also supplies commands that can dynamically (during the course of a consultation) change the control parameters of a frame.

The language provides for variables that are kept in a symbol-table. The scope of the variables is global and they can be used to pass values between rules etc. There are two types of special variable choice-variables and loop variables. These variable allow for universal and existential quantification. The choice and loop variables will be discussed in more detail. A variable's value is set using the set-var command.

There are also a series of commands that change the structure of the knowledge base such as adding and removing relations, creating and deleting frames. A full description of the WISE language is shown in [appendix I].

7.8 Conclusions.

The WISE systems knowledge representation is a synthesis of rules and frames. By placing the rules in frames the advantages of both representations are supplied to the knowledge engineer. The representation attempts to supply the knowledge engineer with tools to model his application domain in a simple manner.

CHAPTER8 : CONTROL OF REASONING IN THE WISE SYSTEM.

8.1 Introduction.

The representation of problem solving knowledge is only useful with good control of reasoning. The knowledge engineer using the WISE shell has a high degree of explicit control over the order of actions that the system performs. The basic principle of the WISE shell's reasoning mechanism is to give the knowledge engineer explicit choice about what reasoning tools he wishes to use. However, if he does not wish to worry about this, the system will automatically set intelligent defaults.

There are two levels of control in the WISE system, global and local. Global control refers to control that the system has over the way it approaches a consultation. Local control refers to the control of the order of rule invocation within a frame.

8.2 Forward and Backward chaining.

The overall global control structure of the WISE system is a mixture of forward and backward chaining. Forward chaining

or, data driven inferencing, allows the inference engine to discover all the conclusions that can be drawn from the data. Forward chaining suffers from the problem of often doing irrelevant inferences. In order to focus the attention of the WISE forward chainer the knowledge engineer is supplied with features that allow him to force the system to consider only relevant rules. Backward chaining, or goal driven control, allows the inference engine to discover if a particular goal can be concluded from the data at hand. It is useful for proving something correct. The WISE system supplies both a forward and a backward chainer.

```
Entry Conditions : Start = 'yes

 Own-slot : start
      if-needed get a value :header "do you want to start?"

Rules :
Rule-1 : If consult-type = 'find-species
          then animal.coat := get value :header "enter coat type"
                enter animal

rule-2 : If consult-type = 'describe-animal
          then display animal.type
```

Diagram 8.1 : Agenda Frame

A WISE consultation starts by entering the **agenda** frame. This is a special frame that is part of every knowledge base. The agenda has explicit control commands which tell the system how to go about solving the problem ie it has

meta-level knowledge. An example of an agenda frame is shown in [diagram 8.1].

The agenda can be used to implement a multi layered control structure ie it can cause the system to enter other frames that have meta-level control. For example the "animal" frame [diagram 7.7] is called from the agenda and it too acts like an agenda in that it controls the way in which the consultation continues by deciding whether to enter the "bird" or "mammal" frame. This allows for hierarchical planning and a general top-down approach to knowledge engineering.

A consultation starts by executing the rules in the agenda frame. These rules are control rules which tell the system how to continue the consultation ie which frames to consider next [diagram 8.1]. As these frames are entered their rules are executed in a forward fashion.

The rule-base is partition in frames and these frames are entered in different contexts. The knowledge engineer can ensure that rules which perform a common task, or are associated with a common object, are considered together by explicitly placing these rules in a frame. This can reduce the problem of the system doing irrelevant inferences. An inference is considered relevant by virtue of the fact that it is in a frame that is entered. Within the frame there is further control over which rules are relevant. This level

of control will be discussed in more detail under the Frame level control section.

However, to find out if a goal is valid, backward chaining is often required. In this case the goal is entered in the agenda, with the goal taking the form of "slot = value" or "frame relation frame". For example the second rule in agenda frame [diagram 8.1] has a consequent "display animal.type" that will cause the system to try and find the value of the slot "type" in the frame "animal". If this knowledge is not immediately available, ie in the frame's slot, the system will back chain ie it will find all the rules that could set this value and see if any are true. In the example animal.type is set to the value 'carnivore in the carnivore frame [diagram 8.2] and in order to get this value the value of "teeth" and "claws" are needed. These slots are inherited and the user is asked for their values (if-needed). If the rule that sets the slot's value fails (its antecedents are not true), another rule that sets the slot will be found and tried. If no rule that sets this slot succeed then the system takes the default value.

If teeth = 'pointed
and feet = 'claws
then animal.type ='carnivore

Diagram 8.2 : Rule in carnivore frame

The backward chainer must beware of self referencing rules or clauses as these will cause it to loop. The backward chainer decides which rules set the value of a slot by looking at that slots index. A slot's index is the address of a set of rules that set the value of this slot. Indices are discussed in more detail below.

```
own-slots : type
          : species

inherit slots : coat
              : if-needed ask-user
              : if-added execute check-coat
              : default hair
inherit slot  : teeth
              : if-needed ask-user
              : if-added execute check-teeth
              : default pointed

inherit slot  : feet
              : if-needed ask-user
              : if-added execute check-feet

Rules :
     R1 if   coat = 'hair          R3 if   type = 'ungulate
        then enter mammal             then enter mammal
     R2 if   coat = feathers
     then enter bird
     R4,R5,...R100
```

Diagram 8.3 : Animal Frame

The WISE system's normal mode of operation is a mixture of forward and backward chaining. The system will forward chain until it needs the value of slot and cannot find it. It will treat finding this value as a goal and feed it to the backward chainer. This happens automatically if the if-needed fact fails to find a slot's value. For example in the "animal" frame [diagram 8.3] while executing a rule in a forward direction the system requires the value of "animal.type", this slot has no value or if-needed method

and the system thus has to backward chain to find its value. This mixture of forward and backward chaining is called automatic goal generation.

8.3 Frame level control.

Placing rules in frames solves the problem of saturation (too many plausible inferences) by:

1 ensuring the retrieval of only relevant knowledge. This is done by placing knowledge in the context in which it will be used, and

2 refining this set of relevant knowledge even further through conflict resolution strategies and other control knowledge.

Within each frame there is a control block that specifies control strategies for that frame. These control strategies are shown in [diagram 8.4].

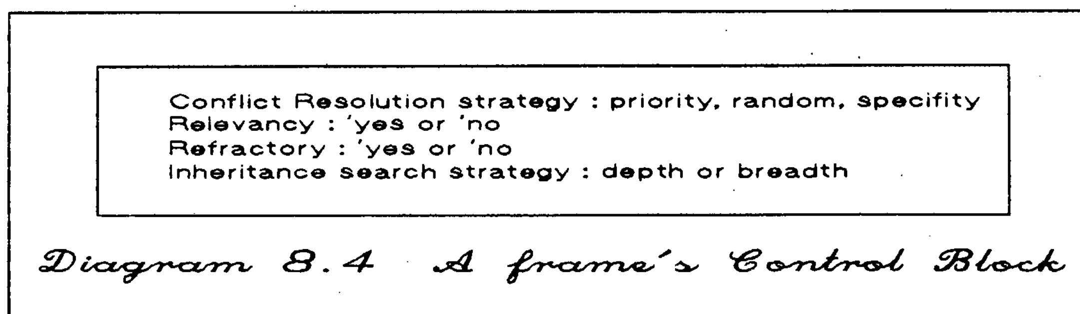


Diagram 8.4 A frame's Control Block

The **conflict resolution strategies** allow the system to decide what rule to fire out of a set of plausibly useful rules. The set of plausibly useful rules is formed prior to checking if the antecedents of the rule are true. Thus rules that are not true might be considered plausibly useful. When it is discovered that their antecedents are not true then they are removed from the conflict set. The conflict resolution set is formed from the rules in a frame. Which rules will go into the set depends on the value of the relevancy and refractory parameters.

The **relevancy parameter** gives the knowledge engineer a further level of control over which rules will be considered relevant by the conflict resolution strategy. The system keeps an internal structure called the relevancy list. This notes which rules in the knowledge base are affected by the items that are being changed. As an item is updated the addresses of all the rule clauses that access this item are stored in the relevancy list. When a frame is entered the relevancy parameter is considered, if it is set to 'yes then only the rules that are in the relevancy list are put in the conflict resolutions set.

The relevancy parameter allows the system to go into a mode where it can follow a line of reasoning. It will ignore superfluous rules. This allows true data driven inferencing as the newly discovered data drives what rules will be considered relevant. The set of static knowledge is not used to decide the relevancy of a rule; only rules that

access items in the dynamic or changing part of the knowledge base are considered. The default value of this parameter is "no".

Consider the "agenda" frame [diagram 8.1] and the "animal" frame [diagram 8.3]. The rule in the agenda frame discovers that the "consult-type" is "find-species" it then sets the value of "coat" in the frame "animal". When it does this all the rules that use this slot are stored in the relevancy list. When we enter the frame "animal" it has a relevancy parameter set to "yes", now even if it had a 100 rules in it only the rules that access the "coat" slot go into the conflict resolution set. In this way the problem of the system doing irrelevant inferences is resolved.

The **refractory** parameter allows the knowledge engineer to control the amount of times that the same rule will be fired. If it is set to "yes", then each time a rule is fired, it is put back in the conflict resolution set to be considered once again. If it is "no" then once a rule has been fired it is removed from the conflict resolution set and not considered again until the frame is explicitly re-entered. The default value is "no". In [diagram 8.3] the "animal" frame has a refractory parameter set to 'no. Thus, when a rule is returned by the conflict resolution strategy, it is removed from the set and not tried again.

Several conflict resolution strategies are implemented in the WISE system. These include : random, specifity and

priority. The **random** strategy chooses an arbitrary rule from the conflict resolution set. If this rule is true then it is fired else another rule is picked. Depending on the value of the refractory parameter the first rule will either be put back to be tried again or left out of the set. This strategy is often used with the relevancy parameter set to 'yes.

Specifity is a strategy that allows the knowledge engineer to indicate that rules that are more specific must be tried before rules that are less specific.

The most commonly used conflict resolution strategy is **priority**. Each rule has a numeric priority associated with it (default 5). The knowledge engineer sets initial priorities at set up time, however the priorities can be dynamically changed as the consultation progresses. In all these strategies, if a rule chosen by the conflict resolution strategy fails then the system will return to the set and apply the strategy again to get another rule to fire.

All these control parameters are settable during a consultation. This allows the knowledge engineer to reason about which control strategies to use.

All the above control strategies are available to the knowledge engineer. If he does not wish to use them then they are assigned defaults. In fact, if the knowledge

engineer so wished, he could put all the rules of the system into one frame and leave all the control at its default. In this extreme case none of the WISE systems control mechanism would be taken advantage of and the system would act like a simple rule based system.

8.4 A WISE Consultation.

A WISE consultation comprises of entering the agenda frame and executing the rules that reside there. The relevancy in the agenda frame is set to "no" as the relevancy list has no elements in it when a consultation starts. If it were set to "yes" then no rules would be considered relevant and the exit actions would be executed.

8.4.1 Entering a frame.

Once the system enters a frame the procedure begins by first checking the entry-conditions. If they all prove to be true then the rules are considered. When a frame is entered, there are several control parameters that together decide which rule will be fired. These are the refractory, relevancy and conflict resolution strategy. The relevancy list is built as values are set. When a value is set, the

rules that access that value in their antecedents are added to the relevancy list. This is achieved through an indexing mechanism that links all the slots to the rules that use them.

The conflict resolution strategy for a frame is found in the control section. In [diagram 8.4] we saw the control block for the "animal" frame. It had it conflict strategy set to priority. This means that the rule with the highest priority is executed first.

8.4.2 Executing a rule.

Once a rule has been picked to be fired, the rule executor executes it. Executing a rule involves first determining the rule type. If it is a choice rule, then an assumption about the value of of one of the variable is made. If it is a loop rule, then the loop variable is bound to a value. The rule is then executed by checking its antecedents and if they are true executing the consequents.

8.4.3 Loop rules.

A **loop rule** is a special construct that allows the knowledge engineer to specify a series of values with which the rule

must be fired. A loop rule gets executed with a series of values one after the next. For example the loop rule in [diagram 8.5] gets executed for every instantiation of "animal". This is in effect universal quantification.

```
loop with ?var for every child of frame
loop with ?var for every instantiaton of frame
loop with ?var for every parent of frame
loop with ?var using loop-frame


loop with ?animal for every instantiation of animal
if    ?animal.teeth = 'pointed
and   ?animal.feet = 'claws
then ?animal.type := 'carnivore
```

Diagram 8.5 Use of loop rules

A loop rule has a loop-clause which binds a loop-variable to a value. A loop clause can take a number of different forms as shown in [diagram 8.5]. The first two clauses in [diagram 8.5] differ in that the first will assign the loop-variable the names of every frame one level down from the "animal" frame (mammal, ungulate,bird) while the second will loop with the instantiations of the "animal frame" ie every frame, below animal, that is at the leaves of the tree (giraffe, zebra, tiger etc). The last clause loops on the values in the multi-valued slot "possible-values". The rule in [diagram 8.5] will loop with the values : zebra, giraffe, cheetah, tiger, eagle, vulture and ostrich.

Once a rule has the values for its loop variables, these variables are instantiated. The instantiated rule is then executed. A rule is executed by checking its antecedents, if they are true then the consequents are executed.

8.4.4 Checking the antecedent

An antecedent or if clause in the source knowledge base is written in the WISE language, or in LISP. However, once compiled, it has several components that are put in by the compiler. These include a validity-flag, content and a type.

A **validity-flag** can take three values: **true, false** or **unknown**. Initially the compiler sets it to unknown. When a boolean clause is executed, it returns true or false. This value is then put into the validity-flag field of the clause. For example if a clause is true then the validity flag is set to true.

The validity-flag is set once the truth value of the antecedent has been found. The next time the rule is executed if the validity flag is not "unknown", the clause need not be checked again. If *any value that could effect the truth value of the clause* is changed, the validity flag

is set to back to "unknown". This is achieved through an indexing mechanism that will be discuss below.

Each compiled antecedent clause has a field which stores its **type**. A clause can be one of two types, namely, relation or normal. A clause of type relation deals with the relation a frame has with another frame. A clause of type normal deals with the relationships between the values of slots.

The content field has a list of all the items that this clause references. This is used by the TMS which is discussed in the next chapter. The function field holds the compiled function. In the compiled knowledge base, the clauses are lisp functions with additional functions that allow access to the knowledge base structures. A compiled antecedent is shown in [diagram 8.6].

```
If animal.coat = 'hair

function : (equal (f-get 'animal 'coat) 'hair)
contents : (animal.coat)
type     : normal
validity : 'unknown
```

Diagram 8.6 : Compiled antecedent

8.4.4.1 Normal Antecedents.

Normal antecedents are antecedents that do not access the relations between frames. These antecedents usually have a LHS and a RHS separated by some comparative operator. Their truth value is found by evaluating each side of the statement and then applying the comparative operator.

Evaluating an expression involves retrieving values form the knowledge base or symbol-table. Functions that return a value are called value functions. There are several value functions :

f-get : returns a value form a slot, or the cardinality of a slot.

var-get : returns a variable's value

num-child : returns the number of children of a frame

num-parents : returns the number of parents of a frame

input functions : return a user inputted value.

F-get [diagram 8.6] is a function that returns a value from a slot. This function is passed to the lisp evaluator and executed. Checking to see if an antecedent is true involves getting a value for the functions that return values and then executing the boolean function with these values. A reference to a slot without a frame causes the compiler to generate code as if the slot was in the current frame. For example the lisp evaluator gets the value from (f-get 'animal 'coat) and then sees if it is equal to 'hair.

There are several functions that return values namely f-get, var-get, num-child, num-parents. There are many boolean functions such as equal, >, <, rel-get etc. The value functions all require accesses to the knowledge base.

f-get returns a slot's value : it takes 2 parameters, a frame name and a slot name. Its algorithm is :

1) If there is a slot of this name within this frame goto 5.

2) Inherit the slot. This involves finding the inheritance strategy in the frame's control block, and applying it. If the slot cannot be found goto 11.

3) Once the slot has been located in an ancestor frame, its exception facet is checked to see if the frame attempting to inherit it, is an exception of this slot, if it is continue the search else copy the slot to the inheriting frame. If the slot cannot be found goto 11.

4) If the *inheriting* frame is a generic frame then put in inherited slot else put in own slot.

5) Look at the slot's retained fact if it is 'yes and the value facet is not equal to () then goto 10.

6) If the retained facet is 'no, or the value is null, the value must be obtained using the if-needed facet. The if-

needed facet must explicitly set the slot. Execute the if-needed, if the value of the slot is not null goto 10.

7) The backward chainer proceeds by looking in the then-index of the slot to find all the rules that set this slot to a value. These rules are put in a conflict resolution set and the frame's conflict resolution strategy is applied to them.

8) A rule is picked and tried. If it fails because one of the antecedents are false, the system will try another rule from the conflict set. The rule that sets the slots value might itself require backchaining in order to be executed. The backward chainer uses a depth first, backward chaining strategy. It stops when the slot's value is set. If the backward chainer finds a value then goto 10.

9) The default is taken as the value. This is equivalent to making a choice about the value that we wish the slot to take. It can be a single value or a list of possible values.

10) return value

11) report error

A slot may have more than a single value. These **multi valued slots** can be accessed using special multi-value functions. These include a cardinality function that

returns a number telling how many values the value facet holds. There is also an ordinal f-get for multi value slots that allows the user to access a particular element in the multi-valued-slot eg third element.

The compiled form of the cardinal f-get is (f-get 'f1 's1 'cardinal). The compiled form of the ordinal f-get is (f-get 'f1 's1 'pos).

The third parameter is an optional one that tells f-get to return either the pos^{th} value of the slot or the number of elements in the slot.

Apart from slots returning a value, there are also variables that can return a value. A variable's name starts with a "?". We have already mentioned one special sort of variable namely the loop variable. There are three types of variables: loop, choice and normal. The compiler can tell from the context in which they are used, what type of variable they are. The variable accesses are compiled to the **var-get** function which returns the value of a variable it takes 2 parameter in its compiled state [diagram 8.7]. Var-get retrieves the variable's value from the symbol-table.

Two other functions that return values are **number of children** and **number of parents**. These functions return the number of children or the number of parents of a frame. These are compiled to :

(num-child frame-name) and (num-parent frame-name) respectively.

```
If  ?x > f1.s1 -3
is compiled to :
(> (var-get 'x' 'normal) (- (f-get 'f1 's1) 3)))
```

Diagram 8.7 : Compiled form of Var-get

The last type of functions that return values are the input functions. These functions are part of the knowledge base's interface to the outside world. They can communicate with a person, database or file.

8.4.4.2 Relational antecedents

An antecedent can also be of type **relation** which means that it refers to the relation ship between two frames. This type of function is compiled to rel-get. This is a boolean function that is the compiled form of the relation statements such as:

If Jack is the brother of Jill

which is compiled to :

```
(rel-get 'Jill 'brother 'Jack)
```

It looks in the relation field in the "Jill" frame to see if there is a relation called "brother". If there is it looks to see if one of the subjects of that relation is "Jack". If it is, rel-get returns true, if not the system will attempt to find the value of the "brother" relation by backward chaining.

Relations are indexed to the clauses that use them, and the clauses that set them, these indices are the same as the slots' indices.

8.4.5 Executing a consequent.

If the antecedents of a rule are all true, then the consequents are executed. The consequents fall into three broad categories : change the knowledge base, input-output commands and control commands.

8.4.5.1 Changing the knowledge base.

There are several ways in which the knowledge base can be changed: one can change a slot's value, add or remove

relations between frames, add and remove frames, change a frame's control block knowledge and set a variable's value.

A slot's value is set using one of the set commands. There are set commands for setting the value of a single-valued slot, as well as adding or removing values from a multi-valued slot. An own or inherited slot's value is set using set commands as shown in [diagram 8.8]

A slot's value is changed using the f-put command which is generated from the set commands by the compiler. F-put has the arguments: frame-name, slot name, type and expression.

```
set f1.s1 := 5 * f2.s2
reset f1.s1
add the value 'x to the multi valued slot s1 in  frame f1
remove a multi f1.s1 'x

Compiled from :
(f-put 'f1 'own-slot 's1 (+ (f-get 'f2 's2) 6))
```

Diagram 8.8 Set commands

F-put works as follows :

1) If the slot is in the frame, goto 4

2) if slot is not in the frame, it is inherited, as in f-get, using the inheritance strategy of the frame which is either depth or breadth first.

3) The slot is copied into the frame. If the frame is generic it goes into a new inherited slot if the frame is an instantiation it goes into a new own slot.

4) Set value equal to the evaluated expression. This involves getting the values from the slots and evaluating the expression. The value is then put into the value facet of the slot.

5) Put the value in the slot.

6) If the value to which the slot is being set is NULL then the if-removed facet's procedural actions are executed. If the value is not null, the if-added facet's procedural actions are executed.

7) Reset validity flags of all the clauses whose truth value depends on the value of this slot.

8) Add the rules which use this slot to the relevancy list

9) stop.

The reset commands are used to set a slot or variable's value to NULL, which is a system defined value much the same as the lisp NULL or (). When a slot's value is reset the if-removed procedure is executed.

There are functions for adding and removing values from a multi valued slot. for example in [diagram 8.8] the third clause adds the value "x" to the slot s1 and the fourth removes the value "x" from the same slot. These functions are compiled to the form

add-a-multi (frame, slot, value) and

remove-a-multi (frame, slot, &optional value) will remove the first element from a multi-valued slot. Remove from multi can be called with a value as a third parameter in this case that particular value is removed from the slot.

The second class of functions that change the knowledge base are those functions that change the structure of the knowledge base. They do this by changing the relationship between frames, or adding new frames.

The **set relation commands** allow the knowledge engineer to dynamically change the relations of a frame. This very powerful command must be used carefully as it changes the structure of the knowledge base. For example in [diagram 8.9] as new relations between objects are discovered the links between these frames are put in.

```
set the relation 'member in frame 'tiger to 'carnivore

compiled from :
(rel-put 'tiger 'member 'carnivore)
```

Diagram 8.9 The set relation command

Apart from setting a relation between two frames there is also a command for removing such relations. The set relation and remove relation commands have 3 arguments : the frame the source frame, relation and object frame. They are compiled to rel-put and rel-removed as shown in [diagram 8.9.]

As mentioned earlier the relations are indexed to the rule clauses that use them. When a relation is set or changed all the clauses that access have their validity flags set to 'unknown.

Another very useful command that changes the structure of the knowledge base is the **create frame** command. This allows the knowledge engineer to create instantiations of the generic frames at run-time. Concept or generic frames store prototypical situations. These prototypical situations manifest themselves in actual occurrences of the concept eg BOB is a occurrence of the concept TEACHER this is known as instantiation.

The create-frame command takes two arguments : a generic frame and a new instantiated frame. The new frame is created from the generic one by putting all the inherit-slots of generic in own-slots of new. Relations, rules , entry and exit conditions are not put in the new frame but the control knowledge is.

In order to allow the WISE language to represent quantified statements the language has variables. There are several types of variables, namely loop-variable, choice-variables and global variables. Loop-variables are used in loop rules and allow the knowledge engineer to make universally quantified statements. The choice variables are discussed in the next chapter, and they allow the knowledge engineer to make existentially quantified statements. Global variables set up bindings using the set-var command [appendix 1]. Variables are treated very much like slots, in that they are indexed and when changed they update the relevancy list and validity flags.

The change-control commands allow the knowledge engineer to dynamically change the control parameters of a frame. There are commands for setting search strategy, conflict resolution strategy, relevancy and refractory fields in a frame. The priority of a rule can also be dynamically changed, thus changing the order of execution of the rules within a frame. This allows for a meta-meta-reasoning where the system can reason about what control strategies it should use.

8.5 Validity-flags and indices.

The entire WISE knowledge base is cross indexed ie there are indices from the slots, variables, and relations, to the rules that use them. For example slot "animal.type"'s index is shown in diagram 8.10]

```
own slot : type
                : if-index ((mammal (R1 1) (R2 1))
                            (bird (R1 1) (R2 1)))
                : then-index (carnivore (R1 1))
```

Diagram 8.10 Own-slot with index

The indices are set up by the compiler at compile time. Each clause that is compiled has all the items that it references stored in a contents field. Each member of the contents has its index updated with the rule-name and the position of the clause in the rule.

The **if-index** is the part of the index that has the address of all the if-clauses in which an item appears. It is used in conjunction with the validity flag of the clause. When a clause's truth value is decided, it is stored in the validity flag of that clause. That truth value should remain unchanged until any item that is referenced by the clause changes. When an item's value is changed, all the if-clauses that reference it have their truth value set to "unknown".

The **then-index** is the part of the index that has the address of all the then clauses in which this item appears. It is used by the backward chainer. When the backward chainer is attempting to find a value for this item, it uses this index to find all the places that the value of this item is set.

When an item is changed the if-index is referenced and all the rules that reference this item in their if clauses are added to the relevancy list. The relevancy list is organized according to frames and rules. It is used to decide which rules are relevant in a frame.

8.6 Control Actions

There are two explicit control actions that can be used to make meta-rules. These are execute a rule and enter a frame. The execute a rule command allows the knowledge engineer to specify a rule to be tried at a specific point. The enter a frame command is used to control which rules the system must consider.

The last action to be considered is the **contradiction** which is used to tell the system it is in a contradictory state. The implications of this are treated in chapter9 on the TMS.

CHAPTER9 : THE WISE TRUTH MAINTENANCE SYSTEM.

9.1 Introduction

The WISE shell uses a truth maintenance system (TMS) to implement non-monotonic reasoning and revision of beliefs. The WISE TMS is modeled on Doyle's JTMS. It is particularly well suited to handling problems that require local heuristics to solve global problems.

The WISE TMS has the generic operations that one would expect to find in any TMS [Mcallister and McDermott 88a]:

- make-TMS-node : adds a node to the TMS.
- add constraint : adds a justification which is a constraint on the belief values that a node could have.
- follows-from? (fact assumption-set) : returns true if the fact follows from the assumptions.
- justifying-literals (node assumption-set) : retrieves the nodes on which a node's truth is based.
- get-the-justification (node assumption-set) : retrieves the constraint on which a node's truth is based.

The WISE TMS does the basic functions that are required of any TMS :

- enforces logical relations between statements
- generates explanations
- finds solutions to search problems

- identifies cause of failure.

One of the specifications for the TMS was that a knowledge engineer must be able to use it without any knowledge of its low level workings. The TMS functions are abstract to the knowledge engineer. He concerns himself merely with building the knowledge base. Where he feels it is necessary, he can get the system to make a choice. In order to facilitate this, all the dependencies and contexts are automatically generated by the system.

9.2 The Theoretical Foundations of the WISE TMS.

The WISE TMS's theoretical foundations are based on the work of Doyle and Reiter. [Reiter 80] defines a default as the most likely or typical value. It is, however, different from a hard fact that is always believed true [Pearl 88]. Taking a default allows us to jump to conclusions based on partial knowledge. These conclusions must however be treated as tentative beliefs that might later be proven untrue. The act of jumping to a conclusion or taking a default is treated by the WISE system as choosing what value to assign a variable. These variables are called **choice variables.**

When a default value is assigned to a choice variable, this binding remains unless a reason is found to believe that this should not be the value. For example, if the system assumes : "?x=3" then the system will believe this unless there is some reason not to. This is the same as Reiter's consistency operator.

The only circumstances under which the system will decide that it is not consistent to believe "?x=3" is if "?x=3" leads to a contradiction. The WISE TMS does not discover contradictions; the onus is on the knowledge engineer to enter contradictory situations in rules such as:

```
If   bird.type = ostrich
and  bird.transport = 'flight
then contradiction.
```

The intuitive idea in this work is that :

- a set of choices (C) induce an extension of some underlying incomplete set of first order Wffs (W).
- any such extension may be viewed as an acceptable set of beliefs that is held about an incompletely specified world.

The WISE TMS works in an extension until such time as it reaches a contradiction. At this point, it returns to the underlying Wffs W and induces a new extension based on some other default or choice. The WISE TMS can keep its beliefs consistent by carefully controlling the way in which

defaults are taken, extensions are induced and contradictions are resolved.

The example.

This chapter will show the workings of the TMS through the use of a scheduling example. The example has been kept small and simple so that the workings of the TMS are not hidden by the complexity of the domain.

The example involves the scheduling of courses. Scheduling a course involves allocating it both a venue and a teacher. Again, in order to keep the example simple it will only cover allocating venues to courses.

[Diagram 9.1] shows the original knowledge base as it is set up at the beginning of the allocation process. The setting up of the knowledge base could have been done by the knowledge engineer or, more likely, it would be done interactively by the user, ie the application would start by asking the user for a description of the courses to schedule, the available venues and the available teachers.

The period over which the courses run is a week. The days will be referred to as day one to day five.

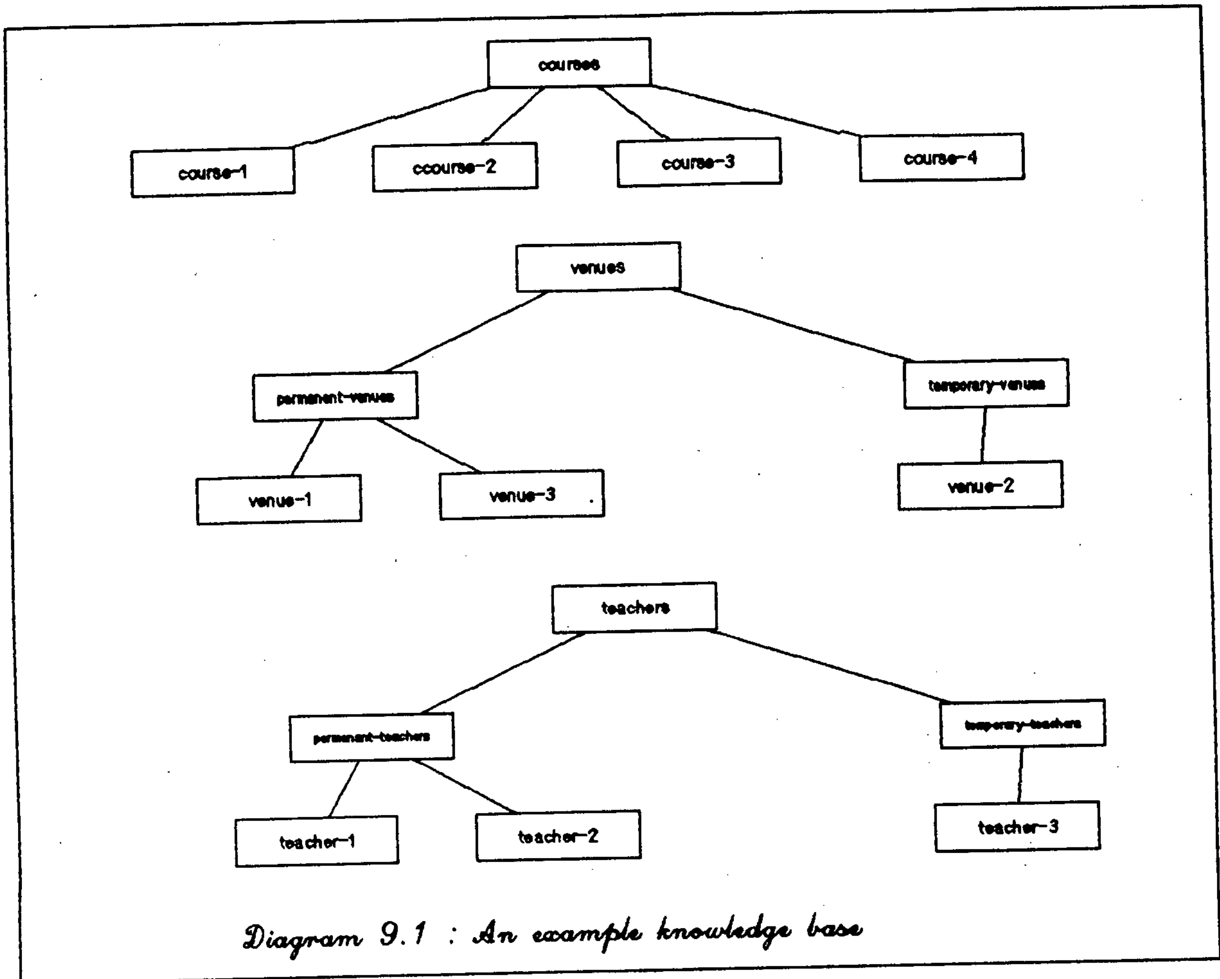


Diagram 9.1 : An example knowledge base

The size of the domain is not important to the workings so the system will be demonstrated on a small example involving 4 courses, 3 venues and 3 teachers. A description of the courses and venues is shown in [diagram 9.2].

List of Courses to schedule

Course name	Subject	size	length	venue	teacher	first day	last day
course-1	Db-3	150	3				
course-2	ES	120	4				
course-3	Db-adv	180	1				
course-4	KE	100	2				

List of venues to be allocated

Venue name	Venue size	Venue Status	courses-allocated
venue-1	50	permanent	
venue-2	150	temporary	
venue-3	200	permanent	

Diagram 9.2 : Venues and Courses.

The solution is found as follows :

:- get the courses to schedule.

:- for each course to schedule do

:- allocate a venue.

:-for each course to schedule do

:- allocate a teacher

:- check the global constraints.

The control structure, which is hierarchical, is shown in [diagram 9.3]. The top level control is done by the agenda and the next level of control is carried out by the schedule-venue and schedule-teacher frames.

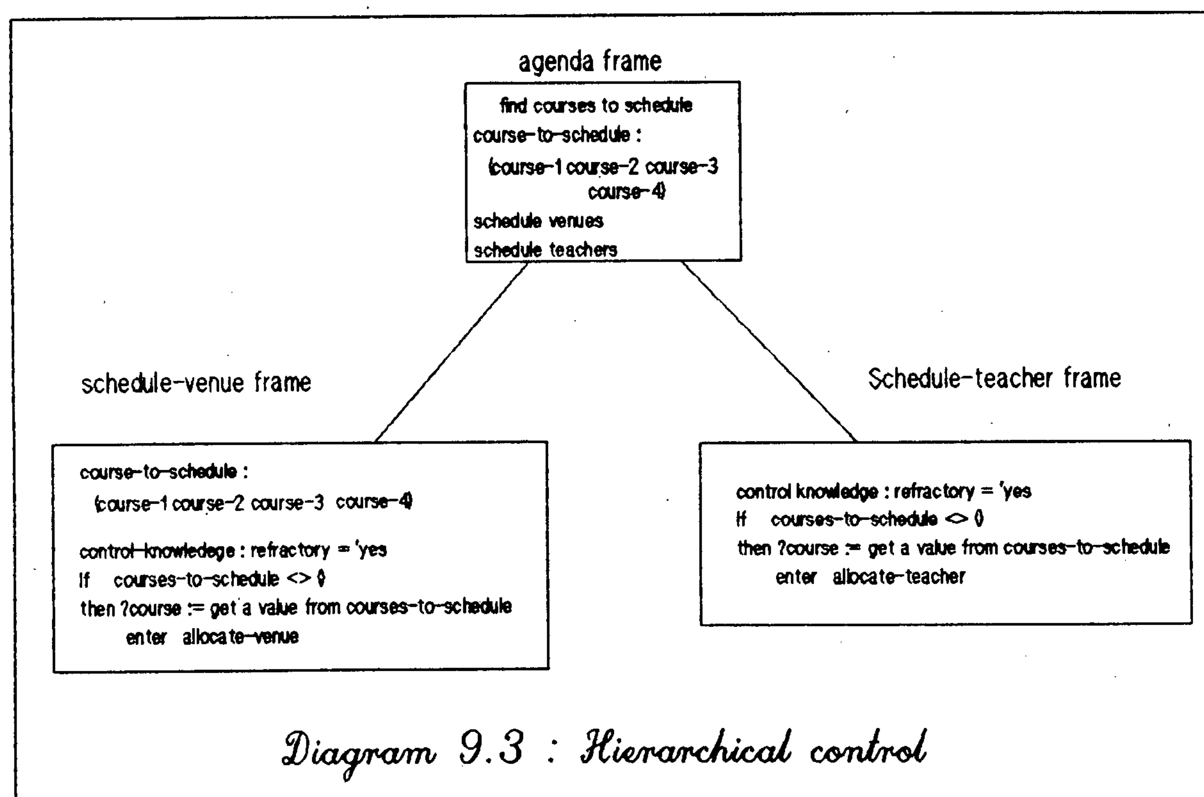


Diagram 9.3 : Hierarchical control

Consider the process of allocating a venue to a course. The allocation of a venue is a defeasible or non-monotonic

allocation as, despite an allocation being correct at the time it was made, it might later cause other courses not to get a venue.

There are some constraints which apply to the allocation of the venues:

- two of the venues (venue1 and venue3) are permanent venues and should always be allocated before a temporary venue
- Since the temporary venues are hired by the week, once a temporary venue has been allocated a single course it may as well be used for the entire week.

The order in which we attempt to allocate venues is governed to a certain degree by heuristics and to a certain degree by the constraints. Ideally the smallest permanent venue available is allocated. Failing that, the system attempts to allocate any other permanent venue and failing that, the smallest temporary venue is tried.

At some point in the allocation of venues process, there will be a need for a revision of beliefs process. This process involves:

- removing a course from an allocated venue
- replacing it with a bigger course in that particular venue
- then finding another venue for the old course (this might also cause a further revision of beliefs).

The revision of beliefs concerning the allocation of venues occurs under the following conditions :

- if a course cannot be allocated a venue because there is no free venue for it and
- there exists a pre-allocated venue big enough to hold this course and
- the course that is currently allocated to this venue is smaller than the course without a venue
- then we must start revision of beliefs.

Once all the courses have been allocated venues, the teachers are assigned. This might lead to belief revision in the venues as certain constraints related to the teachers and venues may be broken.

Note : there are many different ways to solve this problem. A more logical way would be to allocate a teacher and a venue to the course at the same time. However this would make the TMS too complex for an example.

9.3 Choices and assumptions.

In the WISE system, non-monotonic inferences are made using special **choice rules** and **choice frames**. A non-monotonic inference comprises of making a choice about what to believe. This choice could involve what value to assign a

particular attribute or which of a series of objects to consider.

The example starts in the agenda which initially leads the system to enter the schedule-venues frame [diagram 9.3]. This frame has its refractory parameter set to 'yes. This means it will not be exited until there are no true rules left to fire ie even if a rule has been fired, it will be considered again. The schedule-venue frame has a single rule that assigns the ?course (read some course) variable a series of values from the multi valued slot called courses-to-schedule. Initially ?course has a global value of course-1. The rule then causes control to be moved to the allocate-venue frame.

In the example, the "allocate-venue" frame [diagram 9.4] makes a choice about which venue to allocate to a particular course. This is done using the choice rule "assign-venue". A choice rule is a rule that is preceded by one or more choice clauses. The assign-venue rule has a single choice clause, namely :
"choose ?course.venue using c-venue".
Each choice clause introduces a choice-variable, or a frame's slot, which takes the value of the choice. In this case, the choice is about the slot ?course.venue. Initially ?course is equal to course-1 and this rule thus makes an assumption about what venue to allocate to course-1.

```
Rule : assign-venue

choose ?course.venue using c-venue
if   the size of (?course.venue) > ?course.size
and  ((?venue.courses-allocated = ()) or
      (the length of (?venue.courses-allocated) + ?course.length =5))
then ?venue := ?course.venue
      enter the frame put-course-in-venue
      ?venue.course-allocated := ?course
      remove ?course from schedule-venue.courses-to-schedule

RULE : check-assignment

loop ?x-ven on instantiations of venue
if   ?course.venue = ()
and  ?course <> ()
and  ?x-ven.size > ?course.size
and  (?course.size > the size of (?x-ven.course-allocated))
then contradiction
```

Diagram 9.4 : Rules in allocate venue frame

Assigning a value to assign a choice variable, or slot, is a decision that is based on partial knowledge. Obviously if there was no doubt as to which value it should take, there would be no need for assumptions or choices. In such a world, all the knowledge required to make a decision is known and no assumptions are needed. In this example however, a decision regarding the venue allocation for a particular course can only be adjudged correct once the other courses have all been allocated venues and teachers.

Much of the skill of an expert, who solves problems based on partial knowledge, is manifested by making intelligent guesses or assumptions. In the WISE system this amounts to knowing what value to give a choice variable or slot. Because this decision (regarding what guess to make) is vital to the efficient solution of the problem, it must be

knowledge based. Making a guess, or giving a choice variable a value, is a problem that requires the same expert system techniques as any other problem that expert systems solve. The idea is to apply expert system techniques to solve the problem of which non-monotonic assumption to make at a particular point.

Knowledge pertaining to which value to assign a choice-variable, is captured in a special type of frame called a choice-frame. This is associated with a choice variable.

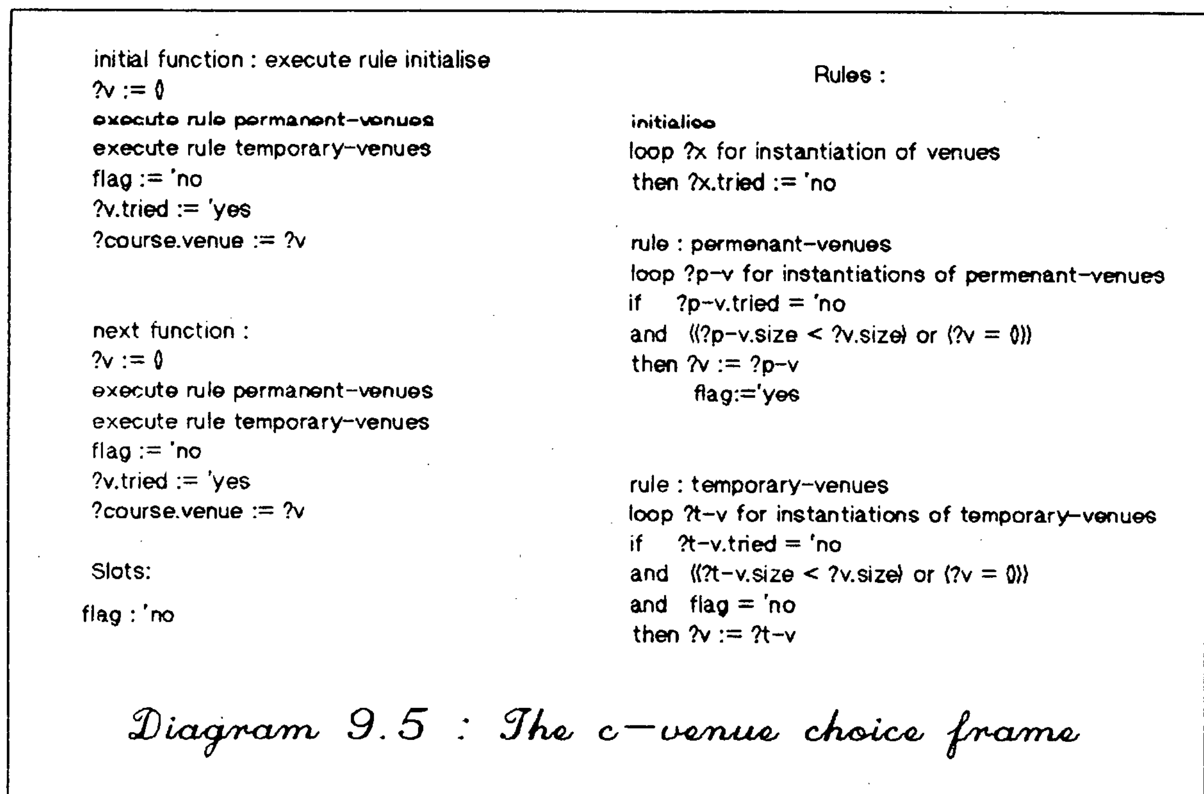


Diagram 9.5 : The c-venue choice frame

[Diagram 9.5] shows the choice frame c-venue. This frame has knowledge about which value to assign to the course-

1.venue slot in rule assign-venue. A choice-frame has several fields:

- Initial function : is a function that returns the most likely value for the choice variable that is associated with this frame.

- Next function : is a function that returns a series of possible values one after the other as each previous value is proven incorrect.

The initial and next functions make explicit bindings to the slot or variable about which they are reasoning. The assumption making problem is kept consistent with all the other problems that the system solves. This is achieved by using choice frames to hold this knowledge.

An expert solves the problem posed in the example by using heuristics which enable him to easily schedule the courses. These heuristics, or rules of thumb, govern the order in which he will consider venues that are suitable for a course.

In the c-venue frame [diagram 9.5] the heuristic used allocates a permanent venue before a temporary one. It also allocates the smallest venue that the course will fit into thus leaving the bigger venues available for larger courses. It is important to note that at the time of the initial allocation of a venue to course-1, the size of the remainder

of the courses is unknown, to the system. Since the sizes of these courses is unknown the system has no way of knowing what effect they might have on the use of venues. As a result it has to make a tentative decision which could later change.

The "c-venue" frame's "initial function" and "next function" [diagram 9.5] are very similar. They do differ though in that that the initial function starts by setting a slot called "tried" to the value 'no, in each of the venues. After that the permanent venues are looped through and the smallest permanent venue that has not been returned before, is returned (venue-1).

The knowledge engineer has a choice about where to place knowledge concerning the allocation of a venue. The if clauses in the rule assign-venue [diagram 9.4] all have to be true for the choice to be valid. They thus play a role in deciding what choice will be made. These conditions could in fact be placed in the choice frame and used to govern what values are considered.

In this example, the choice frame returns the venues in order venue-1 (smallest permanent venue), venue-3 (the next permanent venue) and then venue-2 (the temporary venue).

When the inference engine executes the rule assign-venue, it first applies the knowledge in the choice frame to find a value for ?course.venue. The first value that c-venue

returns is decided by the initial function. The system then checks to see if the rule is valid (antecedents true) with this value. If it is the rule is fired, if it isn't another value is tried. This new value is decided by next-function. The antecedents are checked again using this new value. The process of getting possible values continues until a value is found for which the rule is valid. When the "next-function" has no more values to return, it returns null and the rule fails.

The knowledge in the if clauses could be placed in the choice frame as it, too, pertains to the validity of the choice. This is a matter of style and the knowledge engineer has flexibility about where to place this knowledge. Generally the heuristics that govern the choice are placed in the choice frame while the constraints on the value that the choice can take, are placed in the if clauses of the rule. This allows the same heuristics to be used by more than one choice rule.

In this example, the if clauses of the rule assign-venue [diagram 9.4] check that the two constraints on the allocation of venues have been met, namely :

1 - that the venue's size is equal to, or bigger than, the size of the course.

2 - that the venue that has been allocated does not have another course allocated to it. Should this is be the case, this previously allocated course, together with the

newly allocated course, must completely fill the venue ie run for exactly five days.

In the example: when the rule "assign-venue" is executed, the choice frame is entered and the initial function is executed. This function returns the smallest permanent venue, namely "venue-1" [diagram 9.2]. In this instance, the first if clause of the rule is not true for this value as the venue is too small. The choice frame is thus re-entered and the "next-function" [diagram 9.5] is executed. This causes "venue-3" to be returned.

The first if clause in rule assign-venue [diagram 9.4] is true for "?course.venue := venue-3" and "?course = course-1" since venue-3's size is 200, which is larger than the size of course-1. The second if clause is true because venue-3's course-allocated slot is null. The rule is thus valid and is executed.

When the rule is executed, the slot "venue" in course-1 is assigned the value "venue-3". This is done by the choice-clause. The then part of the rule is executed next. The first then clause of this rule assigns a global variable ?venue to the value ?course.venue. This is done purely for convenience as this value will be referred to on more than one occasion.. ?venue is thus set to the value "venue-3".

The second then clause causes the system to enter a frame called put-course-in-venue [diagram 9.6]. This frame sets

the days during which the course will use the venue. At this stage, only "rule-1" in put-courses-in-venue is true. The if-clause in "rule-2" is false as venue3.courses-allocated is null. Rule-1 is executed; and "course-1.first-day is set to 1" and "course-1.end-day is set to 3" (the length of the course-1). The frame "put-course-in-venue" has no more valid rules and is thus exited. Control returns to the then part of rule assign-venue [diagram 9.4].

```
Rule-1
If    ?venue.courses-allocated = ()
then ?course.first-day = 1
      ?course.end-day   = ?course.length


Rule-2
If    ((length of (?venue.course-allocated)) + ?course.length) = 5
then first-day of ?course := ((end-day of (venue.course-allocated)) + 1)
      end-day of ?course:= ?course.first-day + length
```

Diagram 9.6 : Put-courses-in-venue frame

The third clause of rule assign-venue in frame allocate-venue [diagram 9.4] is executed next. This sets the slot venue-1.course-allocated to the value "course-1". The last then clause of the rule assign-venue causes the value "course-1" to be removed from the multi valued slot courses-to-allocate which is in the frame schedule-courses.

Once the assign-venue rule has finished executing the course, course-1 has been allocated a tentative venue, venue-3. The second rule in frame "allocate-venue" is

considered. This rule, "check-assignment" of frame allocate-venue [diagram 9.5], is not valid as ?course.venue is not null. There are no more true rules in the frame and it is thus exited. Control is now passed back to the rule that called this frame namely the rule, rule-1, in the frame schedule-venues [diagram 9.3]. This frame's refractory flag is 'yes and the consultation thus continues by executing the rule again [diagram 9.3]. This rule sets the variable ?course to course-2 and re-enters the allocate-venue frame.

9.4 System defined choices.

A common type of assumption made, is taking a default, or most likely value. In the WISE system, defaults play two roles:

1 they may merely be values to be taken when all else fails.

2 they may be treated as a list of ordered possibilities of values.

The knowledge engineer can input a default, or an ordered list of defaults, for a slot. If a default slot is referenced in a choice clause as if it were a choice-frame, the default facet's values are viewed as an ordered set of values. The system will then return the first value as the initial value. If this fails, the system returns the rest

of the values, one by one, in the order in which they were entered by the knowledge engineer.

For this example, instead of having a choice-frame c-venue, it could be a default slot as shown in [diagram 9.7]. This is possible only if there is a prior knowledge of the order of possible values. There are however, many cases where this order is dependent on some value that is set at runtime; in this case a choice frame should be used.

```
Slot : c-venue
        Default : (venue-1 venue-3 venue-2)
```

Diagram 9.7 : Choice using a default.

This default mechanism allows an unskilled knowledge engineer, with no knowledge of the WISE TMS or choice mechanism, to build applications which use non-monotonic inferences.

9.5 The Justification network.

The inference engine executes rules. Valid rules are passed to the TMS which places them in a justification network. This network keeps the data dependencies, and the contexts,

in which facts are believed. The network is not involved in deciding which clauses to execute or which clauses are true. It is told only true facts by the inference engine ie only valid rules are passed to the TMS. Clauses that have variables are first instantiated and then passed to the TMS.

datum : The external representation of the data
support : a reason for believing the datum
undo function : a function that will reverse the datum's effect
value : value of the lhs of a set command
consequences : the nodes that are effected by this datum
label : the belief status of the node
context : context in which this node is believed
connections : links to the TMS-pointer-list
type : type of node (choice , user-interaction or normal)

Diagram 9.8 : A TMS node

A WISE node is depicted in [diagram 9.8]. It has several fields :-

- The datum field is used to store the instantiated clause. The form of this datum is of no relevance to the TMS. It is, in fact, a link to the clause it represents in the inference engine. It is used to generate explanations and to tell the inference engine what action to perform when this node moves from OUT to IN.

- The support for a node is the reason for believing the fact represented by the datum. It is in the form of other

nodes ie the reason we believe a fact is based on our belief in other facts. It imposes a constraint on the truth value that this node can take.

- The undo-function is used to undo the effects of bad choices, which are choices that lead to a contradiction.

- The value field of a node is used only in nodes representing a set action. It stores the RIGHT HAND SIDE or value that the item is being set to. This is used by a data structure called the TMS-pointer-list (discussed below) to merge equal entries in the TMS.
eg. x:=y-5 with y=7 will have a value field equal to 2.

- The consequences field of the node has the address of all the nodes whose truth is dependent on this node.

- The label field is used to tell if the fact is believed (IN) or not believed (OUT).

- The context field is the context (defined below) in which this node is believed.

- The type can take the values "choice", "normal" or "user-interaction", which represents a fact that was inputted by the user.

Facts are inferred and passed on to the TMS which inserts them into the network. [Diagram 9.9] provides an example of

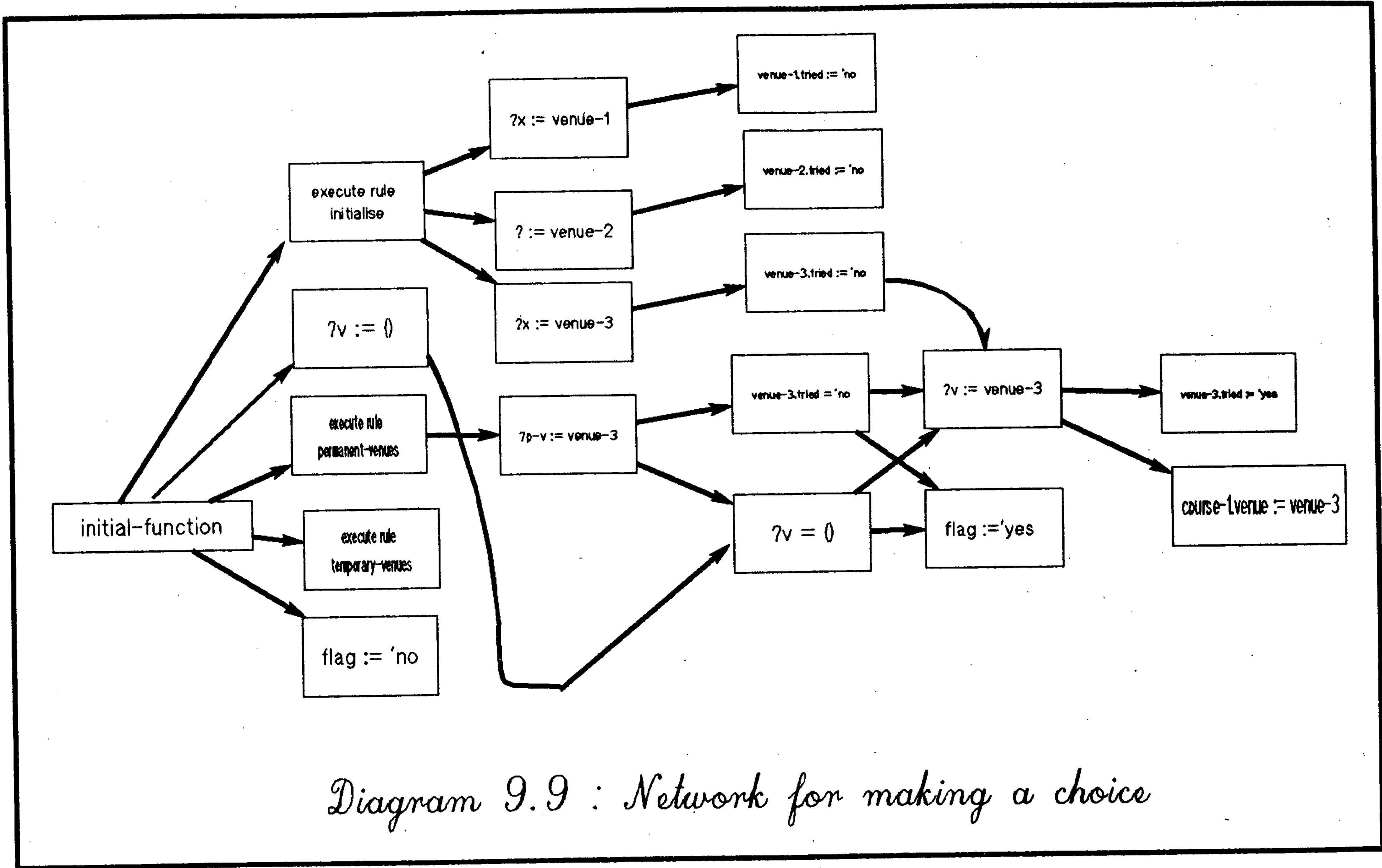


Diagram 9.9 : Network for making a choice

what a network looks like. This diagram illustrates the network that represents the inferences done in the choice frame "c-venue". This chapter will be showing a couple of network diagrams. The relevant diagrams are to be interpreted as follows:

- a node is represented by a box with its datum in it.

- support for a node is represented by the incoming arrows ie if *all the nodes* that point to a node are IN, then the node is IN. The support for a node is held internally as a constraint clause. It constrains the belief values of the nodes that it connects.

- labels are all IN unless they are explicitly labelled OUT.

- the consequences of a node are represented by the outgoing arrows. These arrows point to the nodes whose belief labels' value is dependent on this node.

Not shown on the network diagrams are :

- the undo functions.

- IN labels

- Type, except that a node of type choice will have a "*" above it.

[Diagram 9.9] illustrates the network for the rules in c-venue. On the left hand side is a node called initial function. When the system enters c-venue the initial function is executed, and this fact is passed to the TMS. The initial function calls rule "initialize" in c-venue [diagram 9.5]. This rule loops through the venues setting

their "tried" slot to 'no. It then calls the rule permanent-venue. This rule initially returns "venue-1" but this causes the if-clauses in the choice-rule to fail (this is not shown in [diagram 9.9]). Permanent-venue then returns "venue-3". This value is acceptable and "venue-3" is thus bound to the loop-variable ?p-v. This is all put in the network.

A rule is added to the network once the inference engine determines that its antecedents are true. The process of putting this rule in the network involves establishing the reasons for believing the antecedents and using these reasons to support the new antecedent nodes. The nodes must be "spliced" into the correct position in the network. In order to determine which position is the correct one the system must find all the nodes whose beliefs constrain the belief status of the node. These nodes are then used as the antecedents of a justification that supports this node. The algorithm for supporting an antecedent node is :

1) For each variable in the antecedent do: the node where the variable was last set to a value, is added to the support.

2) For each knowledge base item in the antecedent do: the node where the item was last bound, is added to the support.

3) If there is a relation in the antecedent, then the place where this relation was set is added to the support.

4) If it is the antecedent of a choice rule, it is supported by the node representing the "choice-clause".

This algorithm is intuitively correct.
For example ?v=(), in the rule permanent-venues is true because ?v was set to () and is thus supported by the node that represents this.

Each consequent of the rule is now supported separately. Algorithm for supporting a consequent is :

1) If it is an input-function, contradiction, control-function, reset-function or any other function without items or variables on the RIGHT HAND SIDE, then support with the antecedents' nodes.

2) If it has a RIGHT HAND SIDE, then find the place where items on the RHS got their values and use those, nodes plus the antecedents' nodes to support the node.

Using these two simple algorithms, the nodes are placed in the network and correctly linked to their support and consequences.

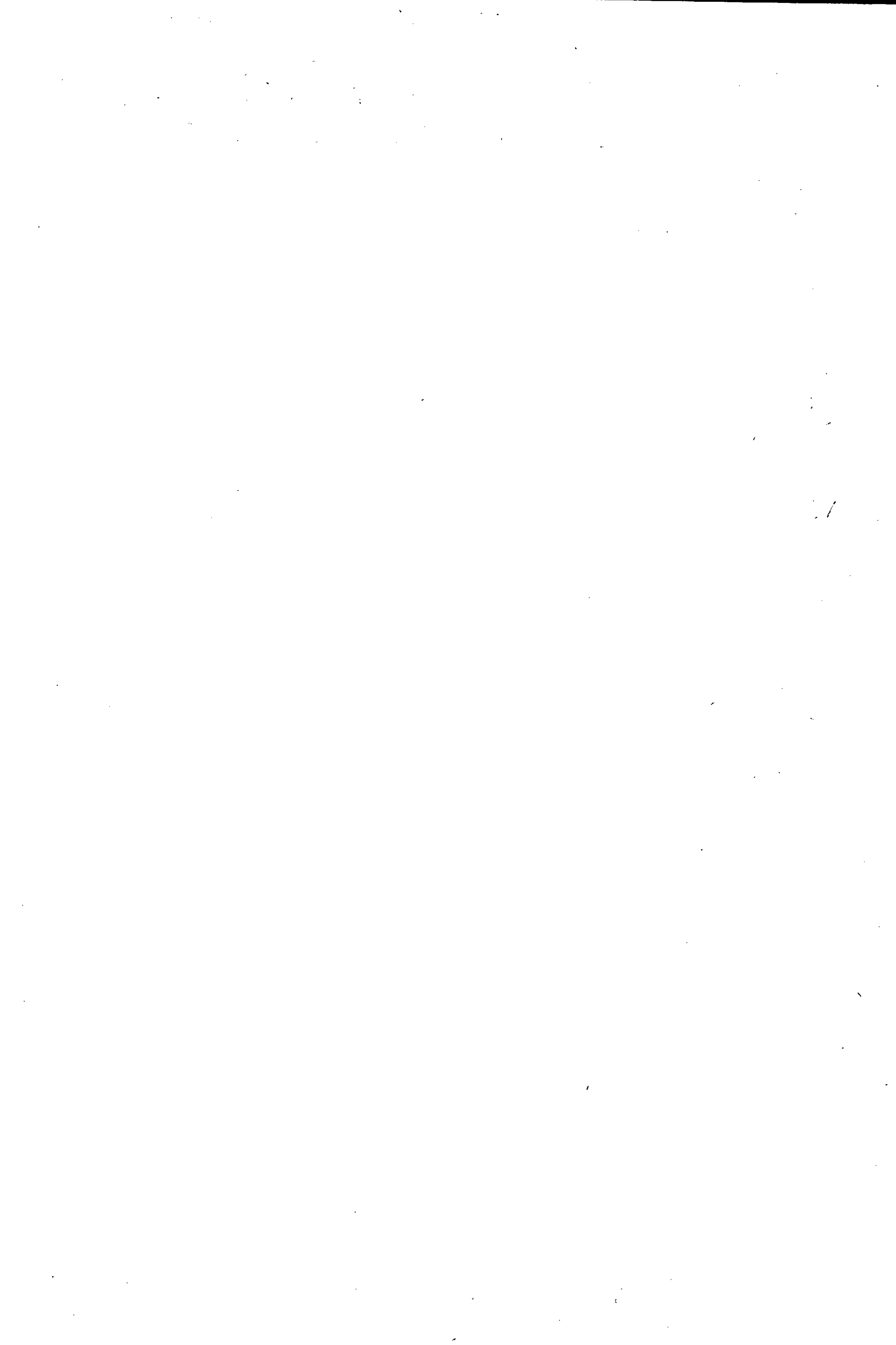
Should there be more than one reason for believing a fact, it will have more than one node in the justification

network. However, all these separate nodes are linked via a special data structure called the TMS-pointer-list which will be discussed in more detail in chapter 11. A fact is believed if there is one or more valid reasons for believing it.

Choices are facts whose belief is based in our disbelief of other facts; this makes them the same as the assumptions defined by [Doyle 79]. These other facts are not represented explicitly; they are all the other values that a choice-variable may have. For instance when "venue-3" was allocated to course-1, the system implicitly believes that course-1's venue is not venue-2 or venue-1. For this reason the system does not allow a slot or variable that is bound in a choice-clause, to have its value changed. The only time it changes the value of a choice-variable or slot, is if there is a contradiction.

A choice is represented in the network by a special node called a choice point. These nodes define beliefs that are assumptions and might later be proven wrong. They are thus the points in the network that the dependency directed backtracker will go to in an attempt to resolve a contradiction (discussed below).

To return to the example: previously discussed was the allocation of "venue-3" to course-1.venue. The network for this is shown in [diagram 9.10]. In this diagram, the



network shown in [diagram 9.9] is encapsulated in the octagon labelled "choice". The slot courses-to-schedule is written as c-t-s. The network starts from when the schedule-venue frame is entered, and ends when we return there after the allocate-venues and put-courses-in-venue frames have been entered. The choice node, labelled with a "*", represents a defeasible fact or assumption. This is shown by its support from "choice" which could return other values.

9.6 Contexts as possible worlds.

In chapter3 reasoning about an action was defined as: the formation of a new world, or situation, that is the same as the world before the action but for the results of this action. As actions are executed, a new situation or world is formed.

Problem solving using assumptions or choices leads to the formation of many possible worlds. These are fictional worlds that are used to reason about conceived states. The differences between these worlds and the actual world, arise because of the choices one faces. Every time we make a choice we could possibly have made a different choice. If we had made the other choice the world would be different ie we would be reasoning in another possible world.

For instance on seeing a bug in our path, we are faced with a choice: whether to squash the bug or leave it alone.

For all practical purposes whichever choice one makes the two possible worlds are the same (not from the bug's point of view, but from ours). On the other hand, a choice such as Khruschev's choice whether to turn his fleet back from Cuba or not, had major consequences. In the world where he decided not to do this the bug wouldn't have to worry about being stood on.

All this is of interest to the expert system developer because he can use such a view to solve problems. The initial knowledge, axioms or premises are viewed as the starting point. As the system makes choices, it builds a possible world. When the system stops, it checks if this world contains the solution it requires, if so it is a **solution world.** If the possible world it is reasoning in reaches a contradiction, or fails to find a solution, then it is a **contradictory world** and it must be changed.

Possible worlds are defined and described by the assumptions, or defaults, that are true within them. For example, in one possible world venue-3 is the venue of course-1, and another possible world could be the one where venue-2 is the venue of course-1. A possible world is like a default extension. It defines a context in which the

beliefs in the world hold. Every belief has a context which is a set of assumptions on which it is based.

To return to the example, after the allocation of a venue to course-1, control passes back to the schedule-venues frame. The refractory parameter of this frame is set to 'yes which causes the rule in the schedule-venue frame [diagram 9.3] to be fired again. This rule takes the next element of the courses-to-schedule slot, namely "course-2", and sets ?course to it.

Now the allocation of a venue to course-2 can begin. This starts the whole process again except, in this case, venue-3 cannot be allocated to course-2 as the second if-clause in rule assign-value [diagram 9.4] will not be true (remember venue-3.course-allocated is equal to course-1, and its length + the length of course-2 is not 5 [diagram 9.2]). Venue-1 can also not be allocated as it is too small, thus the system is forced to allocate a temporary venue (venue-2) to course-2. The allocation of a venue to course-2 is happening in a certain context, namely, the context defined by ("course-1.venue=venue-3"). This context affects the allocation of the venue to course-2.

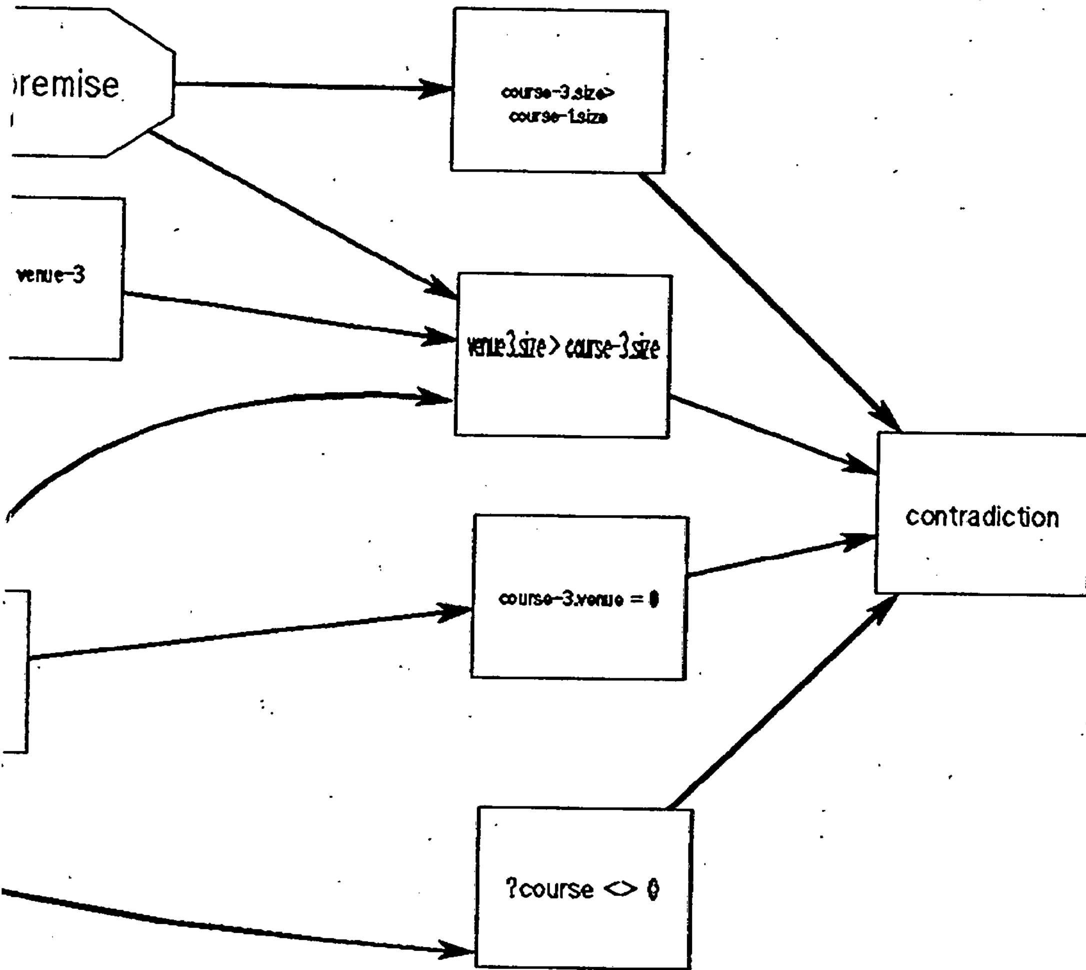
Next, course-3 is allocated a venue. This takes place in the context of ("course-1.venue=venue-3" and "course-2.venue= venue-2"). In this context, the allocation of a venue to course-3 fails. This happens because the only venue large enough to hold this course has already been allocated to

course-1, and the length of course-1 + length of course-3 is not five days. The choice frame returns each venue and each venue breaks one of the constraints that are encapsulated in the if-clauses of the rule "assign-venue".

After the "assign-venue" rule fails, the course-3.venue slot is null. The rule "check-assignment" [diagram 9.4] is executed and succeeds with the variable ?x-ven being equal to venue-3, ie "venue-3.size>course-3.size" is true and course-3.size is greater than the course currently allocated to venue-3 (course-1). [Diagram 9.11] shows the justification network after rule "check-assignment" has been executed. This diagram thus shows the allocation of venues to course-1 and course-2 and the failure to allocate a venue to course-3. This failure leads to a contradiction which the system must resolve.

The context in which a node is believed is automatically updated as a node is put in the network. The algorithm for this is :
If the support nodes are not choice nodes then the context of the antecedent nodes is the union of their contexts. If the supporting node is a choice node then this choice node is added to the context.

When a node representing a contradiction is added to the network, its context immediately defines the assumptions that could be the reason for the contradiction. The


premise
venue-3
course-3.size>
course-1.size
venue3.size > course-3.size
course-3.venue = ∅
?course <> ∅
contradiction

contradiction is now resolved by retracting one of these assumptions and making a new one. A context in which a contradiction occurred is called a **nogood.** This is used to constrain future choices in that a choice must never lead to a nogood context. In this way contradictions are never rediscovered.

Problem solving is now reduced to: build a possible world, assess if it is the solution world and if not, move on to a new world. The operation "move to a new world" is difficult as the system should not have to recalculate the entire new world. A context switch, or possible world change, must be done with the minimum of work. The only beliefs that must change are those directly dependant on the choice which is being retracted. Similarly, the only beliefs that must be added are those dependent on the choice being reinstated. Such a move is a variation of the frame problem.

9.7 Resolving contradictions.

This chapter has outlined three important concepts :

- how non-monotonic assumptions are made
- how the reasons for facts are kept
- how facts are believed in certain contexts.

All this information is kept in the justification network and is used to resolve contradictions that might arise due to incorrect assumptions or guesses.

Contradictions arise when the knowledge engineer explicitly tells the system that a certain condition is contradictory. The WISE language allows the knowledge engineer to enter special rules called **contradiction rules.** A contradiction rule has a single then-clause, "contradiction". The rule check-assignment [diagram 9.4] is a contradiction rule. This rule states that if certain conditions are true then a contradiction exists, ie one of the assumptions that supports the conditions is incorrect.

[Diagram 9.10] showed the network with the contradiction node's label set to IN. This is a contradictory world. In order to resolve a contradiction, the system must establish in which context this contradiction is believed and then change to a non-contradictory one. [Diagram 9.10] shows that the contradiction is believed in the context defined by the assumptions (course1.venue:=venue-3 and course-2.venue:=venue-2) .

The way in which the system goes about resolving a contradiction, is to choose a **culprit** as the guilty assumption and this culprit's choice node is then accessed via the TMS-pointer-list (the details of how the TMS-pointer-list works are in chapter11). The dependency

directed backtracker does not have to do a search back as the context tells it where to go.

A culprit can be chosen in two ways :

- by the system, in this case, the knowledge engineer does not specify any preference and the system arbitrarily chooses a culprit. This is often used in cases where it is impossible to tell which assumption led to the contradiction.

- by the knowledge engineer using a **contradiction frame**. A contradiction frame is a special frame that allows the knowledge engineer to enter a choice, or ordered list of choices, that he feels are more likely to have caused this contradiction. The knowledge engineer can reason about which choice has caused the contradiction. In [diagram 9.12] the knowledge engineer is reasoning using the ?x-ven variable that was set in the contradiction rule.

```
Bad choice list :
        venue of (?x-ven.course-allocated) := ?xven
```

Diagram 9.12 : Contradiction frame for check-assignment

The contradiction frame for the rule check-assignment is shown in [diagram 9.12]. This frame says that the assumption most likely to have caused this contradiction is the one which assigned course-1 the venue venue-2. The

knowledge engineer can reason about this because venue-2 is the only venue big enough for course-3. The system thus chooses this choice as its culprit. Once a culprit is chosen, it must be retracted (OUTed) and all the beliefs that are based on it, must be set to OUT. This is known as **propagate-out**.

For example, when course-1.venue=venue-3 is marked OUT, all its immediate consequences are marked OUT. For each of these consequences all their consequences are marked OUT etc. [Diagram 9.11] shows that propagate-out causes all the nodes to the right of the culprit to be set to OUT. As facts are disbelieved, the effect they had on the knowledge base is undone. The knowledge base is thus in a state as if the bad choice had never been made.

The context in which a contradiction occurs is called a **NOGOOD**. The system will never attempt to reason in that context, or any supercontext of that context, again. The nogoods are stored in a special hierarchical structure that allows them to be easily accessed and updated. They are global in that they can be viewed in any context. The nogoods are used by the choice mechanism to exclude future choices. If making a choice will lead to the formation of a nogood, it is not made.

When making a new choice, there are several reasons that could exclude a value from being allocated to a choice-variable or slot :

- If the choice leads to the antecedent of the rule not being true.

- If the value has been tried in this context before ie it will be creating a nogood context.

Prior to a new choice being made, the system must ensure that the new choice is going to be made in the same context as the old choice. This is necessary as the choice might depend on some value in the knowledge base that is dependent on the old choice ie that exists in a contradictory context. Once propagate-out has been completed the knowledge-base has been restored to the state it was in when the original choice was made. This is not really true in that other actions, unrelated to this choice could have been made, but they are not influential on the decision about what to do next.

A new choice is now made by referring to the choice frame that is attached to this choice. In the example, a new choice about the venue for choice-1 is made. The "next-function" is executed with venue3.tried set to 'yes. The next function therefore returns the temporary venue-2 (the only other venue large enough) as the venue for course-1. The rule assign-venue is executed with the new assumption and the consultation continues.

A major issue in the above scenario is that of control. Firstly control must be returned to the choice node. Once

the new choice has been made, the system must know what to do next, namely, return to the rule that called it (rule-1 in schedule-venues). This is achieved via links to a special control-stack which keeps the control-actions that led to a choice. This will be discussed in more detail in the chapter on implementation details (chapter 11).

To return once more to the example, a new choice concerning the venue of course-1 has been made. Propagate-out has OUTed a choice concerning course-2. This causes the control to pass back to the calling rule, namely the rule in "schedule-venues" which now continues. Bearing in mind that, as the system propagated-OUT, the "course-to-schedule" slot has had its value set back to (course-2 course-3 course-4), the rule in schedule-venue thus sets ?course to course-2 and continues. In this new context (with course-1.venue = venue-2) course-2 is allocated venue-3. In this context ((course-1.venue = venue-2) and (course-2.venue = venue-3)) course-3 is allocated venue-3 and course-4 gets venue-2. All the courses have thus been allocated a venue and the system returns to the agenda. Next the teachers get allocated in a similar manner.

9.8 Inference Engine-TMS interface.

The WISE TMS lacks many of the advantages of the ATMS however, its advantages lie in its easy implementation and

efficiency for the class of problems for which it was designed.

All the inferencing and finding of truth values occurs in the inference engine, and the inference engine has overall control over a consultation. True rules are passed to the TMS as a series of constraints (justifications) on the values of beliefs of facts. The TMS has no knowledge of the contents of a rule, it simply stores the relationships between the clauses of a rule. These relationships are later used to pinpoint culprits in the resolution of contradictions.

As the inference engine performs inferences, so it updates or changes the knowledge base. The knowledge base reflects the currently held beliefs about the knowledge while the TMS reflects the reasons for these beliefs. As a node is put in the network it is linked to the rule clause that it represents. The nodes are also given methods to undo the effect that the rule clause had on the knowledge base.

The WISE TMS works in one context at a time. The knowledge base is updated as it follows the effect of a set of assumptions. Should this assumption set lead to a contradiction and one of these assumptions is retracted, the system then has to:

- propagate-out through the network
- undo the effects caused by the actions represented by the just OUTed nodes.

Undoing the results of a bad choice is an expensive, but necessary, operation. The undoing must be done in the correct order ie. in the opposite order from the way in which it was done. As propagate-OUT runs it makes a list of undo-actions that are then executed in reverse.

Because retraction of an assumption and the making of new ones is so expensive, the WISE system attempts to minimize making wrong assumptions in the first place. To facilitate this, the choice mechanism allows heuristics that guide a choice to be captured. Choosing the wrong culprit when resolving a contradiction is also an expensive process; the WISE system tries to minimize this by capturing heuristics with which to guide this decision in the contradiction frame.

Undoing choices that are not the guilty ones has to be carefully controlled. If the original reason for making the choice is still applicable in the new context, then the choice must be remade after the undoing has been completed. In order to facilitate this a control stack is kept. The control reason for making a choice is kept. If at the end of an undo the choice is OUT but the reason for making the choice is IN, then the choice is remade and the control passed back to the calling control action.

Undoing control and normal set commands involves several actions :

- firstly, the value of the slot, variable or control-field is set back to its old value.

- secondly, the validity-flags of the clauses that are dependent on these values is set back to unknown.

- thirdly, the Tms-pointer-list is set back to its original state.

User inputs are stored for later use so that if the action clause that requests an input is OUTed and then later INned, the user is not pestered with the same question again and again.

9.9 TMS-Pointer-list and circularity

The WISE system keeps a data structure called the TMS-pointer list. This structure (details in chapter 11) keeps track of all the items in the TMS and where they have changed their values. It was stated in the section on finding support for nodes that a node such as x=3 is supported by the place x got its value. To facilitate this a pointer into the TMS, to where each item that has been changed, is kept along with a series of pointers to the nodes in the TMS that made the changes.

In other JTMS type systems a major problem encountered is the circularity of support and multiple justifications for a

belief. In the WISE system, each node has only one justification which is its support. Every reason for believing a fact is kept separately. This implies that there could be multiple physical nodes with the same fact. However if this were the case, logically, many of the advantages of the TMS would be lost. The disparate nodes that represent a fact are thus merged via a common entry in the TMS-pointer-list.

When a new fact such as x:=3 is supported , an entry is made in the TMS-pointer-list under x. This entry points to the node representing the new fact. Should another entry that also sets x to 3 arise, then both these entries are merged and the beliefs are made the same. Thus, if the other x=3 node was OUT, it is set to IN and propagate-IN is started.

Circular support is easily detected by simply checking if the same node supports itself. Since there is no pointing back except via the stack the mechanism to control circular support is simple. The problem of circularity now resolves itself as a node never points back to a previously defined fact. When a new justification for a fact is added, the system can easily check if merged entries are supporting each other.

9.10 Conclusions.

The WISE TMS is a simple and easy to use facility that allows even relatively unskilled knowledge engineers to use non-monoticity. Since it is based on a JTMS, only one context can be worked in at a time and context switching is thus an expensive operation. In order to minimize the need for context switching, the system has facilities for capturing heuristics that will allow the correct guess to be made the first time round.

CHAPTER10 : COMPONENTS OF THE WISE SHELL.

10.1 Introduction.

In addition to the inference engine and TMS modules which have already been discussed, the shell comprises of several other modules, namely : a knowledge engineering environment, a compiler and a user interface [diagram 10.1].

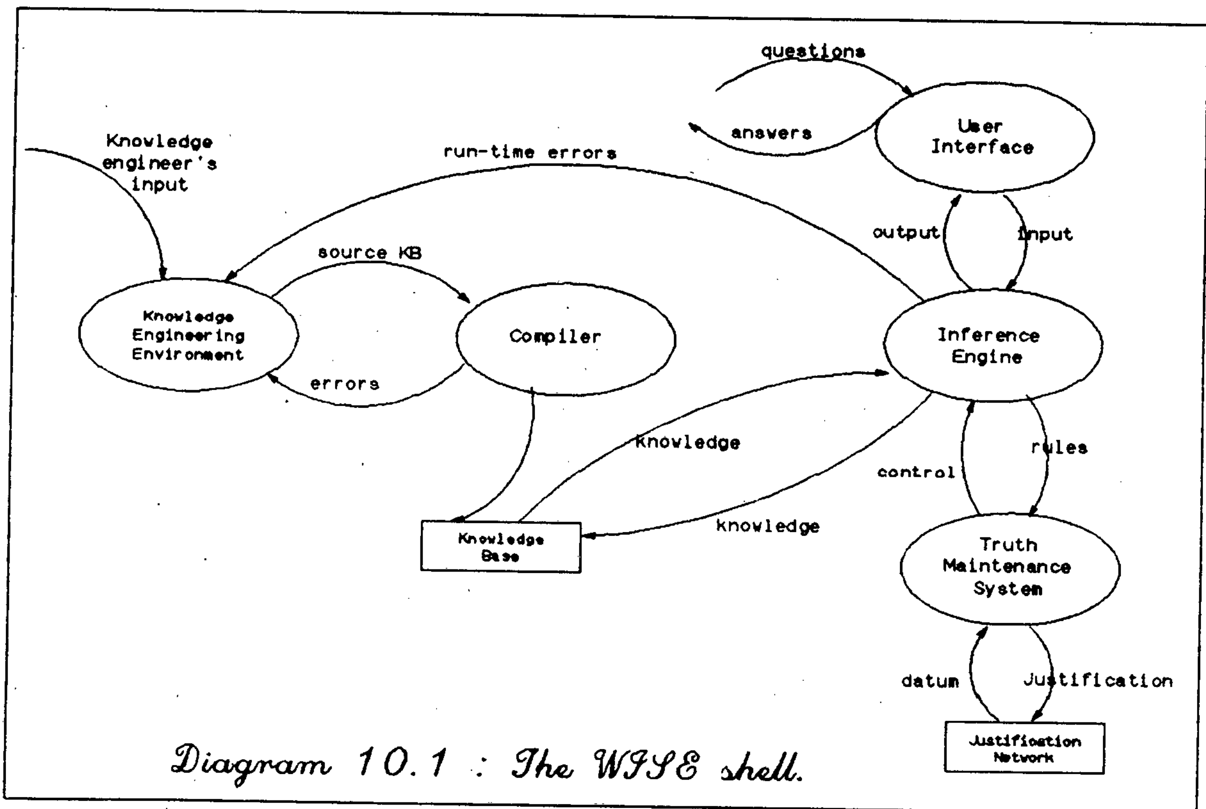


Diagram 10.1 : The WISE shell.

These modules do not fall within the scope of this research but a short description of their features is included in this chapter. This is done in order to show how the system fits together and how the representation, inference and control mechanisms are made accessible to the knowledge

engineer and final user. These modules will be only very briefly looked at. The workings will not be explained in any detail, nor will the reasons for certain decisions be discussed, as this is beyond the scope of this thesis.

10.2 The Knowledge Engineering Environment.

10.2.1 Knowledge Entry.

The knowledge engineer environment supplies the knowledge engineer with powerful tools to develop new knowledge bases or edit existing ones. The tools include an extensive graphics interface which is designed to facilitate the easy entry, and editing, of knowledge. The design of the graphics interface is very similar to the layered design of the knowledge base. Just as the knowledge base can be viewed as a taxonomy, so can the knowledge engineering environment [diagram 10.2].

During an editing session the knowledge engineer is always in a certain context . A context may be the knowledge-base, frame, slot or rule contexts. Each context is reflected by a screen. If the knowledge engineer is in the context of frames , the screen will be the frame screen and any action he performs will be on frames. Enhancing (highlighting) is

used to select the particular object upon which an action is to be performed.

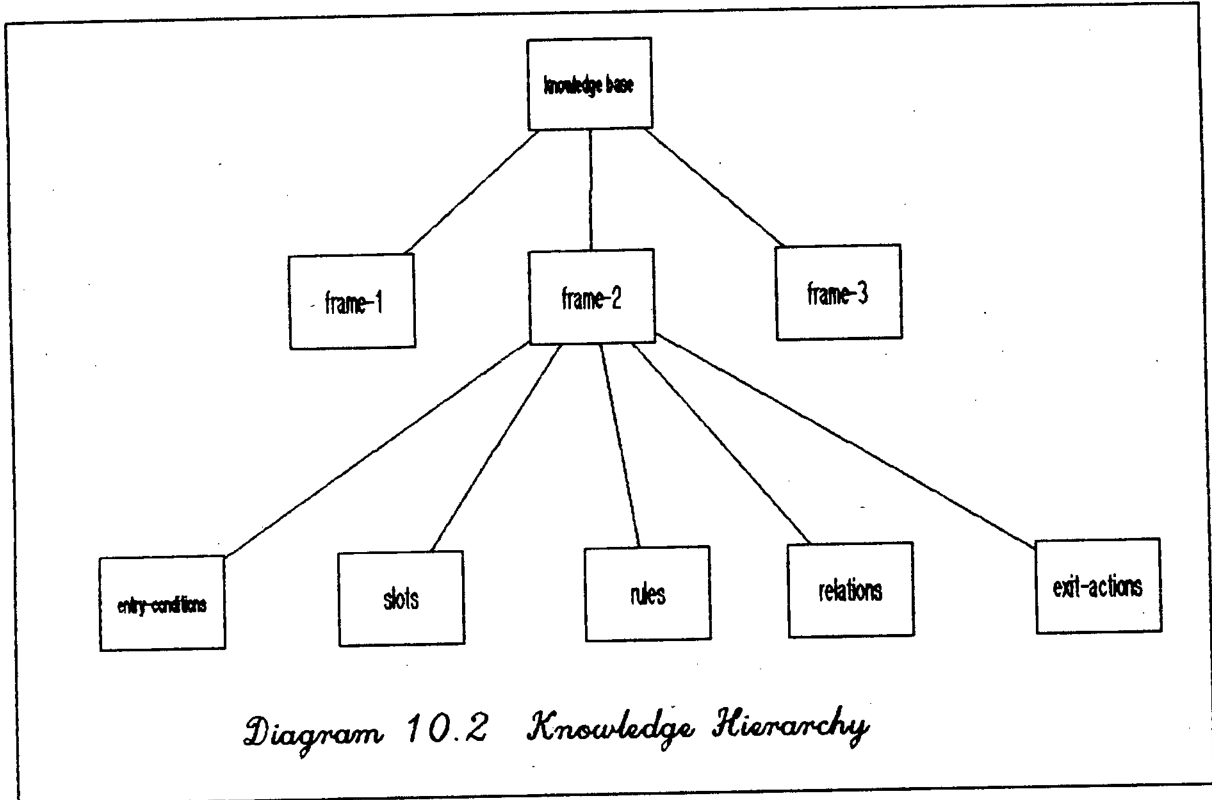


Diagram 10.2 Knowledge Hierarchy

A knowledge engineering session starts at the knowledge base level. Any command executed is performed on a knowledge base. [diagram 10.3]. The commands are: select, delete, add, print, consult and exit. The command is chosen using the softkeys, F1-F8.

To add a new knowledge base, the "add" option is chosen. The knowledge engineer is then prompted for a name. To edit an existing knowledge base, the knowledge base required is selected. Once a knowledge base has been selected, a new screen is drawn and the knowledge engineering environment is now at the frame level.

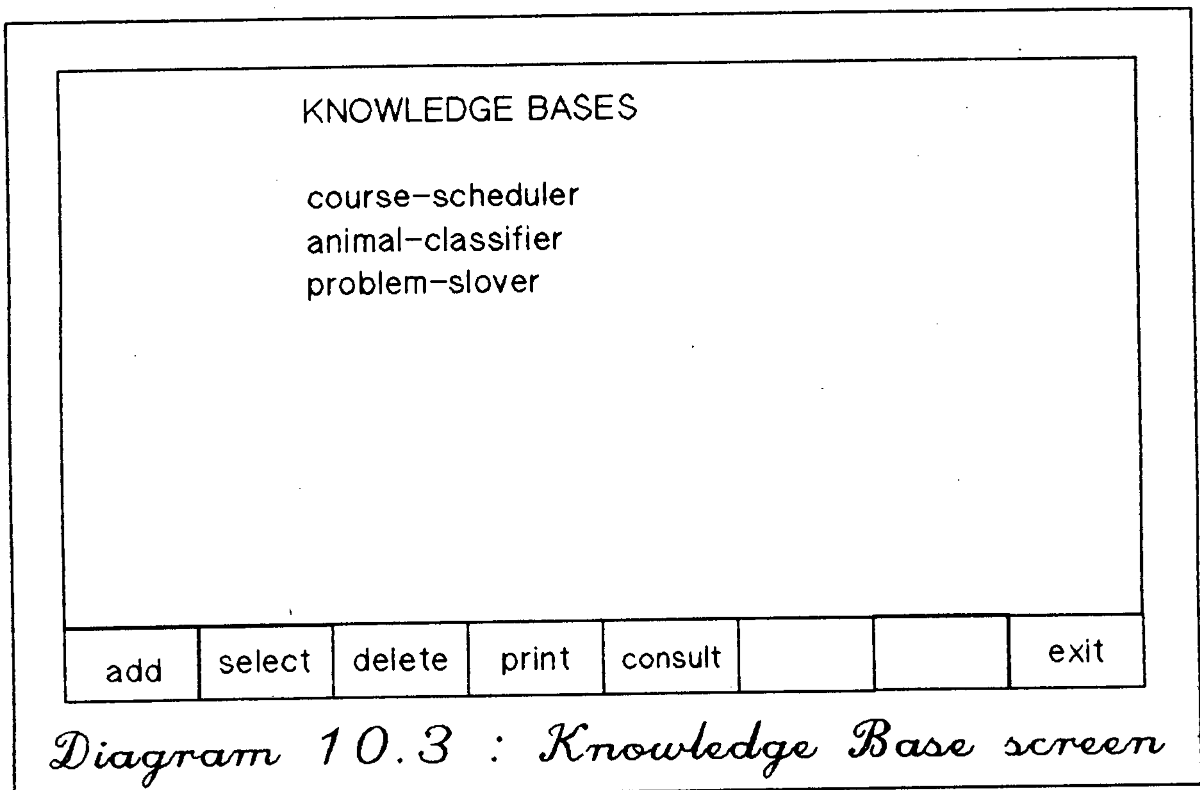


Diagram 10.3 : Knowledge Base screen

Once at the frame level, the knowledge engineer can select, add, or delete a frame as well as exit to the knowledge base level. The frame screen [diagram 10.4] shows the frame taxonomy and allows the knowledge engineer to interactively change the structure of the knowledge base. He can browse through the frames with ease and enlarge any one that he wishes to edit.

To select a frame for editing, the knowledge engineer uses the mouse, or arrow keys, to highlight the frame he wishes to select. He then presses the "select" soft key and the knowledge engineering environment moves to the frame components' screen.

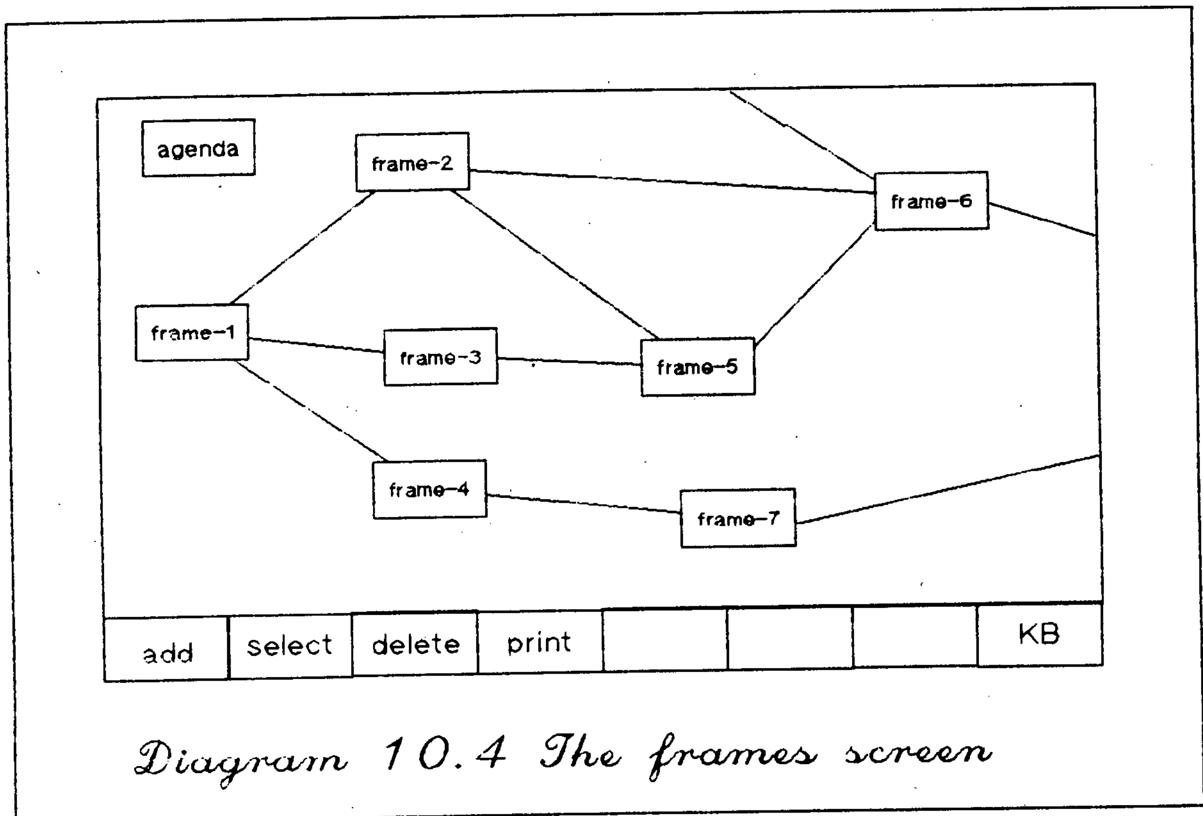


Diagram 10.4 The frames screen

To add a frame, the add key is selected and once the new frame's relations have been entered, it is placed in the correct place in the taxonomy. The frame taxonomy scrolls up, down, left and right.

The frame components' screen [diagram 10.5] shows the different fields that comprise a frame. The knowledge engineer selects which component type he wishes to edit and then selects a specific element in that type, using the arrow keys and select button. Once a specific component has been selected, the knowledge engineering environment moves into that component's screen.

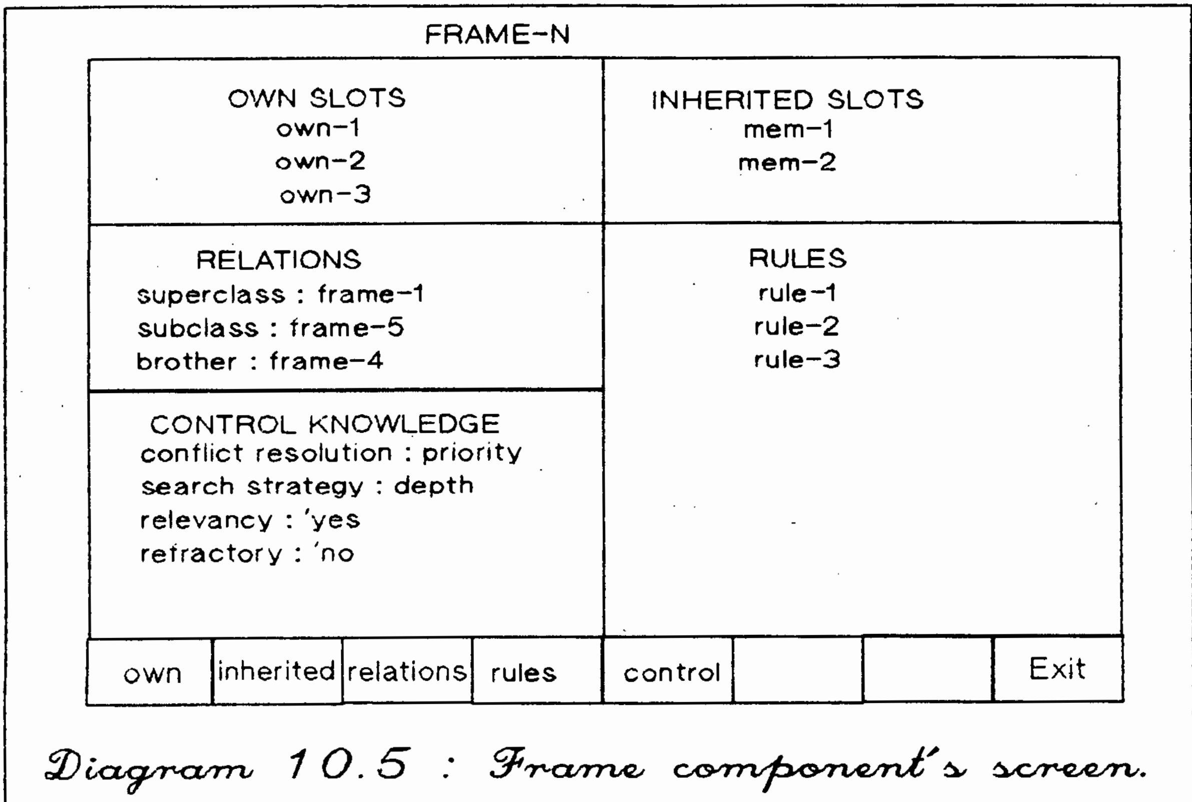


Diagram 10.5 : Frame component's screen.

[Diagram 10.6] shows the own slot screen, [diagram 10.7] shows a rule screen. There are also screens for control knowledge and relations.

The knowledge engineering environment supplies an editor which can be used to modify a rule or procedural attachment, by either changing existing clauses or adding new ones.

The screens were especially designed for ease of use. All system messages and editing are done in a specially defined message region which is at the bottom of the screen. The system attempts to be user friendly but not at the expense of power.

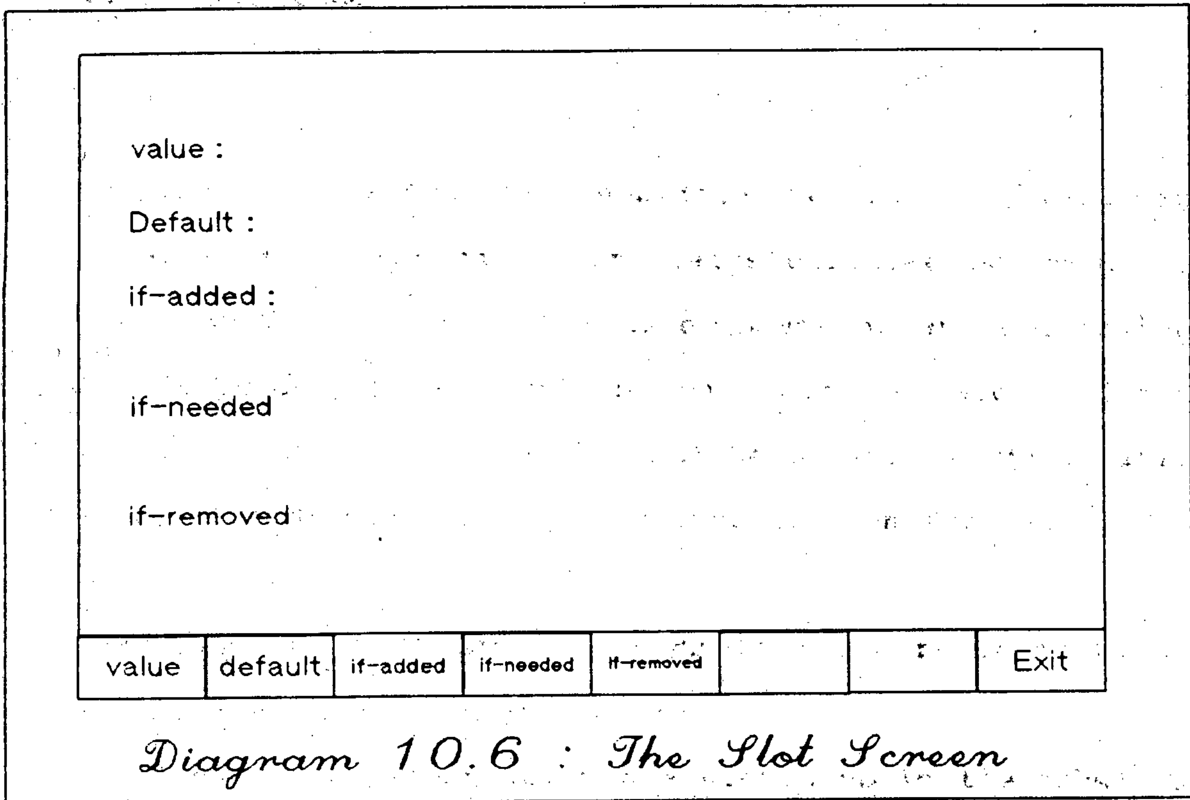


Diagram 10.6 : The Slot Screen

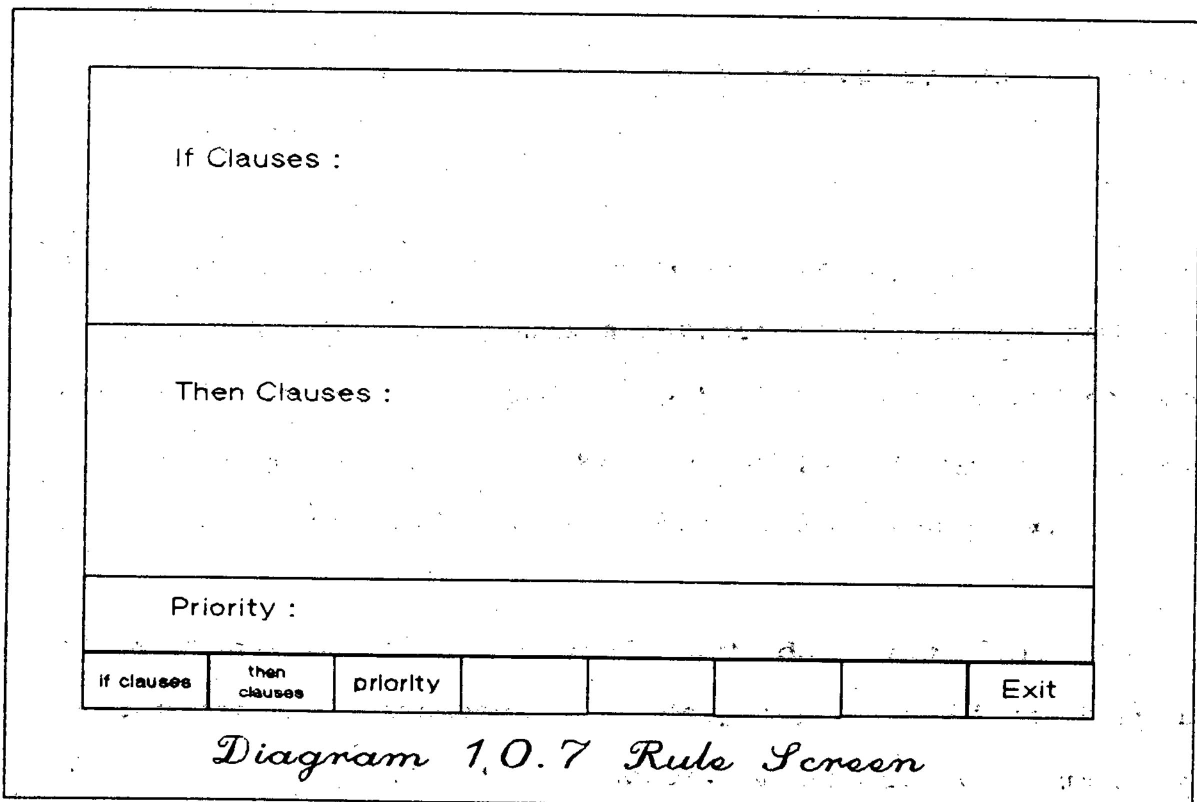


Diagram 10.7 Rule Screen

10.2.2 Debugging.

An important aspect of the knowledge engineering environment is the ease with which the knowledge engineer can debug his knowledge base. During the development of an application, the knowledge engineer will be entering knowledge and then testing it. It is thus vital that he be able to move easily between the development and the execution environment.

There are several types of errors that are reported to the knowledge engineer. These are :-

- spelling errors which are picked up in the knowledge engineering environment
- syntax errors which are picked up at the compiler stage
- run-time errors that are picked up by the inference engine and TMS.

Input is parsed as it is entered and syntax errors are reported immediately. Once a correct clause is entered, the compiled code is generated. Most of the syntax errors made by the knowledge engineer will be be picked up as they are entered, the remainder will be detected at run-time.

If a frame or slot is used that does not exist in an antecedent or consequent, the compiler asks if it must be created. Thus if it is merely a misspelled name, the

knowledge engineer declines whereas if it is a genuine new requirement, it is automatically created.

Run-time errors are usually related to structural changes made at run-time. Another common run-time error occurs when variables that are bound to values, are used as a frame name that do not exist.

The error message tells the user exactly where the error occurred; this includes the frame, rule and actual clause with the error. The knowledge engineering environment takes him to the correct context where the error can be rectified.

There is a trace facility based on the TMS which can be used to debug logical errors. Using the knowledge in the justification network, the TMS can provide a trace of every rule fired in their chronological order. It can also provide a trace of how a fact was found. A trace of a slot with all the values that it has held and how it was changed is also available to the knowledge engineer.

10.3 The Compiler.

10.3.1 Generating an object knowledge base.

The compiler is responsible for taking a source knowledge base, as entered by the knowledge engineer, and turning it into a object knowledge base, as used by the inference engine. It has several components: a lexical analyzer, a parser, an indexer and a code generator.

The lexical analyzer and parser are integrated with the knowledge engineering environment. This allows syntax errors to be detected and corrected at entry time. Once a correctly formed procedural clause has been entered by the knowledge engineer, the compiler generates lisp code for it.

10.3.2 Indexing.

The compiler is responsible not only for the normal parsing and code generation but also for the indexing of the knowledge base. Each slot, variable and relation has two indices:

- an if-index which serves two purposes: to decide firstly which rules are relevant and secondly which clauses'

validity flags are set to "unknown" when the item's value is changed.

- a then-index which is used by the backward chainer to determine which rules set the item's value.

For example in [diagram 8], if the value of slot s1 is changed, then the rules R1 and R2 in frame F1 are made relevant and the first clause in each of these rules has its validity flag set to "unknown". S1's if-index is thus F1.R1.1 and F1.R2.1. (frame.rule.clause-position)

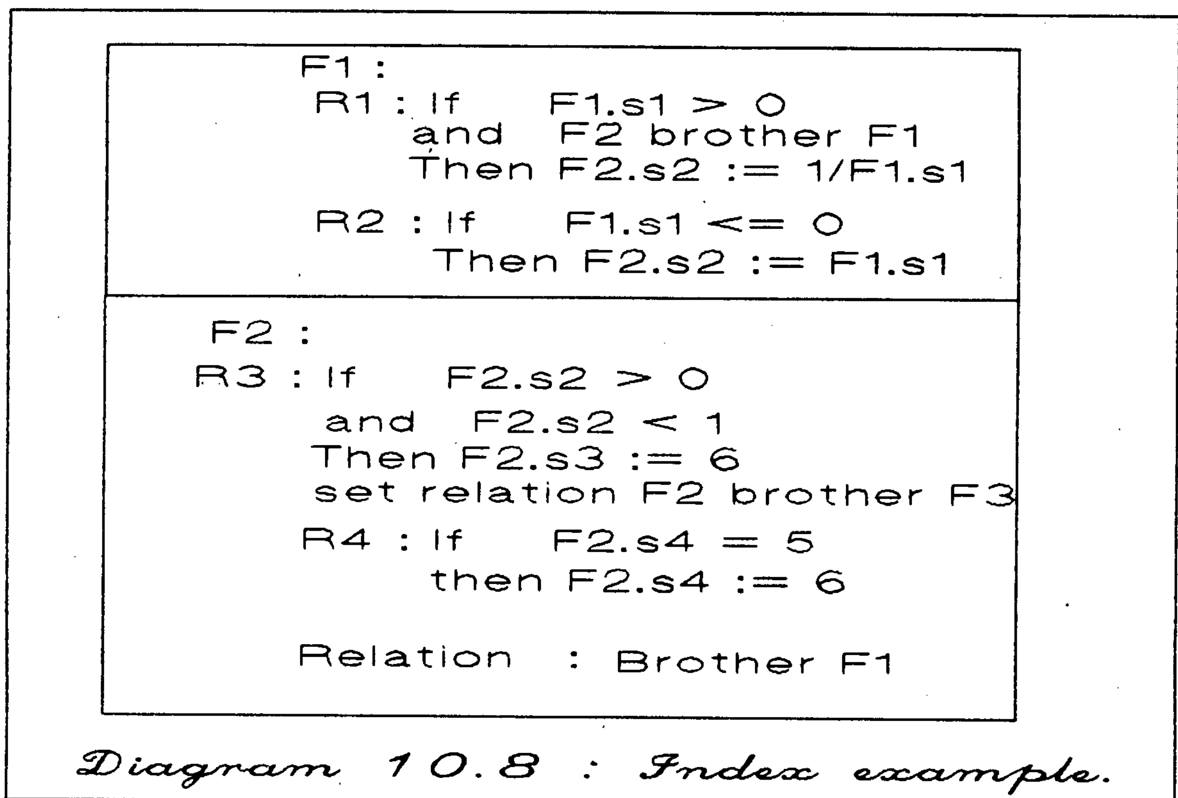


Diagram 10.8 : Index example.

The then index of F2.s3 is set to where its value is set ie F2.R3. The compiler utilizes some intelligence in setting the then-index. For example in [diagram 10.8], the then-index for F2.S4 does not include the rule F2.R4 as this is a

self referencing rule that will cause the backward chainer to loop.

In situations where self referencing loops occur over a series of rules and thus cannot be picked up by the compiler, the inference engine's backward chainer has a mechanism to stop it looping.

As previously stated, the relations are indexed too. For example, the relation "brother" in frame F2 : has a value "F1, an if-index to the rule F1.R1 clause 2, and a then index of F2.R3. The indexing of a knowledge base is done partially at rule entry time and partially at compile time.

The compiler interacts with the knowledge engineering environment to help the knowledge engineer build his knowledge base correctly. If the knowledge engineer references items (slot, relation, frame) that do not exist in the knowledge base, the compiler questions the knowledge engineer as to whether he wishes to have them automatically generated. If so, the compiler finds out the necessary details and creates the new item.

Another field that the compiler adds to the procedural clauses is a contents field. This a list of all the items that a clause references and is used by the TMS to find the support for a clause.

10.4 The User interface.

There are commands for displaying results, or facts, to the end user of the expert system. There are also several other user interface features.

10.4.1 Graphical displays.

The knowledge engineer can display graphical information to the user during a consultation. This graphical information is built using X-windows commands which are made available to the knowledge engineer via external routines.

The knowledge engineer can in fact use the windows package of his choice, provided it is callable from C. The graphics information is captured in files which are called via the "display-graphics" command.

Currently there are no features for achieving moving, or "active", graphics but with a little ingenuity the knowledge engineer can achieve graphics whose movements are linked to values in the knowledge base. A higher level interface to these sort of commands is a feature that will be included in the next version of the WISE shell.

10.4.2 Explanation Facilities.

A crucial aspect of any expert system application is being able to explain how conclusions were reached. There are two approaches to explanation generation :

- one can generate an explanation from the rules that have been fired or actions that have been performed.
- one can have facilities for the knowledge engineer to associate text with certain rules or conclusions. When these rules are fired, or conclusions reached, the application user may request an explanation. This causes the pre-entered text to be displayed.

Both of these features are available to the knowledge engineer on the WISE system. There are HOW and WHAT-IF explanations that are generated by the system using the TMS. The HOW facility is used to explain how a certain conclusion was reached and works on the path taken to get to the conclusion. Because the TMS can provide the reasons for our belief in a fact, the HOW explanations are not cluttered with irrelevant information that is not directly related to the conclusion.

The WHAT-IF facility is a non-monotonic facility that allows the user to ask what would have happened if he had entered

different values when prompted. It uses the same revision of beliefs process that the contradiction resolver uses.

The WHY facility is used to explain to the user why a certain question is asked. At any point in the consultation the user can receive help in the form of explanation text. "Why" text is entered in special why-text slots. When a user is asked for an input, he can enter "?" and the text explaining why that information is required will be displayed. Thus the power of the why facility lies in the hands of the knowledge engineer the explanations will be as good as those he has entered. The explanations are shown to the end user via pre-defined screens.

10.5 Conclusions.

The WISE expert system shell is a complete system that allows the development of large, real life, commercial applications.

This development is facilitated by an easy to use knowledge engineering environment and features for building a user interface.

CHAPTER11 : IMPLEMENTATION DETAILS.

11.1 Introduction.

The following chapter will cover some of the interesting implementation details of the WISE inference engine and TMS. The "why?" concerning the choice of LISP with flavors as the programming language and the "how?" regarding the implementation of the main data structures will be answered.

[Appendix 2] shows a fuller description of the internal specifications of the system.

11.2 The LISP Programming Language.

The WISE shell's inference engine and TMS is written in the LISP programming language. LISP is a symbol manipulation language. A computer works in terms of bits and strings of bits. In LISP, groups of bits are viewed as a code for wordlike objects [Winston and Horn 84]. These wordlike objects are referred to as atoms and are the basic objects of LISP. They are strung together to form sentancelike objects called lists.

Because LISP's basic data structures are atoms and lists, it is an extremely powerful language for the representation of knowledge. Representing knowledge is the act of defining symbols, and conventions for using them [Winston and Horn 84]. Lisp is useful not only for representing knowledge of a domain but also useful for writing routines to utilize this knowledge. Nearly all experts systems and expert system development tools were initially developed in Lisp, although there is a current tendency to rewrite these programs in a conventional language such as C. This is done primarily for performance and portability.

However, with the the dramatic decrease in the cost of computing power, commercial applications written in LISP can now be delivered to the business community at a reasonable cost. There are several popular LISP dialects, but COMMON LISP [Steele 84] has emerged as the industry standard, which means that common LISP programs are now easily ported.

Object orientated programming is a programming paradigm that has some advantages over non-object orientated programming languages. Some of these are :

1 - the user can define general procedures that compute facts about whole classes of objects and then have those procedures inherited by subclasses.

2 - a specific operation will have varying effects on different data, the actual operation is decided at runtime.

Flavors is an object orientated extension to COMMON LISP that is integrated into the LISP language. It has been very useful for the defining of data structures and the actions performed on them.

The language used to develop the WISE system was COMMON LISP with FLAVORS.

11.3 Data Structures

There are several major data structures used by the system, the main ones are obviously the knowledge base and the justification network. There are additional structures, namely the choice-frames, contradiction-frames, symbol-table, relevancy-list and the TMS-pointer-list.

A knowledge base is an object with several fields :

- name : knowledge base name.
- frames : a hash table of frames with each frame hashed on its name.

A frame object is represented in [diagram 11.1]. It has fields for slots, relations, rules, entry conditions, exit actions as well as an index.

The fields that are accessed by name ie slots, rules and relations, are stored in hash tables of objects and the fields that are accessed sequentially ie entry conditions and exit actions, are stored as arrays or vectors of objects. The control knowledge and indices are stored in objects.

Name : the frame's name
Entry-conditions : array of boolean clauses
Own-slots : Hash Table of slot objects
Inherited slots : Hash Table of slot objects
Relations : Hash Table of relation objects
Rule : Hash Table of rule objects
Control Knowledge : Control Knowledge object
Exit Actions : Array of action clause objects
Type : String
Index : Index Object

Diagram 11.1 : The Frame Object.

Each element of the entry conditions array is an object called a boolean clause. Each element in the exit actions array is an action clause. The way that these objects are defined is illustrated in [diagram 11.2]. They both inherit attributes from the clause object.

The procedural knowledge objects are defined in a hierarchy where the lower levels are components of the higher ones. For instance, functions that are defined on the clause object are inherited by boolean clause objects. This allows the

programmer to write general routines for clauses and apply them to the different types of clauses.

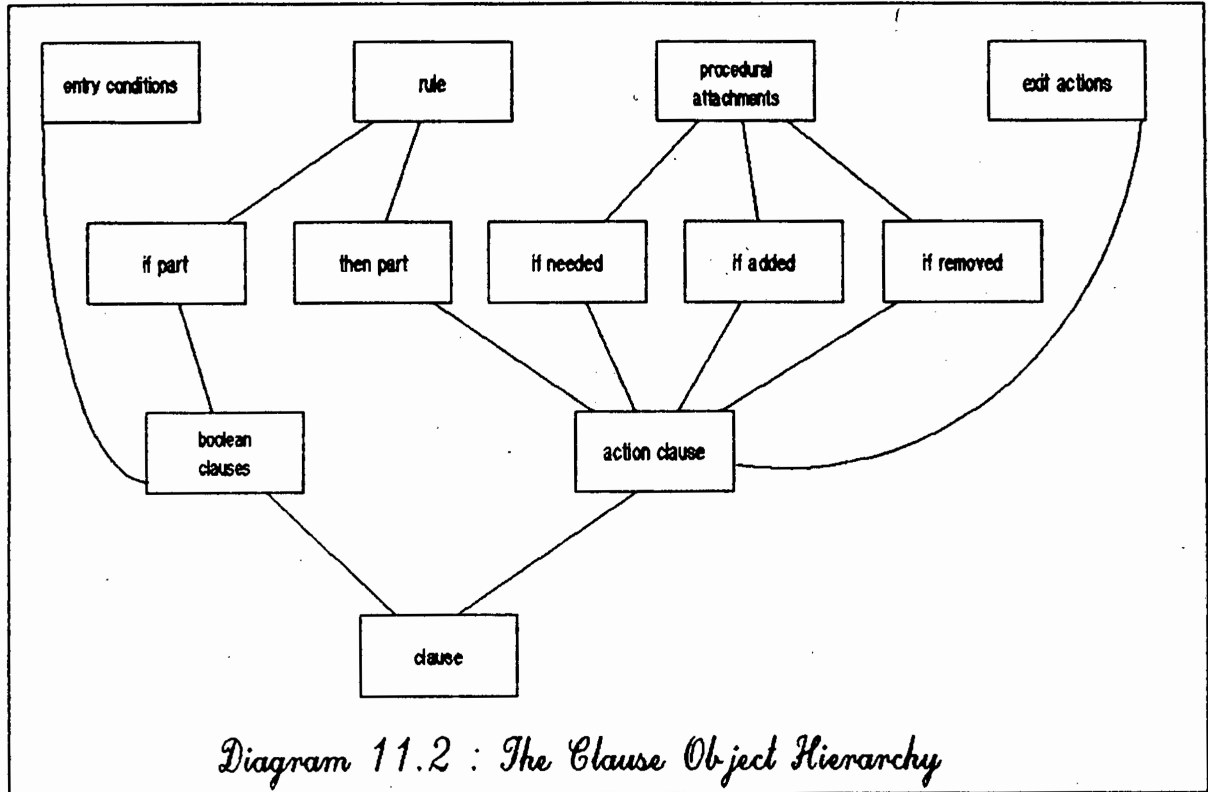


Diagram 11.2 : The Clause Object Hierarchy

A rule object has attributes called name, priority, if-clauses and then-clauses [diagram 11.2]. The priority is a number, if-clauses are an array of boolean clauses and the then-clauses are an array of action clauses.

The own and inherited slots, rules and relations are stored in hash tables hashed according to their names. Each element in the hash table is an object as shown in [diagram 11.1].

The control knowledge is stored as an object with four fields as shown in [diagram 11.3]. Each field has a single value. The refractory and relevancy fields can take the values 'yes

or 'no depending on the control set by the knowledge engineer. The conflict resolution strategy can take specifity, priority, random or any other strategy that the knowledge engineer has built. The search strategy is depth or breadth.

Relevency flag :
search strategy :
conflict resolution strategy
refractory flag

Diagram 11.3 : Control Object

A slot object has fields for each of the facets that a slot has. These facets are: value, default, if-added, if-needed and if-removed [diagram 11.4]. Slots also have indices and the inherited slots have an exception field. The value and default facets take single or multiple values. The procedural attachments are arrays of action clauses [diagram 11.2].

The slot index is an array of index elements with each element representing the rules in a particular frame that the slot accesses. Thus, when the system needs to update the relevancy-list, or validity flags, it need access a frame's rule set only once. For example, [diagram 11.5] shows the if-index for a slot that is accessed in rules that are in several different frames namely F1, F2 and F3.

name : string
retained flag : string
value : single or multi value
default : string or multi value
if-needed : array of action clauses
if added : array of action clauses
if removed : array of action clauses
index : index object

Diagram 11.4 : Slot Object.

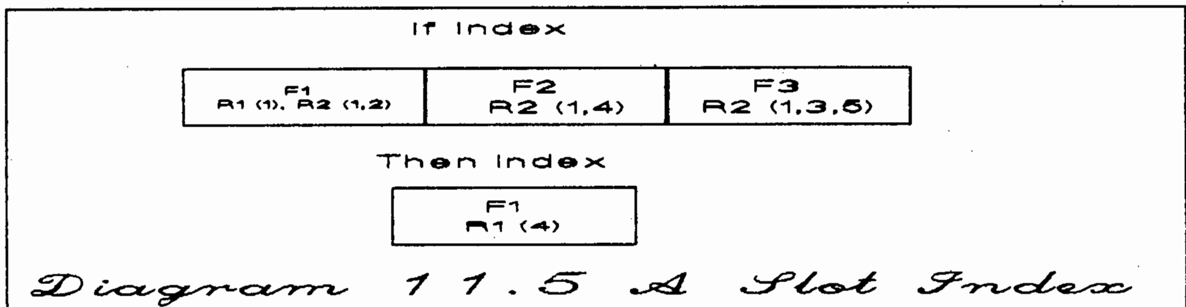


Diagram 11.5 A Slot Index

A choice frame [diagram 11.6] is an object which is used in determining what value to give a choice-variable. It has fields called initial-value and next-value which contain procedural knowledge. It also has own slots and rules.

The **TMS-pointer-list** is a data structure used to keep track of where the value of an item is set in the justification

network. It consists of a nested hash table of arrays. The top level array is hashed on frame names, or variable names.

variable name : the choice variable's name
initial function : array of action clauses
next function : array of action clauses
rules : hash table of rule objects
own slots : hash table of slot objects
previous values : list

Diagram 11.6 Choice frame object.

Within each frame entry is the hash-table of items that have had their values set during the consultation. The entry for each of these elements has an array of pointers to the justification network. In [diagram 11.7] the TMS-pointer entry for F1 is shown. It has a entry for each slot that has been set, namely X and Y. The X entry points to the places where X has been set.

The TMS-pointer-list is used to keep equal entries in the JTMS consistent. It is also used to find the dependencies for a node in the TMS.

The CONTROL-STACK is used in conjunction with the justification network to control the revision of beliefs. The control stack has pointers to the control actions that cause a choice. The choice nodes in turn have a pointer to the control stack. In this way, a choice and the reason for performing are connected. As the revision of beliefs process

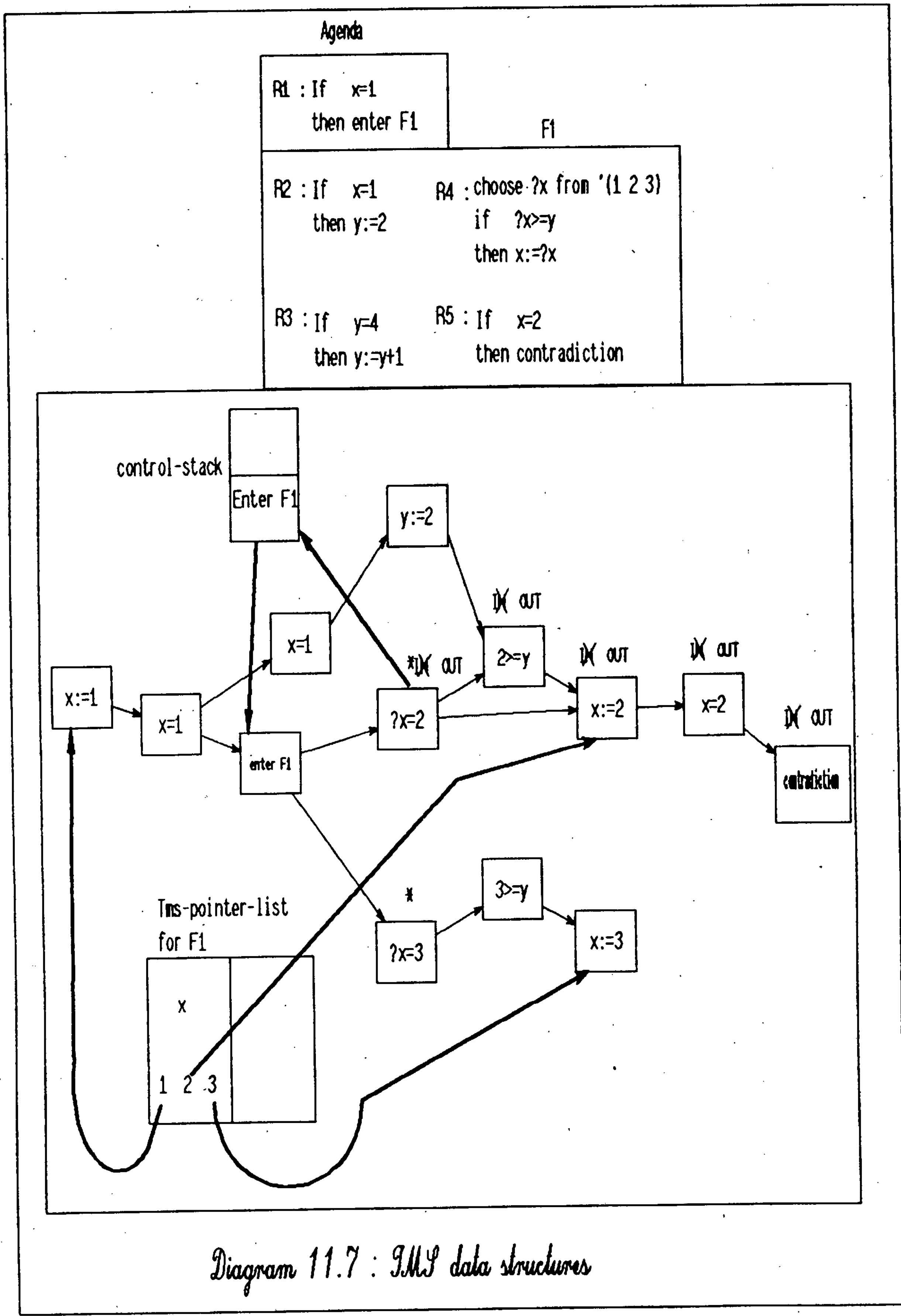


Diagram 11.7 : TMS data structures

propagates-OUT through the justification network, it checks to see if it is OUTing a choice. When a choice (that is not the culprit) is OUTed the system marks the entry in the control stack that points to its "control reason". If at the end of the propagate-OUT process the control action is IN but the choice is OUT, a new choice is made.

In [diagram 11.7] the choice made in the rule, R4, has "Enter F1" as its control reason. When the contradiction causes the choice to be OUTed, it marks this control stack entry. The actions that have been performed are undone during propagate-OUT. This causes the frame F1's conflict resolution set to have R4 and R5 put back in as unfired. Once the propagate-OUT process is finished, it checks the "enter F1" node and sees that it is still IN. Frame F1 is then entered but because R3 and R2 have not been put in the conflict set, they are not considered. R4 is fired and a new choice is made.

Another important data structure is the **relevancy list**. This is the structure that keeps the rules that are relevant in each frame. It is updated from the if-indices of the slots, relations or variables that have been changed.

The variables, their indices and their values are stored in a **symbol table**. The variables are split into three types namely loop, choice and normal.

11. 4 Conclusions.

This chapter has briefly considered some of the design and implementation decisions that were faced when building the WISE system.

I feel that use of LISP and FLAVORS has greatly speeded up the development of the system as compared to the time it would have taken using a conventional language such as C.

CHAPTER12 : USE OF THE WISE SYSTEM.

12.1 Applications built on the WISE system.

The WISE system has been used to build four applications from a variety of domains. Two of the expert systems that are being developed for commercial concerns are of a confidential nature as the companies using them feel that they will provide a competitive advantage over their competitors. These systems are:

1) An advisory system : this expert system advised a manufacturing concern about which materials to use in a manufacturing process.

2) A classification system : this expert system was used for human resources management.

The other two systems are :

1) A network configurer : this is a demonstration expert system that was developed inhouse to exhibit the WISE shell to potential customers.

2) A chemical plant simulation : this expert system will be used by a large chemical concern to discover new methods of chemical processing.

Since the first two systems are confidential, and the network system is for purely demonstration purposes, the chemical plant simulator will be discussed.

12.2 The Chemical Processing Application.

A large chemical concern was looking into the feasibility of processing waste materials into useful products. The waste materials are a by-product of other chemical plants and the final materials that they wish to produce are pure materials such as metals.

The expert system's knowledge base consists of descriptions of processes, descriptions of possible raw materials and incomplete descriptions of possible final products.

The chemical processes that will be used are standard chemical processes such as :

- steam distillation : removes excess oils
- calcining : oxidizes hydrocarbons and metals
- screening : sieves for separating components.

The processes are described in a taxonomy of frames. A frame describing a process will have :

- a description of what inputs it can process. This description includes chemical composition of inputs, size and shape of inputs and temperature for reaction etc.
- a description of a series of chemical reactions that it performs on the input (this is in the form of rules).

- calculations for working out the costs incurred, materials used, and the by-products and waste produced.

The raw materials that the plant will process are waste materials that have a variety of compositions.

The application runs in two directions ie backwards and forwards. When it runs backwards, the user enters a description of the specifications of a final product and the system chains backwards through the processes until it finds a raw material and a route of processes that will make this final product. The basic algorithm is as follows :

1) takes its goal material and for each process that could produce this material do: look to see what the input to the process would have to be to form this material and call this input X.

2) check if X is in the raw material stock. If it is goto 4

3) if not take X and make it a subgoal material goto 1.

4) report solution.

When the system runs forwards the user enters a description of a raw material and possibly some constraints. The system then works out all the materials it could make. It uses the choice mechanism to decide what process to try next. The choice depends on the material and the constraints that the user gives it. The algorithm is as follows :

1) X := raw material

2) use choice mechanism to decide what process to perform on X.

3) X := output of process, if X breaks user constraints then backtrack else if it is a useful final product goto 4 else goto 2.

4) report solution.

This application in effect simulates what could be done to a raw material or how a final product could be made. Its final report includes the cost, use of electricity, water, labor etc, as well as by-products and waste generated.

Using this simulation the company could decide on the cost effectiveness of the plant, the different materials it could produce and the most effective raw materials to process at the plant.

3 Conclusions.

The WISE shell has been used to develop applications from a wide variety of domains. It has proved itself as a useful tool for the development of expert system applications.

PART THREE

CHAPTER13 : RELATED WORK.

13.1 Introduction.

This chapter will cover some of the major commercial expert system shells, that are presently on the market. Up until a few years ago, the market for sophisticated expert system development tools was dominated by two products, namely KEE[c] and ART[c]. Recently, a number of companies have brought cheaper sophisticated products onto the market, the most successful of these being NEXPERT OBJECT[c].

In this chapter three systems will be considered, namely NEXPERT brought out by Neuron Data, KEE brought out by Intellicorp and IBM's ESE.

13.2 NEXPERT OBJECT.

All the information about Nexpert was gleaned from literature brought out by [Neuron Data 87].

13.2.1 Knowledge Representation.

The knowledge representation paradigm used by NEXPERT is a hybrid one, mixing objects and rules. The main representational structure is called an object but is for all intents and purposes, a frame. Objects can represent both concepts and normal objects.

An object has several types of fields:

- name field which is a unique identifier.
- a field representing the one or more classes to which it belongs.
- fields to describe its attributes.
- subobjects which make up its components. This allows for a nested object structure.

Each slot has a couple of parameters that can be customized. These include

- salience
- inheritance relations
- sources of information.
- procedural attachments are stored in structures called meta-slots. Meta-slots include if-change methods and order-of-sources. These methods are inheritable.

An object inherits slots (attributes, properties). One of the positive features of this system is that the user can

define the inheritance strategy. These strategies are declared in meta-slots.

13.2.2 Inference Mechanisms.

The inference engine integrates forward and backward chaining. The same rule may be executed in either direction. NEXPERT automatically generates new goals as new facts become available to it. The system has a mechanism for controlling the resulting focus of attention.

An if-clause's truth is decided by a pattern recognition process which matches it against the knowledge in the objects. These patterns can include existential and universal quantification, as well as the matching of subobjects to an object. A rule clause can refer to a single instance, or a list of instances.

The rules are stored separately from the objects. However the rules have a context mechanism called **contextual links** which allows rules that are related to each other to be stored and used together. The system can focus attention on a relevant group of rules. There is also an agenda which decides the order in which knowledge will be looked at. The agenda can receive messages consisting of goals to investigate with a relative priority. The user can also

control how data that will effect the agenda reaches the system.

The strategy menu allows the user to specify his own control strategies. In the case of conflict resolution, the user can specify run-time priorities on the rules. The strategies use the context structure, and coupled with user specified inheritance methods, an efficient inference process can be designed.

NEXPERT OBJECT allows the user to design his own uncertainty methodology. The overall design principle in NEXPERT seems to be to supply the user with a wide variety of strategies for control, inheritance and uncertainty and to allow the user to choose which one or mixture of ones he wishes to use. In this way, it is similar to the WISE system.

There are powerful tools for non-monoticity which include default reasoning, and a truth maintenance system. The system keeps track of all the logical dependencies discovered during a consultation. The system thus knows the reasons for a conclusion. This allows it to find, and correct, reasons for contradictions and then re-evaluate portions of the reasoning path.

The storing of the reasoning paths and the capacity to propagate contradictions allows one to simulate cases using

the What-if enquiry. It is also used to re-run consultations with different data.

13.2.3 Other Features.

There is an advanced user interface for the capture of knowledge. The object editor is a program used to create objects. Objects can also be created dynamically during the course of a consultation. Once an object has been created, any changes to it results in an incremental compilation.

The editor allows the objects to be defined in any order ie instances first and then classes or visa-versa. The knowledge engineer might want to start with rules; in this case, any object that is referenced in these rules is automatically generated with the correct properties.

The incremental compilation of structures (objects, rules and properties) allows for iterative modification of the knowledge base. The knowledge engineer can start with the rules which will then generate the objects and classes. There is a graphics based knowledge base browser (called NETWORK) that allows the user to view the current state of the knowledge base.

NEXPERT OBJECT provides a graphics environment which is used to build user interfaces. There is a special SHOW operator which allows information to be displayed during the reasoning process. There is also an active values facility, which allows moving graphics.

13.2.4 Conclusion.

NEXPERT OBJECT is a complete shell that uses a mixture of frames and rules. It provides all the tools necessary to build successful commercial expert systems.

13.3 Expert System Environment[c].

Expert System Environment (ESE) is an expert system development tool brought out by IBM. ESE consists of two programs:

- the Expert System Development Environment (ESDE)
- the Expert System Consultation Environment (ESCE).

Both are written in Pascal [ESDT 87]. ESE is claimed to be particularly good at solving structured selection problems.

13.3.1 Knowledge Representation.

The knowledge representation is rule based. The rules are English-like with the aim being that an expert can enter the knowledge without the help of a knowledge engineer. The domain knowledge in the ESDE knowledge base is represented in several ways [Hirsch 86] :

- Parameters (slots) are used to represent facts and constraints. They have a name, type and value. They can be multi-valued and have some procedural constraint tied to them.

- Rules represent the relationships between parameters. It is the rules that are the main representational form in the knowledge base. A rule has an IF-part and a THEN-part.

- Focus Control Blocks (FCB) are used to organize the knowledge base and specify control strategies. Each block has a collection of rules and parameters. Each focus control block is a module of procedural knowledge that is geared towards performing a task.

- Groups are collections of similar knowledge base objects. These objects can be parameters, rules or FCBs. This allows for the grouping of common objects for the purpose of referencing.

- Screens are used to display questions and results. The system supplies a set of default screens. There is also a facility that allows the knowledge engineer to design his own screen, using a screen design editor.

Uncertainty is implemented using certainty factors on all the parameters of the problem domain. The system provides several rule-constructs for the use and assertion of uncertainty. Input from the user can also be given a weighting.

3.2 Control Mechanisms.

IBM have attempted to combine the best of two paradigms :

- by retaining rules as the main means of representation, they hope to allow the expert to enter the knowledge directly into the system himself
- by supplying focus control blocks and control functions, they achieve some of the advantages of frames.

ESDE supplies the user with a control language that allows him to specify the search strategy that the system must use. Each FCB can have a separate search strategy combining forward and backward chaining at will. These search strategies can be explicitly controlled using the control language[Hirsch et at 86].

Overall control is simple; one of the FCBs is designated as the root FCB. The system starts a consultation by entering this FCB and executing its control statements. Any reference to a parameter not in the present FCB causes the system to shift its focus to a new FCB. A consultation terminates when all the work generated from the initial FCB has been completed. This is similar to the agenda system used in the WISE shell.

13.3.3 Other Features.

ESDE has external data routines used to obtain data from an external source such as a database. A parameter can acquire its value in four possible ways it can be set by a rule, inputted by the user, set by the default, or passed from an external data source.

ESDE supplies special procedures for accessing the external data. These are invoked by the backward chainer when a value is required. When the backward chainer must obtain a value from an external source, it executes a user specified procedure.

Apart from this method of obtaining external data there are two control language commands, ACQUIRE and PROCESS, which

operate on external data. The acquire command gets values from an external source and process passes control to an external routine.

13.4 KEE.

KEE is an expert system shell developed by Intellicorp. KEE provides an environment for building expert systems. It provides facilities for:

- the transfer of knowledge from the expert to the system
- representing the transferred knowledge
- rule-based reasoning and lisp functional programming
- a collection of inferencing facilities.

13.4.1 Knowledge Representation.

The KEE knowledge representation is a hybrid one [Intellicorp 86] , consisting of frames to store facts and rules to store the actions to perform on these facts.

A frame is called a unit which is a collection of attribute descriptors or slots. Units are linked together in an inheritance taxonomy. Instance units can inherit slots from

more than one generic unit thus forming a lattice type structure [Lorentin 87].

The developers of KEE used an object-orientated paridigm to program the system. This facility is passed on to the user. Like the WISE system, a slot can hold either a simple value or procedural information in the form of LISP statements or production rules. This procedure is activated by sending a 'message' to the object or by accessing the slots value.

There are 2 kinds of slots :

- Own slots hold information pertaining only to the frame in which they appear.

- Member slots which are inherited by instances of the generic frame.

Partial description of slot values can be represented eg one can represent how many values a slot has and the classes to which it must belong, without actually specifying the value. This allows type checking and a means to represent some negative facts [Richer 86]. The other representational paridigm used in KEE is rules. The rules are parsed and can be used for both forwards and backwards chaining.

Rules are stored as frames with slots for conditions, actions and compiled form. Thus, rules can use the standard subclassing facilities to group themselves into classes.

This allows the rules to be indexed and grouped according to their use.

13.3.2 Inference mechanisms.

KEE provides both backward and forward chaining rule interpreters as well as an advanced facility to retrieve facts. To retrieve a value one, specifies a pattern with variables and the retrieval function retrieves true instances of that statement.

KEE has facilities for conflict resolution which include allowing the knowledge engineer to specify his own conflict resolution strategy.

The backward chainer is used to derive answers to queries. KEE provides different search strategies such as depth-first or breadth-first and the user can specify which he wants used. If he does not specify a particular strategy a default one is used.

KEE has an assumption based truth maintenance system called KEEworlds. This allows the system to assume a solution to a problem and then check if that assumption is true. KEEworlds uses rules to show the consequences of actions in separate 'worlds' without changing the original units

[Filman 87]. Keeworlds supplies the knowledge engineer with tools for creating, exploring, comparing and merging these worlds.

A knowledge engineer will begin by creating a knowledge base with objects that are present in every world. Each new world that is created is a child of this or existing worlds. A new world is described by the facts that differentiate it from other worlds.

New worlds are created and explored using three new types of rules :

- Action rules create a new world when their antecedents are true. The rules perform actions that specify how this new world will differ from its parent world [Intellicorp 86b].

- Reaction rules do not create a new world. They are used to model responses to changes in an existing world.

- Inference rules are used to deduce the implication of facts that are true. They are also used for expressing constraints. After an inference rule has been fired, the rule's conclusions should be true in any world in which its premises are true ; if their consequents are false and their premises are true then the world is inconsistent.

The logical consistency of the worlds is maintained by an assumption-based truth maintenance system. This facility automatically removes any beliefs when the assertion on which they were based becomes false. It also maintains a

list of facts which can not be true simultaneously. This prevents the merging of worlds with contradictory facts.

13.4.3 Other Features.

The knowledge engineer is supplied with a set of tools that facilitate the easy transfer of knowledge from the knowledge engineer into the system. These tools allow the rapid prototyping of systems.

The KEE shell has a facility that allows the user to interact directly with the frames, using an icon which may be either an object or a value of an attribute. Manipulating the graphical images interactively updates the knowledge base. This feature is part of the ActiveImages package which comes with KEE. ActiveImages allows the knowledge engineer to build exciting and usable user-interfaces [Richer 86]. Many of the images are supplied by the package but the knowledge engineer can extend these himself. It is implemented as a KEE knowledge base.

BIBLIOGRAPHY

Aikins, J. S., Prototypical knowledge for expert systems, *Artificial Intelligence* **20** (1983).

Bobrow D. G., and Winograd, T., An overview of KRL, a knowledge representation language in: Brachman R., and Levesque H., (eds), *Readings in Knowledge Representation*, (Morgan Kauffman, Palo alto, CA), (1985).

Brachman, R. J., I lied about the trees (or, Defaults and definitions in knowledge representation, *The AI Magazine*, **6**(3), (1985).

Brachman, R. J., What IS-A is and isn't : An analysis of taxonomic links in semantic networks, *IEEE Expert*, (Septembr 1983).

Brachman, R. J., Whats in a concept : structural foundations for semantic networks, *International Journal of Man-machine studies*, **9**(2) (1977).

Buchanan Bruce, Expert Systems ; Working Systems and the Research Litreature, *Expert Systems*, January 1986 pp32-45

Charniak, E., McDermott, D., *Introduction to artificial intelligence*, (Addison-wesley, Reading, MA) (1985).

Davis, R. and Lenat D., B., *Knowledge Based Systems in Artificial Intelligence* (McGraw-Hill, NY) (1980)

Davis, R., Buchanan, B., Shortcliffe, E., Production rules as a representation for a knowledge base consultation program, in : R. Brachman and H. Levesque (Eds.), *Readings in Knowledge Representation*, (Morgan Kaufmann, Palo Alto), (1985).

Davis, R., Buchanan, B.,G., and Shortliffe, E. H., Production rules as a representation for a knowledge-based consultation system, *Artificial Intelligence* **8**(1) (1977)

Davis, R., Meta-Rules: Reasoning about control, *Artificial Intelligence* **15** pp 179-222 (1980).

De Kleer J., An assumption based TMS, *Artificial Intelligence* **28** (1986).

- b) Extending the ATMS, *Artificial Intelligence* **28** (1986).

- c) Problem solving with the ATMS, *Artificial Intelligence* **28** (1986).

Doyle, J., A truth maintenance system, *Artificial Intelligence* **12** (1979) 231-272.

ESDT, Survey of Expert system development tools, *Expert Systems*, (October 1986).

Etherington , D. W., On inheritance hierarchies with exceptions: Default theories and inferential distance, *Proceedings of AAAI-87*, (1987).

Etherington, D. W., and Reiter, R., On inheritance hierarchies with exceptions, *Proceedings of AAAI-83*, (1983).

Fikes, R., E. and Nillson, N., J., STRIPS A new approach to the application of theorem proving to problem solving, *Artificial Intelligence*, **(2)** 1971.

Fikes, R., Keller, T., The role of frame based representation in reasoning, *Communications of the ACM*, (September 1985).

Filman, R. E., Reasoning with worlds and truth maintenance in a Knowledge-based system shell, *Intellicorp* (1987).

Genesereth, M., R., Ginsburg M.,L., Logic Programming, *Communications of the ACM*, (September 1985).

Georgeff, M., P., Procedural control in production systems, *Artificial Intelligence* **18** pp 175-201 (1982).

Ginsberg, M. L., and Smith, D. E., Reasoning about actions II: in: Brown, F. M., (ed.), *Proceedings of the 1987*

workshop on the frame problem in artificial intelligence, (Morgan Kaufmann, Los Altos, CA) (1987).

Ginsberg, M. L., Introduction of Ginsberg, M. L., (Ed.) *Readings in non-monotonic reasoning,* (Morgan Kauffman, Los Altos, CA) (1987).

Halpern, J.Y. and Moses, Y.O., A guide to the modal logics of knowledge and belief, in: Proceedings *of IJCAI-85,* Los Angeles, CA (1985) 480-490.

Hayes, P. J., The logic of frames, in : R. Brachman and H. Levesque (Eds.), *Readings in Knowledge Representation,* (Morgan Kaufmann, Palo Alto), (1985).

Hayes-Roth, F., Rule based systems, *Communications of the ACM,* (September 1985).

Hayes-Roth, F., Waterman, D., Lenat, D. B., (Eds.) *Building Expert systems,* (Addison-wesley, Reading, MA) (1983).

Hendrix, G., Encoding knowledge in partition networks, in: Findler, N. V., (ed.) Associative Networks: *Representation and use of knowledge by Computers,* (Academic Press, NY) (1979).

Henson, D., G., Gold works expert system user's guide, *Gold Hill Computers,* 1987.

Hewitt, C., Planner : a language for proving theorems in robots, Proc. *IJCAI-71*, London, England (1971)

Intellicorp, KEE software development system user's manual, Version 3.0 (1985).

Jackson, P., **Introduction to Expert Systems** (Addison-Wesley, Reading, MA.) (1983).

Kunz, J. C., Shortcliffe, E. H., et al. Computer assisted decision making in medicine, *Journal of Medicine and Philosophy,* (1984).

Lifshitz, V., Pointwise circumscription, in: Ginsberg, M. L., (Ed.) *Readings in non-monotonic reasoning,* (Morgan Kauffman, Los Altos, CA) (1987).

Marcus, S., Stout,J., and McDermott, J., VT: An expert elevator designer that uses knowledge based backtracking, *AI magazine*, **(9.1)** (Spring 1988).

Martins, J., P. and Shapiro, S., C., A model for belief revision, Artificial Intelligence **34(1)**, (1988).

McCarthy , J., Circumscription a form of non-monotonic logic, *Artificial Intelligence* **13**(1,2), (1980).

McCarthy, J., Applications of circumscription to formalizing common sense knowledge, *Artificial Intelligence* **28**, (1986).

McCarthy, J., Epistemological problems of artificial intelligence, in: Ginsberg, M. L., (Ed.) *Readings in non-monotonic reasoning,* (Morgan Kauffman, Los Altos, CA) (1987).

McCarthy, J., and Hayes, P., Some Philosophical problems from the standpoint of artificial intelligence, in: Meltzer, B., and Michie, D., (Eds.), *Machine Intelligence 4,* (Edinburgh University Press) (1969).

McDermott, D., V. and Sussman. G., J., The CONNIVER refrence manual, MIT AI memo 259a, MIT, (1974).

Mcdermott, D. and Doyle, J., Non-monotonic logic, *Artificial Intelligence* 13 (1980) 41-72.

McDermott, D., Contexts and data dependencies: A synthesis, *IEEE Trans. Pattern Anal. Machine Intelligence*, 5(3) 1983.

Mcdermott, D., and Mcallister, D., What is a TMS?, *AAAI-88 TMS Tutorial*, (1988).

- Implementing TMSes, *AAAI-88 TMS Tutorial,* (1988).

- Inference with a TMS, *AAAI-88 TMS Tutorial,* (1988).

Charniak, Riesbeck, McDermott and Meehan, *Artificial Intelligence Programming (2nd edition)*, (Lawrence Erlbaum) (1987).

Mettrey, W., An assessment of tools for building large knowledge based systems, AI Magazine 8(4) (1987).

Miller R. A., Internist/Cadeceus Problems facing Expert Consultation Programs, *Methods of Medical Information*, (1984).

Minsky, M., A framework for representing knowledge in: J., Haugeland (ed.), *Mind Design*, (MIT Press, Cambridge MA) (1981).

Nado, R., and Fikes, R., Semantically sound inheritance for a formally defined frame language with defaults, *AAAI-87*, (1987).

Neuron Data, *Nexpert Object Promotional Material*, (1987).

Newell, A,, and Simon, H. A., GPS, a program that simulates human thought, in: Feigenbaum, E., and Feldman J., (eds) *Computers and Thought*, (McGraw-Hill, New York) (1963)

Nillson, J., N., *Principles of Artificial Intelligence*, (Springer-Verlag, New York) (1982).

O'Hare G.M, Bells D.A., The co-existence approach to Knowledge representation, *Expert Systems*, (October 1985).

Orlawska, E., Pawlak, Z., Expressive powers of representation systems, *International journal of man-machine studies*, (1984)

Pearl, J., *Probalisitic Reasoning in Intelligent Systems: Networks of Plausible Inference*, (Morgan Kaufmann, CA), (1988).

Petrie C., J., Revised dependency directed backtracking for default reasoning, *Proceedings of the AAAI-87*, (1987).

Petrie C., J., Revised dependency directed backtracking for default reasoning, *Proceedings of the AAAI-87*, (1987).

Post, E., L., Formal Reductions of the general combinatorial decision problem, *American Journal of Mathematics*, **(65)** (1943).

Rapheal, B., The frame problem in problem solving, in: Findler, N. V., and Meltzer, B., (Eds.), *Artificial Intelligence and Heuristic Programming*, (Edinburgh University Press, Edinburgh) (1971)

Reiter, R., On reasoning by default, in : R. Brachman and H. Levesque (Eds.), *Readings in Knowledge Representation*, (Morgan Kaufmann, Palo Alto), (1985).

Rich, E., *Artificial Intelligence*, (McGraw-Hill, NY), (1983).

Richer, M., H., An evaluation of expert system development tools, *Expert Systems*, (July 1986).

Rulifson, J. L., et al., QA4, A procedural calculus for intuitive reasoning, Stanford Research Institute, Technical note 73, 1972

Shoham, Y., Non-monotonic logics, in: M. L., Ginsberg (Ed.), *Readings in non-monotonic reasoning,* (Morgan Kauffman, Los Altos, CA) (1987).

Stallman, R.M. and Sussman, G.J., Forward reasoning and dependency directed backtracking, a system for computer aided circuit design, *Artificial Intelligence* **9**(2) (1977) 135-196.

Steele, G., L., Jr., *COMMON LISP Refrence Manual*, (Digital Press Bedford MA), (1984).

Stefik, M. et al, The organization of expert systems a prescriptive tutorial, in: F.H. Roth et al (Eds.) *Building Expert Systems*, (Addison Wesley, Reading, MA.) (1983).

Stefik, M., Planning with constraints (MOLGEN: Part 1), *Artificial Intelligence* **16** pp 111-140 (1981).

- Planning and meta-planning (MOLGEN: Part 2), *Artificial Intelligence* **16** pp 141-170 (1981).

Takenouchi H., Iwashita Y., An integrated Knowledge Representational Scheme For Expert Systems, *Expert Systems*, (February 1987).

Waldinger, R., Achieving several goals simultaneously, *Artficial Intelligence Center Technical Note* (107), Stanford research instituite, (1975).

Waterman, D.A., *A guide to expert systems*, (Addison Wesley, Reading, MA, 1985).

Williams C., Expert Systems, Knowledge Engineering and AI tools - an Overview, *IEEE Expert*, Winter 1986 pp66-70.

Winograd, T., Extended inference modes in reasoning by computer systems, *Artificial Intelligence* **13**(1,2), (1980).

Winston P., H., *Artificial Intelligence, Second Edition*, (Addison-Wesley, Reading, MA) (1984).

Winston, P., H., and Horn, B., K., P., *Lisp 2nd edition*, (Addison-Wesley, Reading MA), (1984).

Woods, A., Whats Important about knowledge representation?, *IEEE Expert*, (Winter 1986)

APPENDIX1 : WISE LANGUAGE DESCRIPTION.

The WISE language allows the knowledge engineer to manipulate frames, their attributes and their relations. It is used in the formation of rules and procedural attachments.

In this chapter I will describe the language using the following syntax.

A production has the form :
left hand side = right hand side
[x|y] means x or y
{x}* one or more instance of x
{x}+ zero or more instances of x
Atoms are described in quotes eg. "x" means the letter x.

This is not meant to be a formal description, as at times certain features will be described in normal text form.

A rule is made up of an if part and a then part :
rule = [{loop-clause}+ | {choice-clause}+] {if-clause}*
{then-clause}*

if-clause = [comparative-clause | relational-clause |
boolean-clause]

comparative-clause = expression comparative-operator expression

expression = [slot-access | number | string | value-function] operator expression

slot-access = ["the" slot-name "of" slot-access | frame-name "." slot-name | "the" number "element of" slot-access]

value-function = [number-children | number-parents | slot-cardinality]

number-children = ["the amount of children of" | "amount children"] frame-access

number-parents = ["the amount of parents of" | "amount parents"] frame-access

slot-cardinality = [the amount of elements of" | "amount elements] slot-access

frame-name = string

slot-name = string

string= [character | number | separator]*

operator = [+ | - | / | * | "concatenate"]

comparative-operator = [< | > | <= | >= | =]

relational-clause = frame-access relation frame-access

frame-access = [frame-name | slot-access | variable]

relation = [user-defined | system-defined]

user-defined = string

system-defined = [subclass | member | superclass | superset]

member = [frame-access ["is in the class of" | "is a"]
frame-access]

subset =frame-access ["is a subset of" | "is a kind of"|
"is a"] frame-access

superclass = frame-access ["is a super class of mammals" |
"contains"]

superset = frame-access ["is a super set of"| "contains"]

boolean-clause = [check-commands | action-commands]

check-commands = type expression

type = ["numeric" | "alpha-numeric"]

The check-commands return true if the expression is of the type specified. The action commands are consequents that can be used in the antecedents of rules. For instance:

- enter-frame (described below) will return true if the frame is successfully entered else it will return false.

- execute-a-rule will return true if the rule succeeds else it returns false.

- the other actions all return true.

action-command = then-clause

Then-clause = [set-commands | in-out-commands |control-commands | "contradiction"]

set-commands = [set-slot | set-relation | set-control | set-frames]

set-slot = [set-single | set-multi]

set-single = ["set" {"-inherited"}+ | "reset" {"-inherited"}+] slot-access ":=" expression

set-multi = "add"{"-inherited"}+ expression "to"
{"front of"}+ slot-access
= "remove"{"-inherited"}+ expression "to"
{"front of"}+ slot-access

If the "-inherited" extension is used then the system looks for an inherited slot of that name before a own slot.

set-relation = [add-a-relation | remove-a-relation]

add-a-relation = "set relation" frame-access relation frame-access

remove-a-relation = "remove relation" relation from frame-access

relation = string

"set-control" frame-access control-access expression

control-access = [["s-strategy" | "search strategy"] |
["crs" | "conflict resolution strategy"]
| "relevancy" | "refractory"]

This command dynamically changes the control parameters of a frame.

set-frame = [add-frame | remove-frame]

add-frame = "add frame" frame-name frame-access

remove-frame = "remove-frame" frame-access

In-out-commands = [input-commands | output-commands]

input-commands = [get-value | choose-a-value | choose-a-graphic | choose-from-form]

get-value =

"get a value" {":header" string}+ {":help" string}+

choose-a-value =

"get a value" {":header" string}+ {"options" [slot-access | {string}*] {":help" string}+

choose-a-graphic =

"get a graphic" {":header" string}+ {":options" [slot-access | {string}*] {":help" string}+ {":file" string}+

choose-from-form =

"get a form" {":header" string}+ {":options" [slot-access | {string}*] {":help" string}+ {":form" string}+

output-commands = [show-values | show-graphics | show-form]

show-values = "display" {[expression|frame-name|string]}*

show-graphics = "display" file-name {[expression|frame-name|string]}*

show-form = "display" "file-name" {[expression|frame-name|string]}*

control-commands = [execute-rule | enter-frame]

execute-rule = ["execute" | "execute the rule"] rule-name

enter-frame = ["enter" | "enter the frame"] frame-name

Universal quantification is achieved using loop rules. These rules allow statements such as:
For all the managers set their salaries to $20,000.

The syntax of a loop-rule is :
loop-rule = {loop-clause}* {if-clause}* {then-clause}*
loop-clause = "loop" var-access" on [list | slot-access | "children of" frame-access | " instantiations of" frame-access | "parents of" frame-access"]

Existential quantification is achieved through the use of choice rules. choice rules allow you to make statements such as :
If there is some man who is a manager then set that man's salary to $20,000.

choice-rule = {choice-clause}* {if-clause}* {then-clause}*

choice-clause = "choose" var-name [{ "." slot-name}* | var-name] ["using" | "from"] [choice-frame-access | slot-access]
choice-frame-access = frame-name

A1.2 Compiled Form of the knowledge base.

In this section the compiled form of the language is shown. Each command of the language has a compiled form that can be understood by the lisp evaluator.

Slot access.

A slot access is compiled to a function called **f-get** that takes a series of parameters and returns a value or cardinality of a slot.

Examples

employee.name or the name of the employee is compiled to :

```
(f-get 'employee 'name).
```

amount of courses-to-schedule is compiled to

```
(f-get 'courses 'courses-to-schedule 'cardinality)
```

number of parents F1 is compiled to :

```
(num-parents 'F1)
```

The antecedent F1.s1 + 4 > F2.S2 + number of children F1 :

```
(> (+ (f-get 'f1 's1) 4) (+ (f-get 'f2' s2)(num-child 'F1)))
```

Relational Clauses.

The relational clauses return true or false and are compiled to a function called r-get.

Example : F1 brother F2 is compiled to :

```
(r-get 'F1 'brother 'F2)
```

Boolean Clauses.

A boolean clause is any clause that returns true or false, most of the action clauses can be used in as an if-clause thus for example it is legal to say :

Set commands

The set commands are compiled to a series of put functions for example :

```
set F1.S1 := 4 is compiled to :
(f-put 'F1 'S1 'own-slot 4)

set-inherited F1.S1 := F2.S2 + 6
(f-put 'F1 'S1 'inherited (+ (f-get 'F2 'S2) 6))

add 6 to F1.S1
(add-a-multi 'F1 'S1 'own-slot 6)

add-inherited 'y to front of F1.S2
(add-a-multi 'F1 'S2 'inherited 'y 1)

set relation 'F1 (the value of F2.s3) 'F2
(rel-put 'F1 (f-get 'f2 's3) 'F2)
```

```
set control 'F1 search strategy 'depth
(control-put 'F1 'search-strategy 'depth)

add frame 'f1 'employee
(add-a-frame 'f1 'employee)

remove frame 'f1
(remove-a-frame 'f1)
```

Control Commands

execute the rule R1 in frame F1 is compiled to :

(exec-rule 'F1 'R1)

enter the frame F1 is compiled to :

(enter-frame 'F1)

Input/Output commands.

examples :

```
employee.name := get a value :header "what is the employee
                 name" :help "input the employees name"
     (get-val :header "what is the employee name" "input the
employees name")

get a graphic  :header "Choose one of the processes"
               :options (process1 process2 process3)
               :help "This picture represents the three
processes enter 1,2 or 3 to choose a process"
```

```
              :file "process-picture"
    (get-graphic   :header "Choose one of the processes"
              :options (process1 process2 process3)
              :help  "This  picture  represents  the  three
processes enter 1,2 or 3 to choose a process"
              :file "process-picture")
```

Loop rule

```
    loop ?x on children of 'F1
    if   ?x.s1 > 0
    then ?x.s1 := 5

if   (> (f-get (var-get '?x 'loop) 's1) 0)
then (f-put (var-get '?x 'loop) 's1 5)
```

Choice-rules

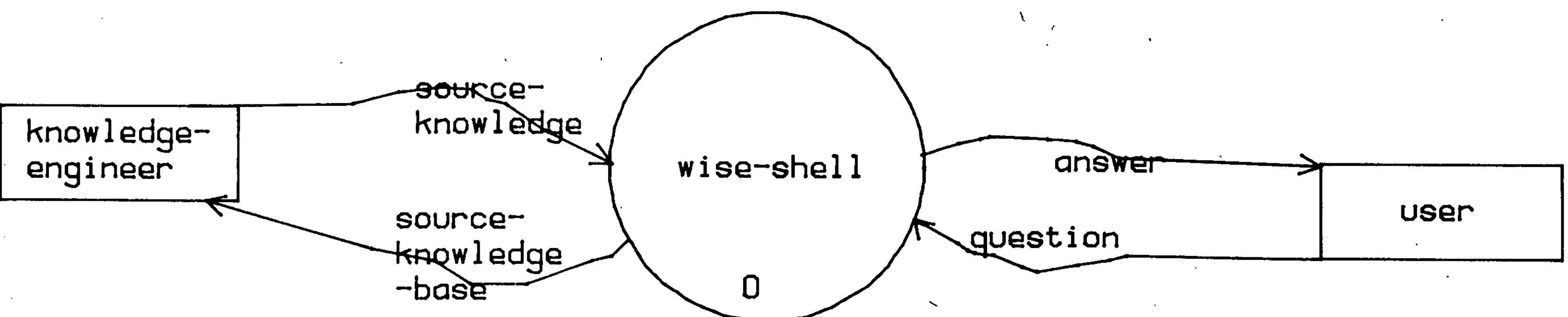
```
choose ?x using c-frame
if ?x.s1 > 0
then ?x.s1 := 5

if (> (f-get (setq temp-val (get-the-choice-value '?x 'c-
frame)) 's1) 0)
then (f-put temp-val 's1 5)
```

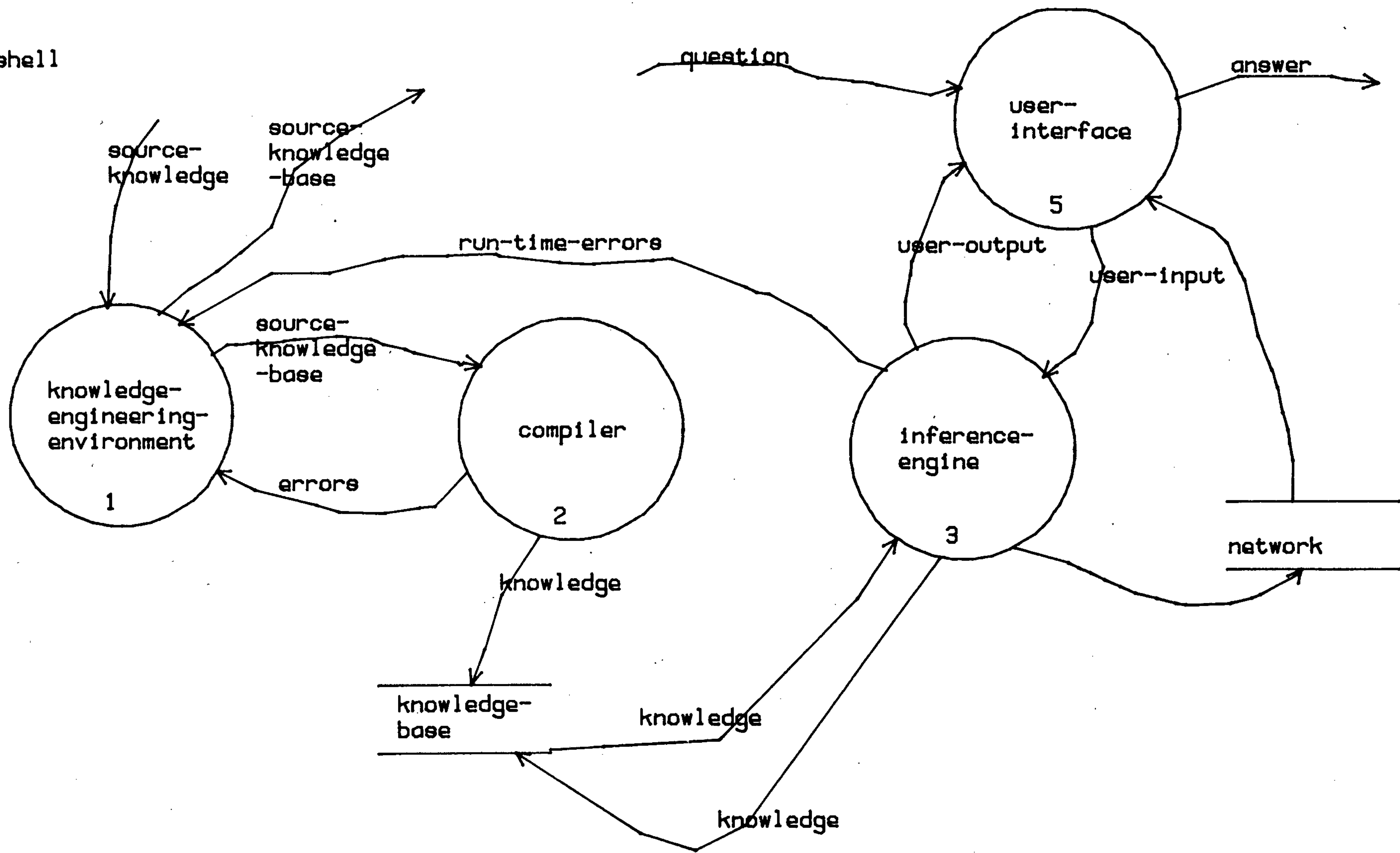
APPENDIX2 : SPECIFCATION DIAGRAMS FOR INFERENCE ENGINE AND TRUTH MAINTENANCE SYSTEM.

This appendix shows some of the diagrams that were used to specify the WISE inference engine and truth maintenance system. The diagrams were produced by a computer aided software engineering (CASE) tool called teamwork.

Context-Diagram; 5
top level



0; 8
wise-shell



3.17
inference- engine

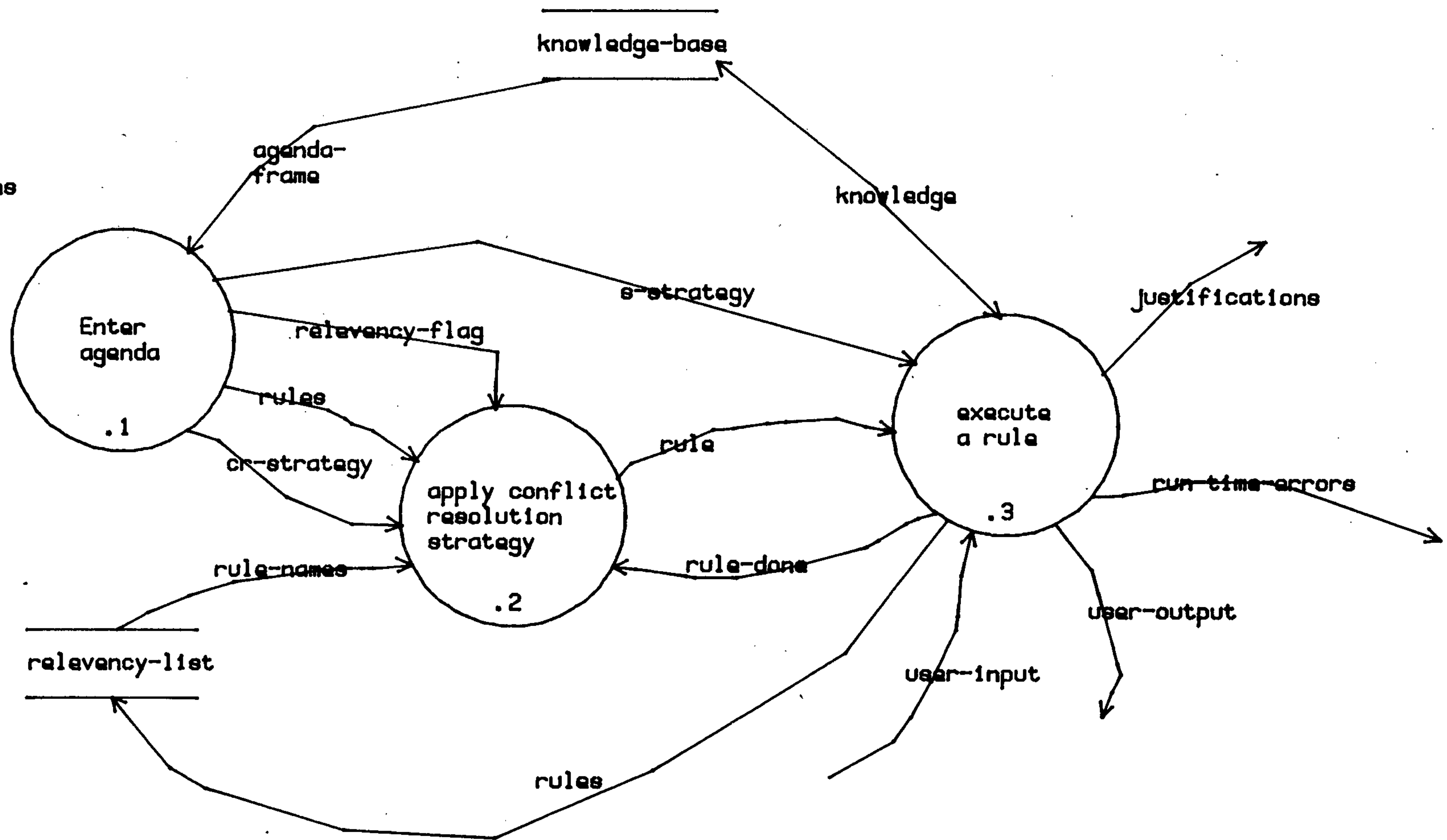
* out * node

* out * errors

* out * output

* in * input

* in * justifications



3.2.10
apply conflict resolution strategy

* in * relevancy-list

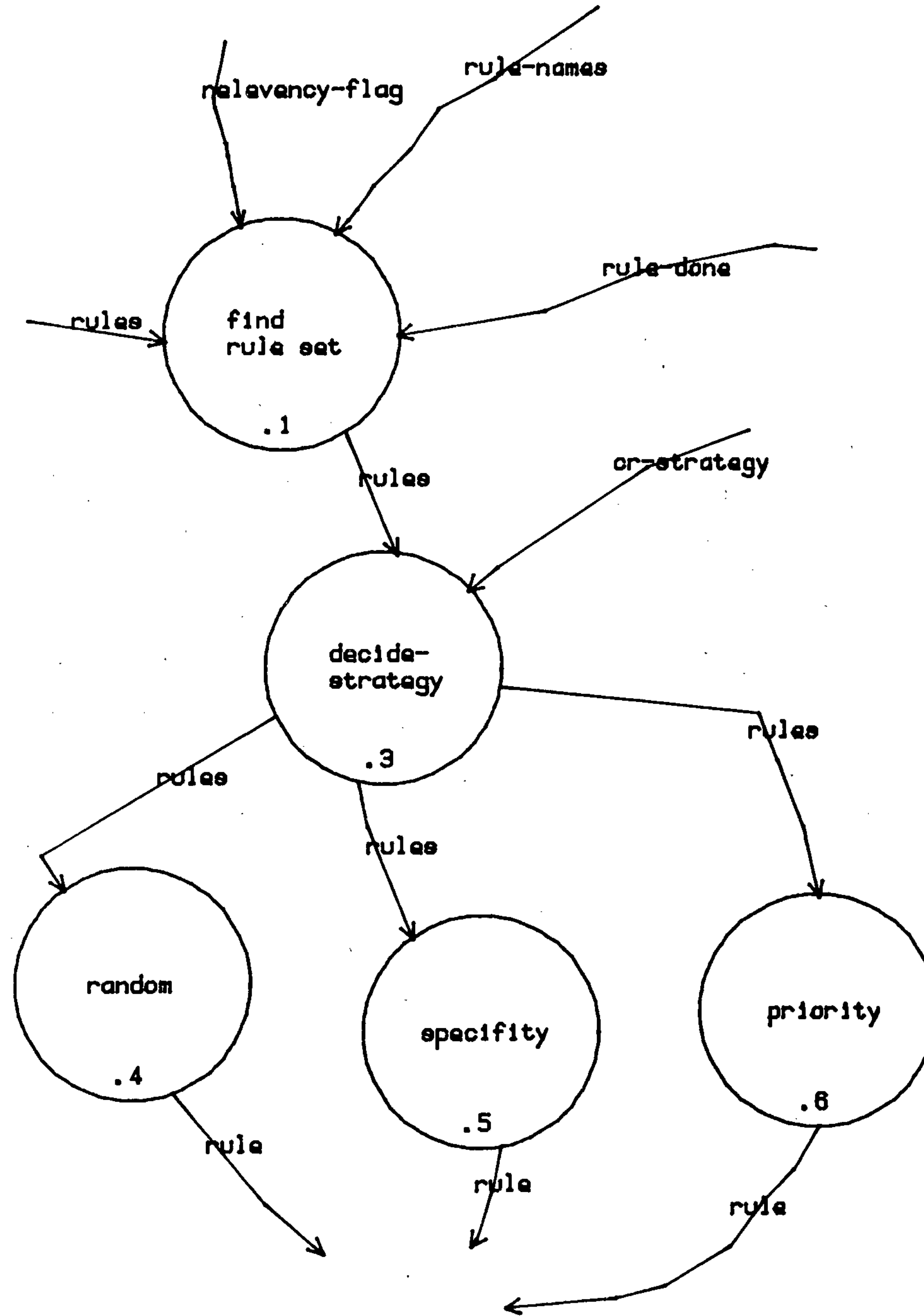
* in * rule-done

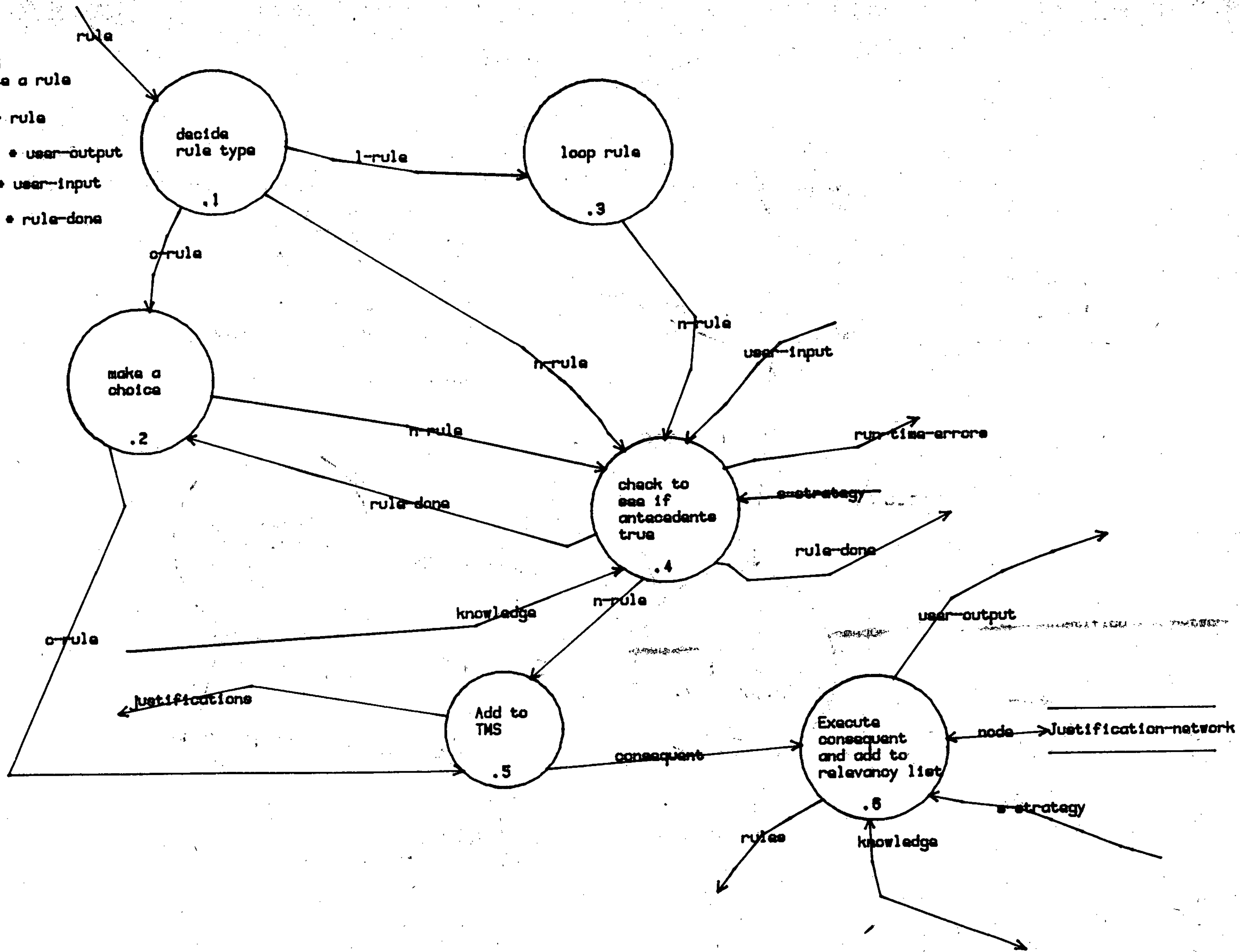
* in * cr-strategy

* out * rule

* in * rules

* in * relevancy flag




3.3.25
execute a rule
* in * rule
* out * user-output
* in * user-input
* out * rule-done
rule
decide rule type
.1
l-rule
loop rule
.3
c-rule
n-rule
n-rule
n-rule
make a choice
.2
user-input
n-rule
run-time-errors
rule-done
check to see if antecedents true
.4
s-strategy
rule-done
knowledge
n-rule
c-rule
user-output
justifications
Add to TMS
.5
consequent
Execute consequent and add to relevancy list
.6
node
Justification-network
s-strategy
rules
knowledge

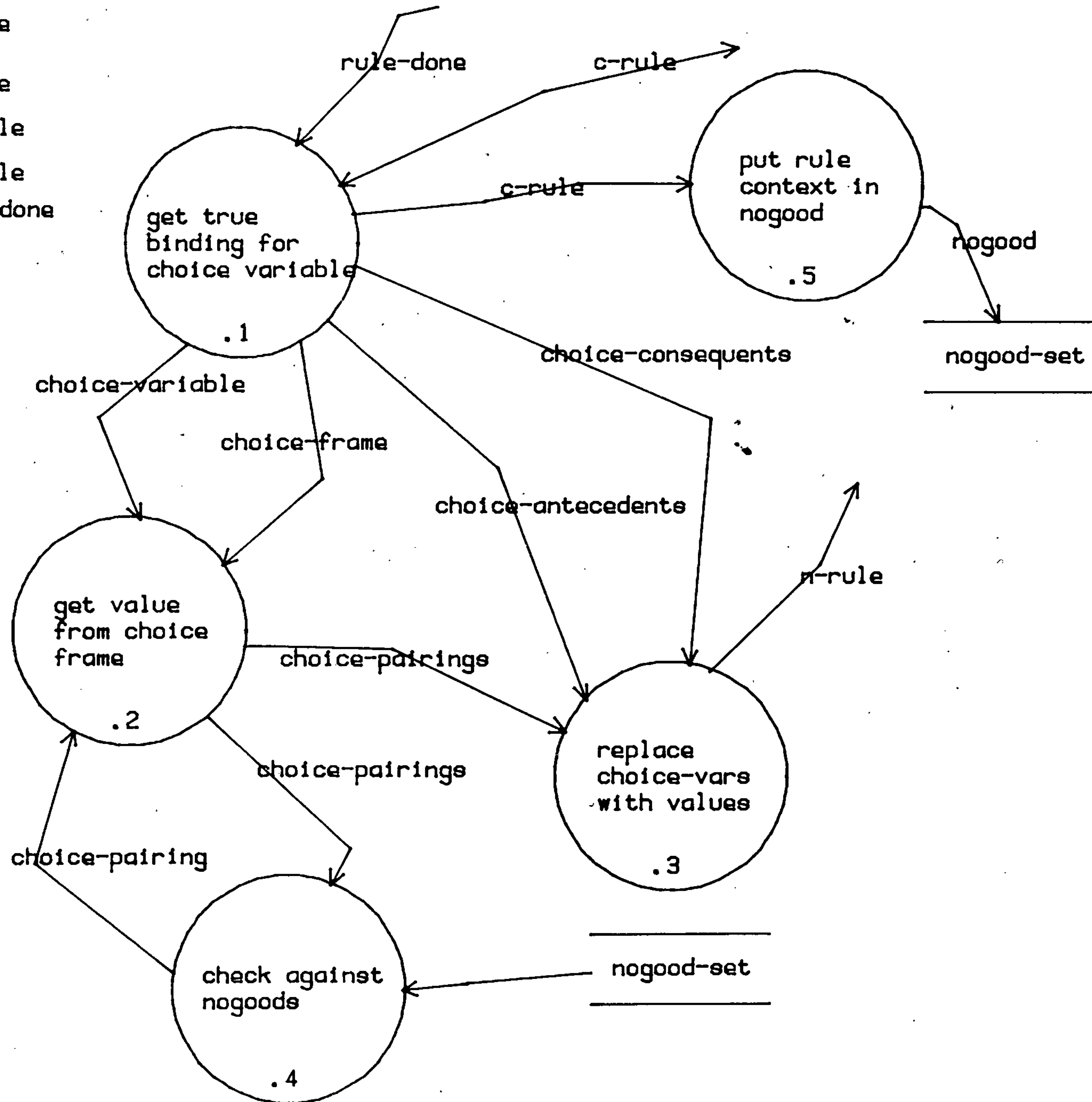
3.3.2.9
make a choice

* in * c-rule

* out * n-rule

* out * c-rule

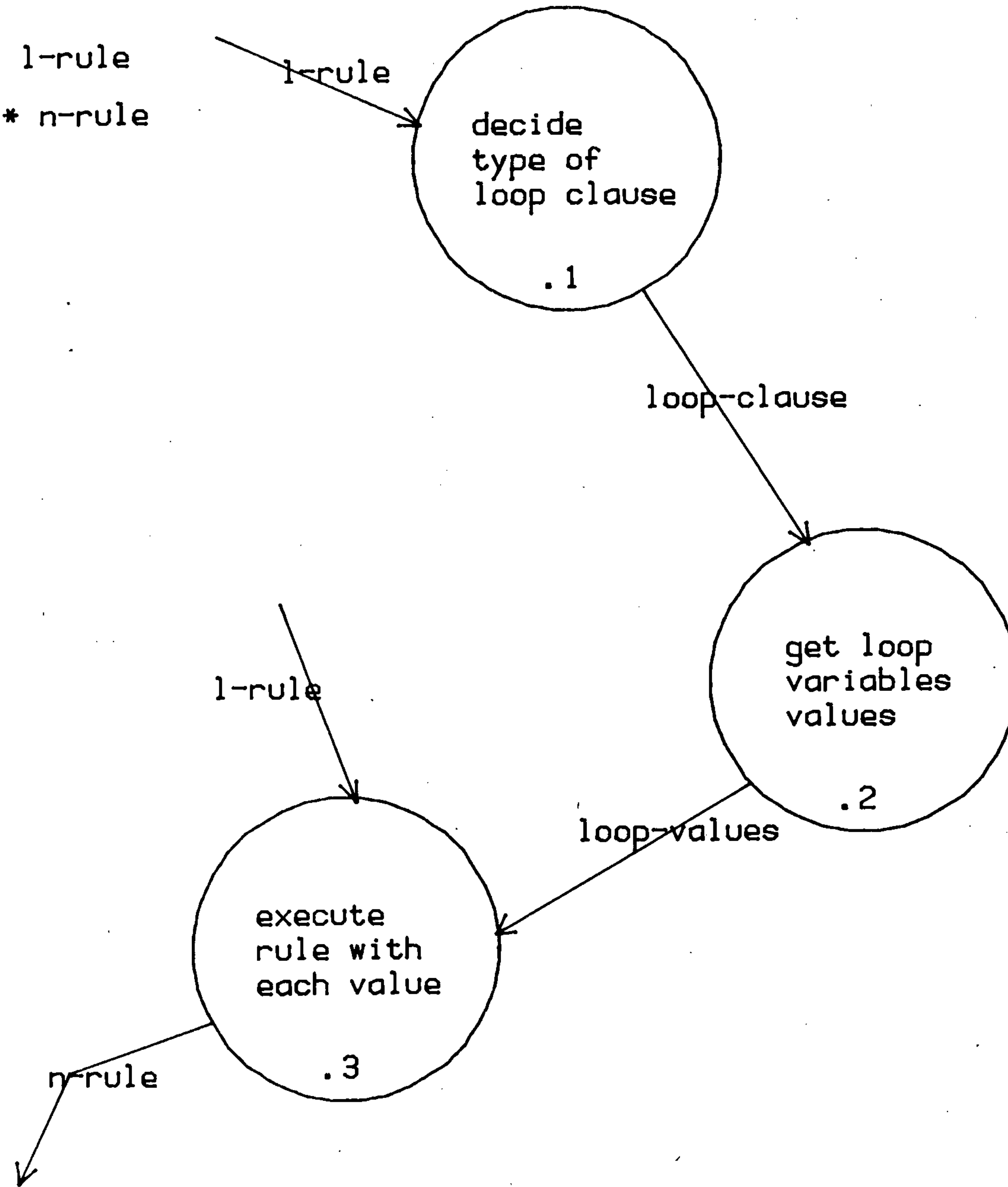
* in * rule-done



3.3.3;1
loop rule

* in * l-rule

* out * n-rule



3.3.4;6
check to see if antecedents true

* in * s-strategy

* in * n-rule

* in * user-input

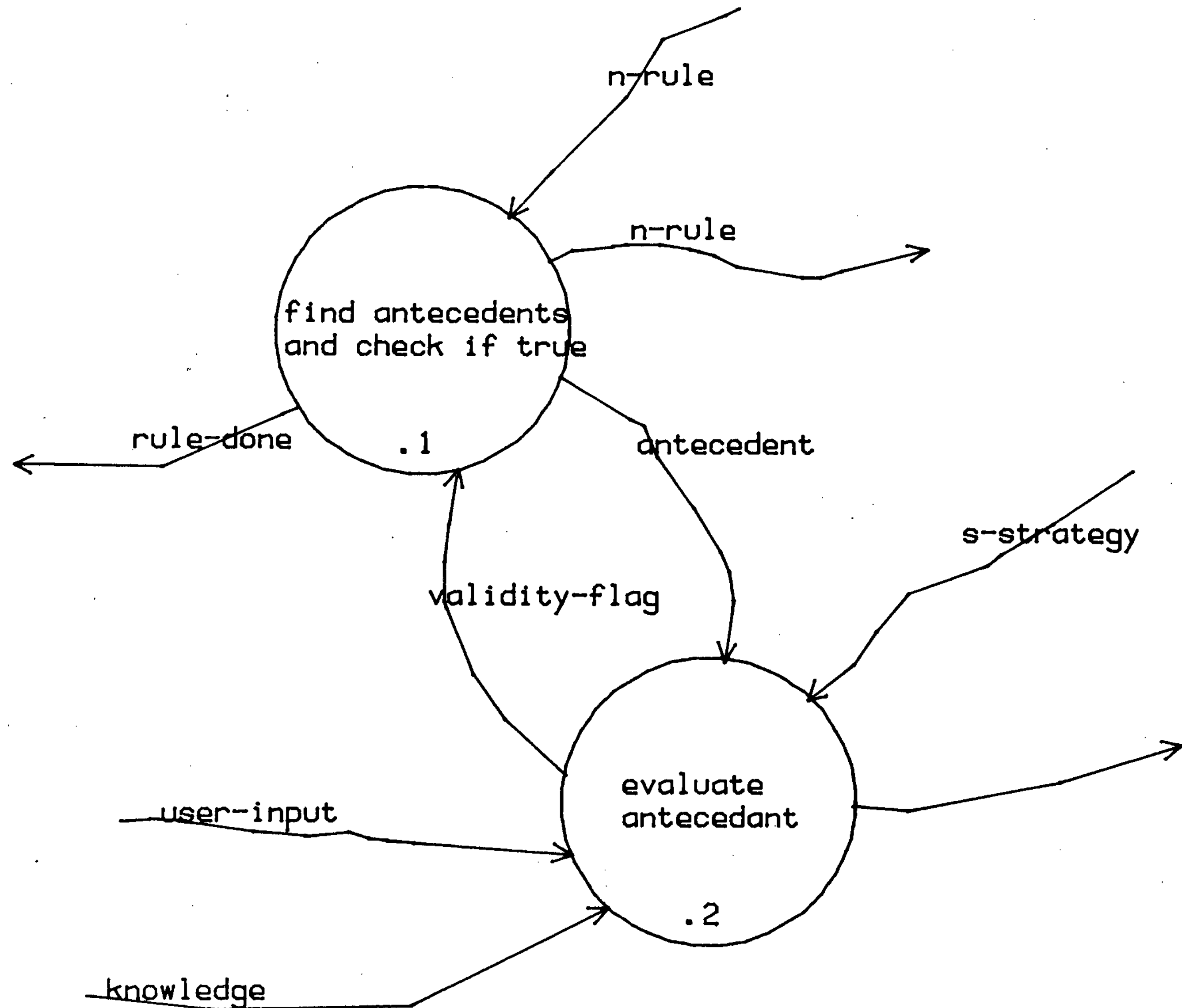
* out * validity-flag

* out * rule-done

* out * run-time-errors

* in * knowledge

* out * n-rule



3.3.4.2.10
evaluate antecedant

* in * antecedent

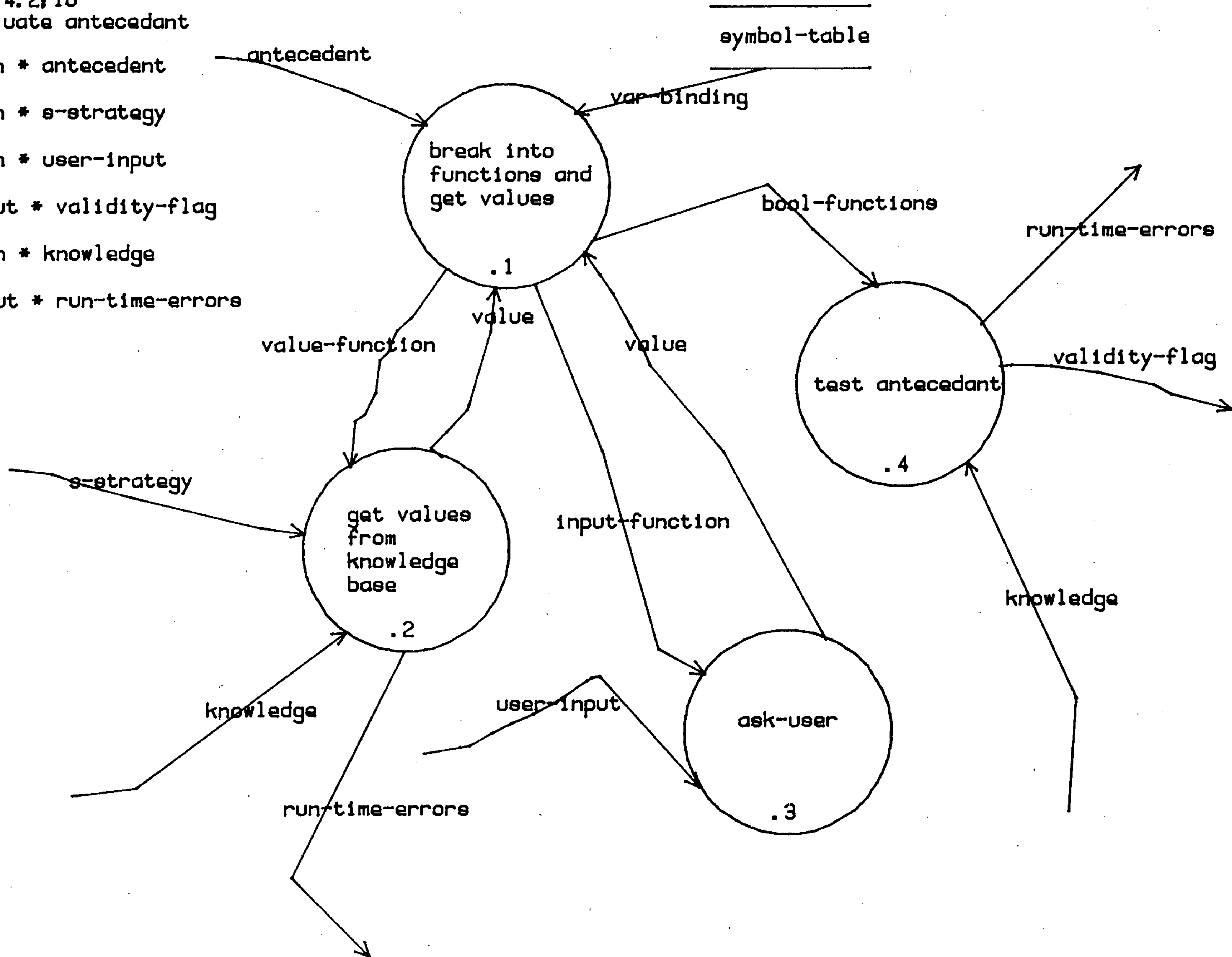
* in * s-strategy

* in * user-input

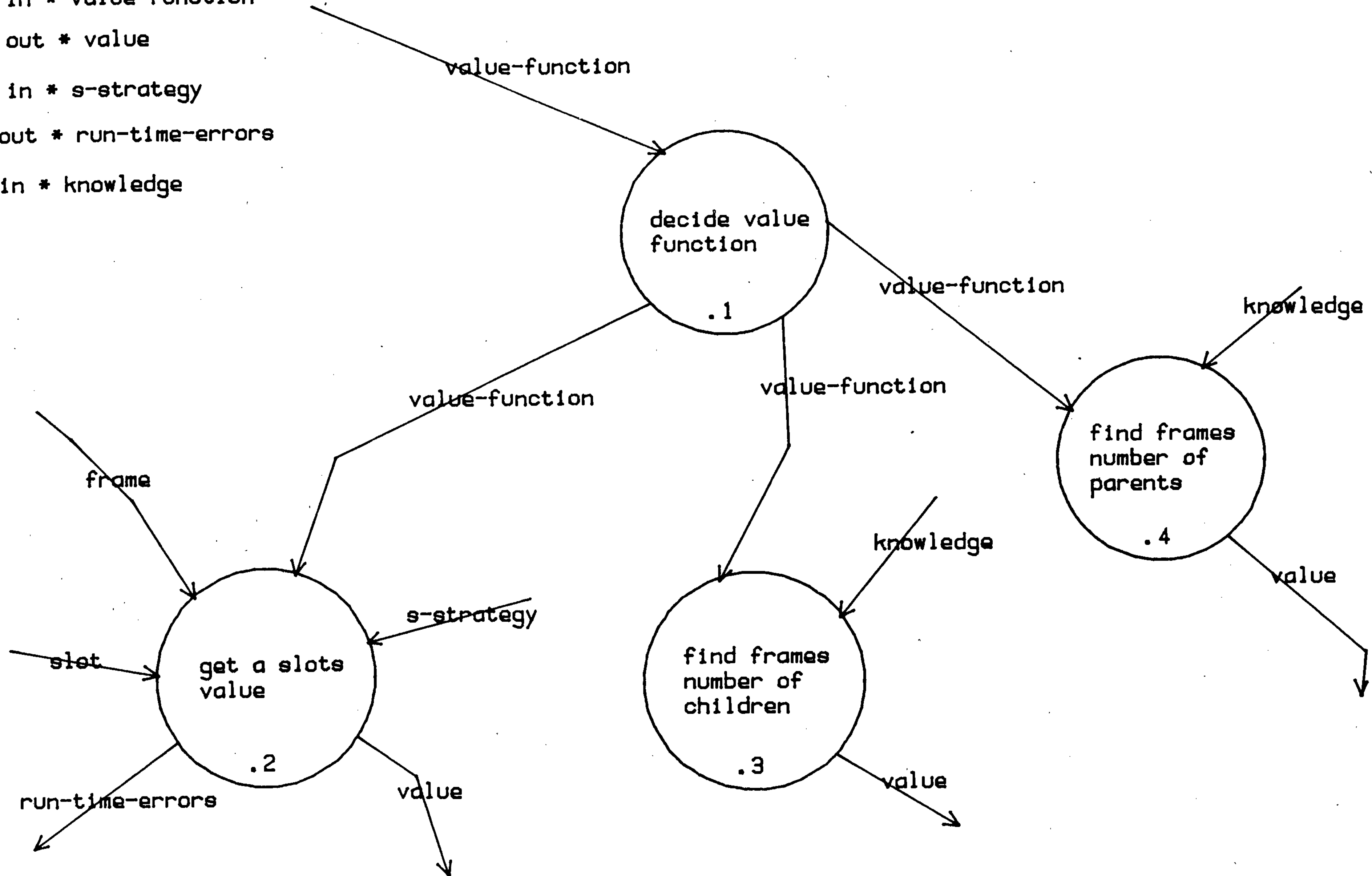
* out * validity-flag

* in * knowledge

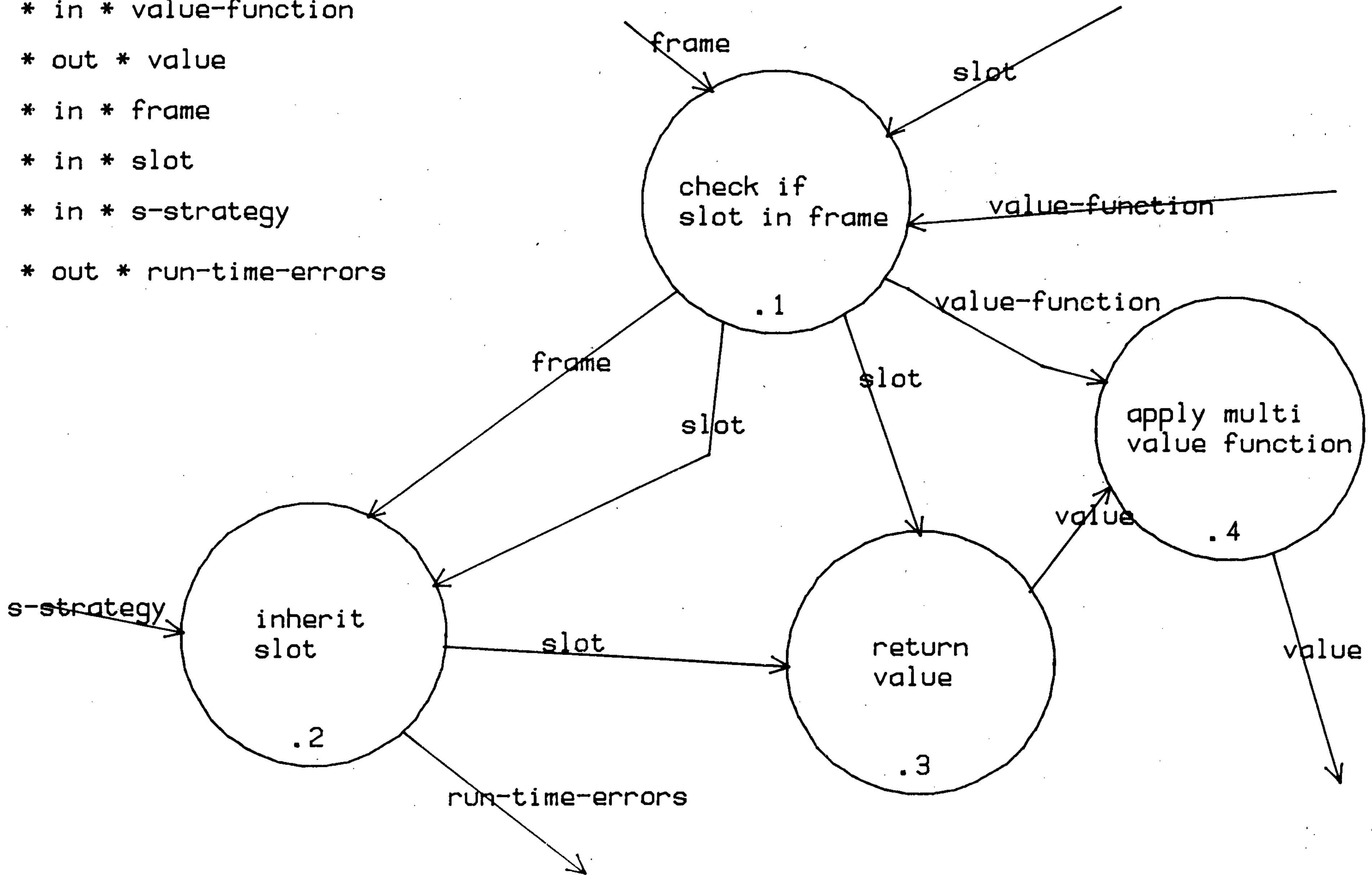
* out * run-time-errors



3.3.4.2.2;5
get values from knowledge base



3.3.4.2.2.2;5
get a slots value



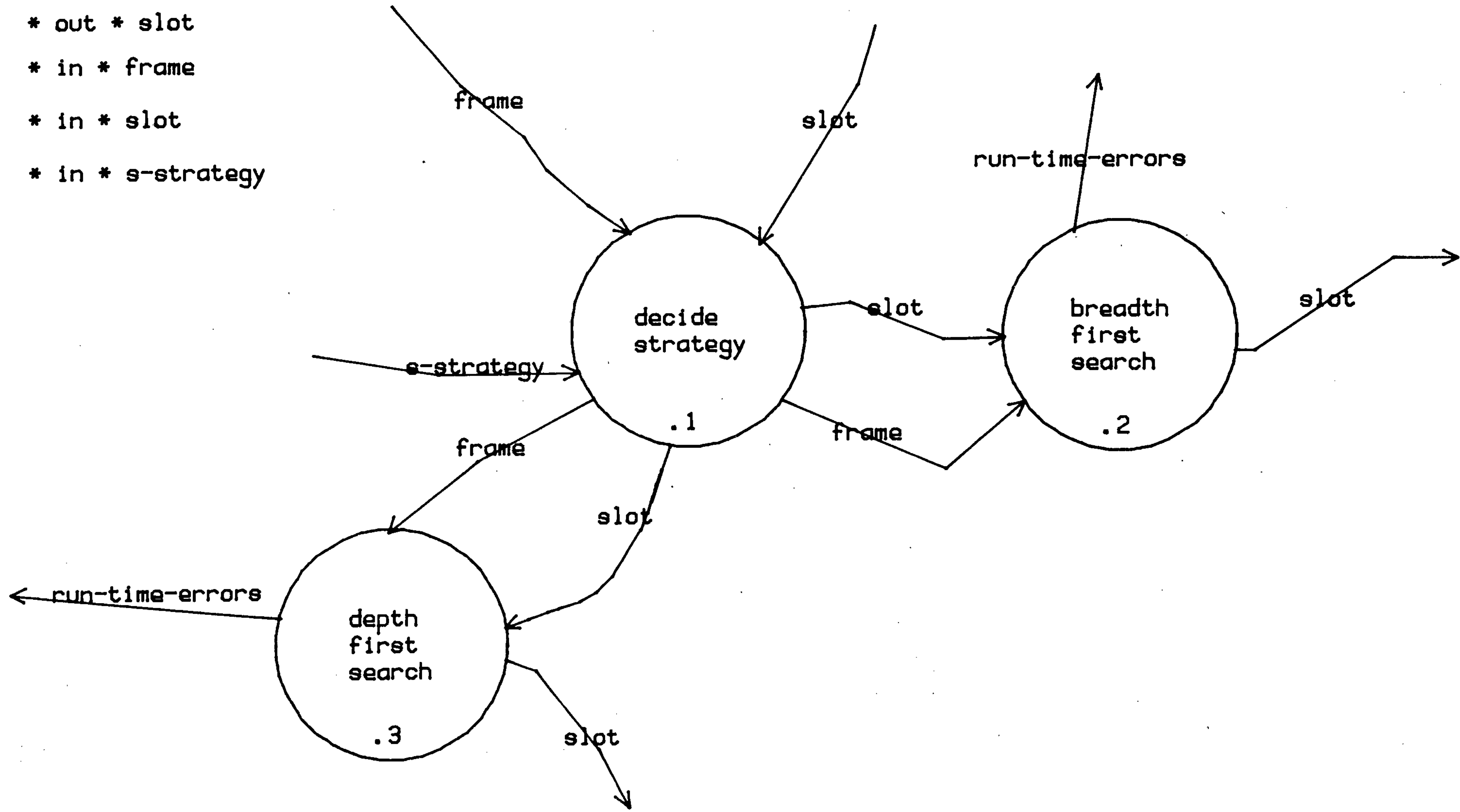
3.3.4.2.2.2.2;5
inherit slot

* out * slot

* in * frame

* in * slot

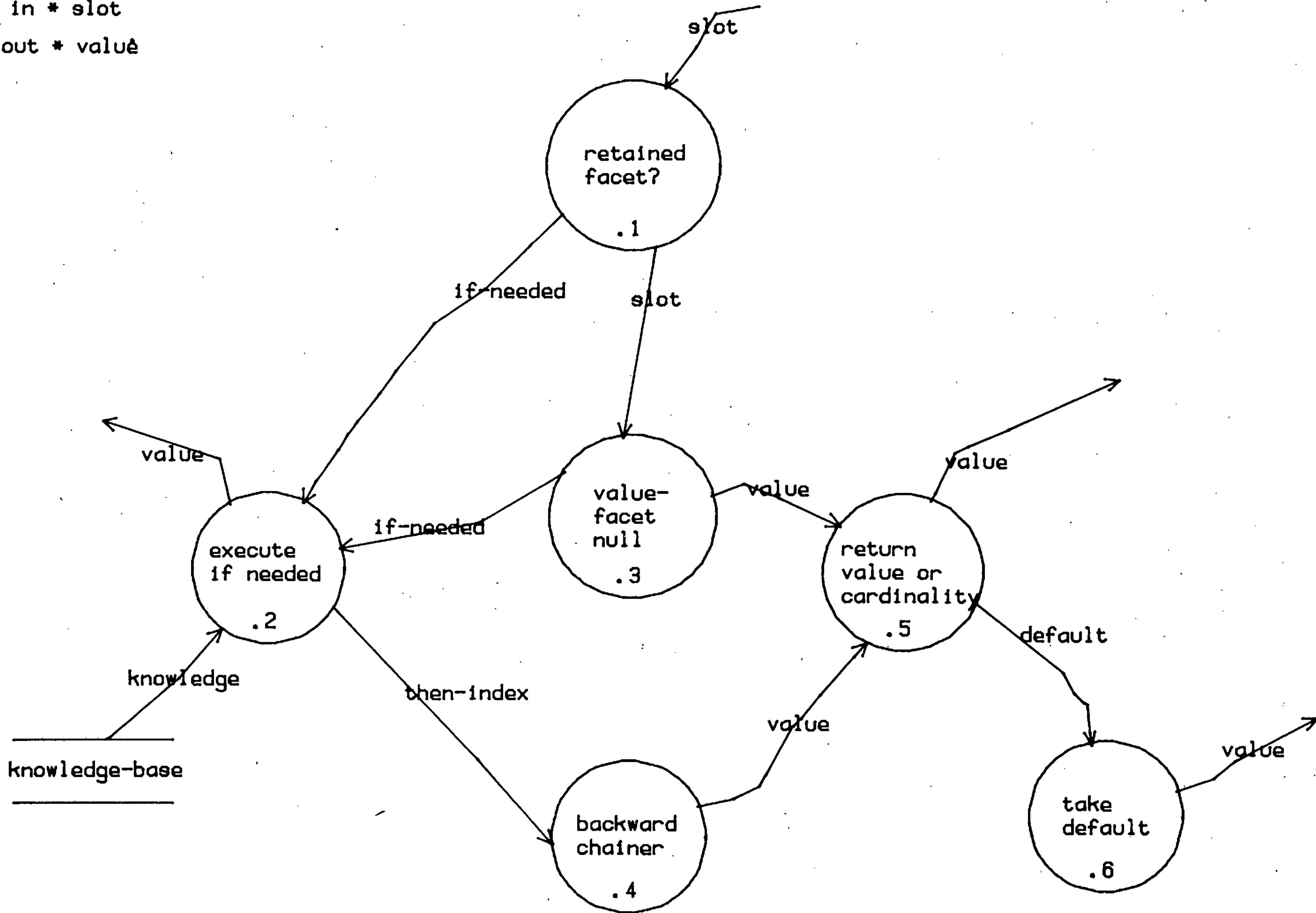
* in * s-strategy



3.3.4.2.2.2.3.8
return value

* in * slot

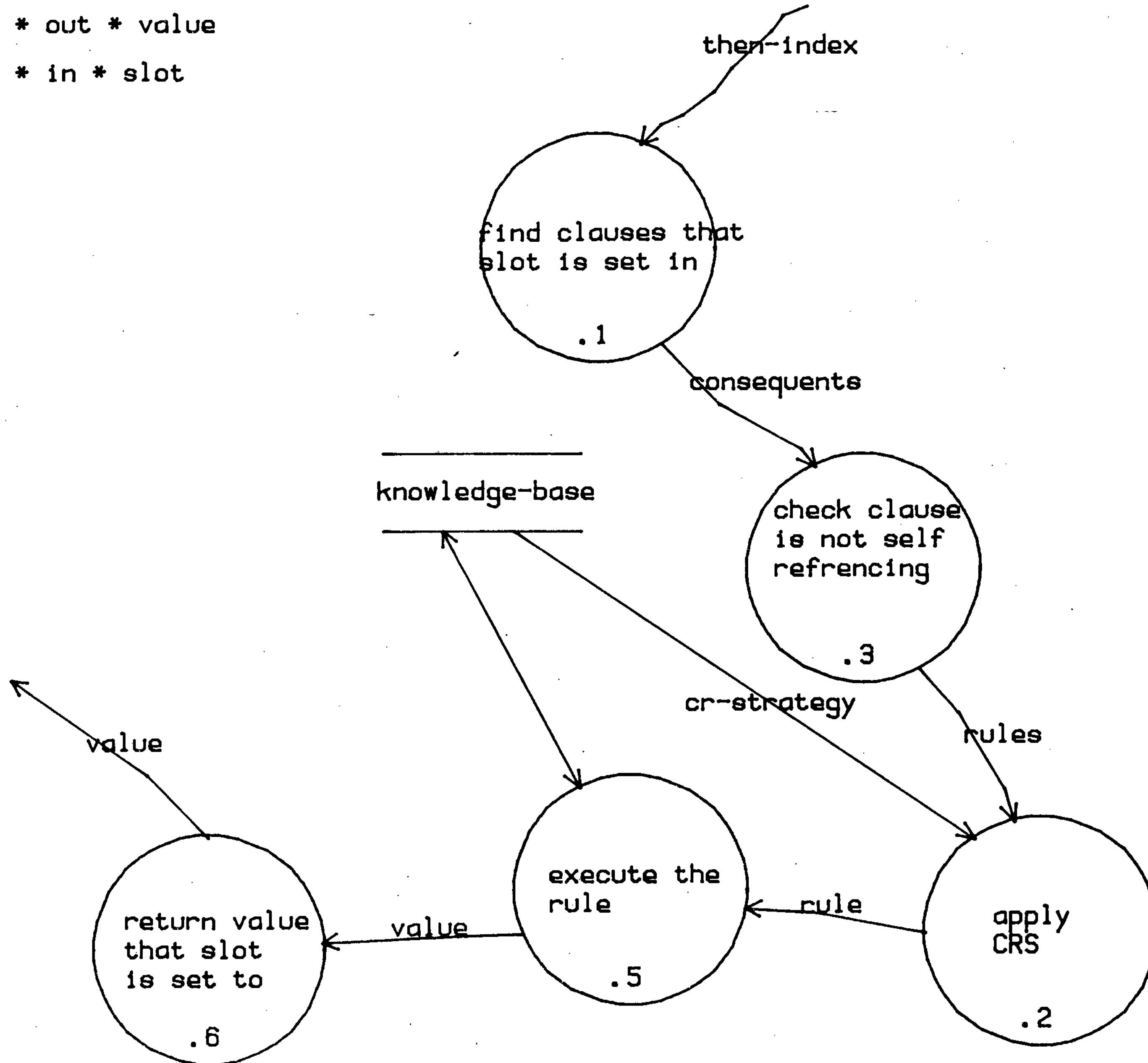
* out * value



3.3.4.2.2.2.3.4;7
backward chainer

* out * value

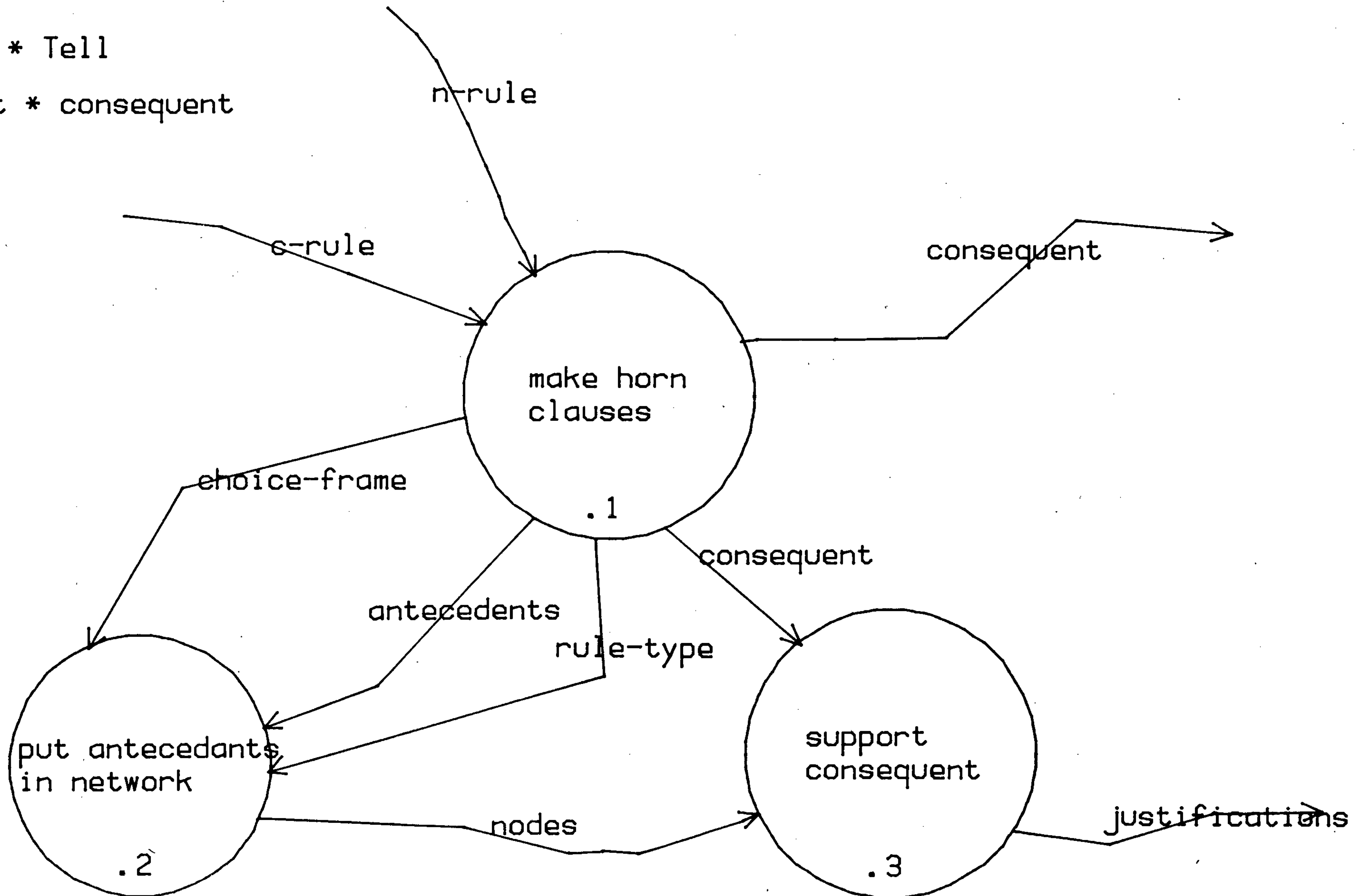
* in * slot

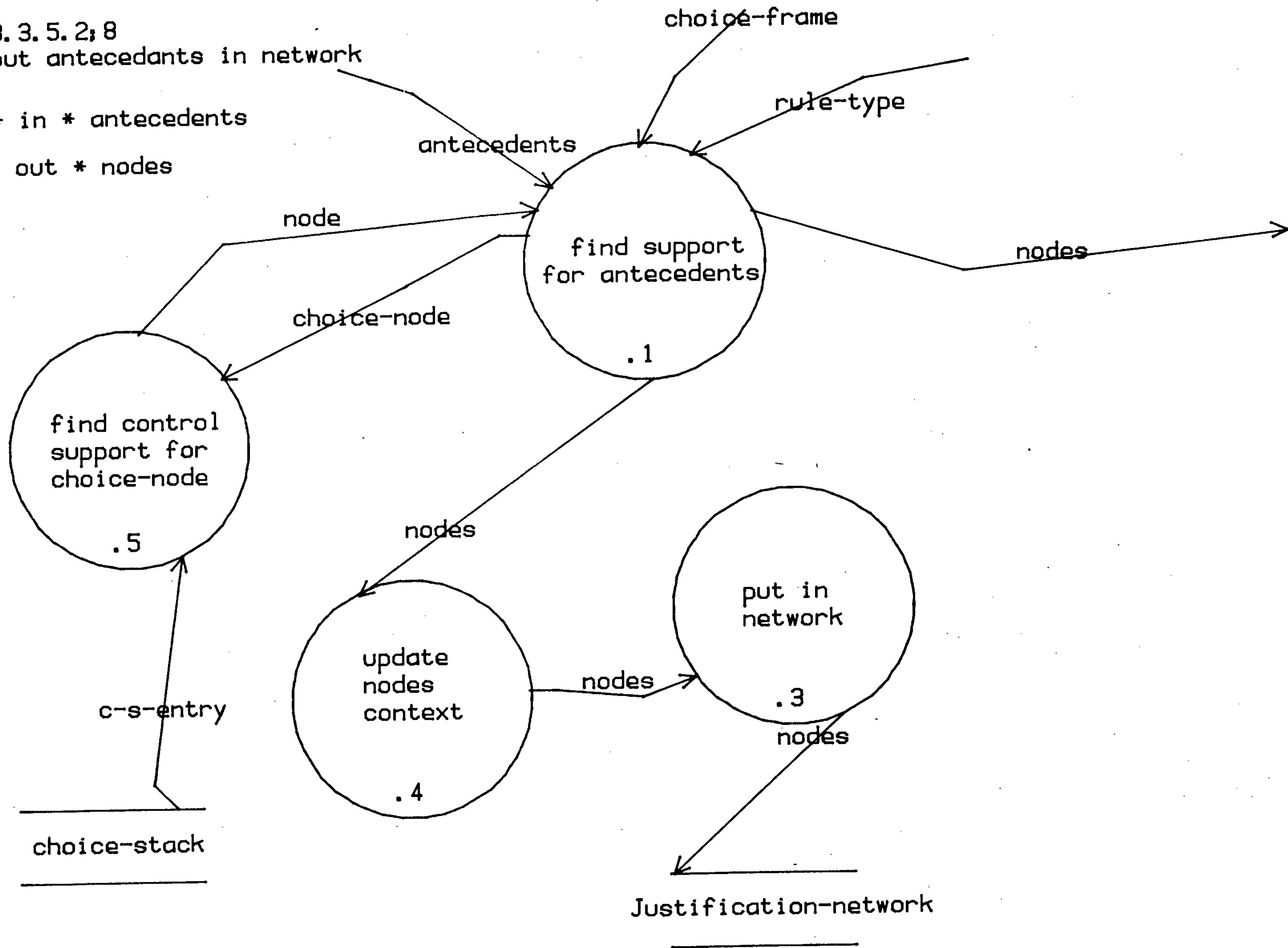


3.3.5; 12
Add to TMS

* in * Tell

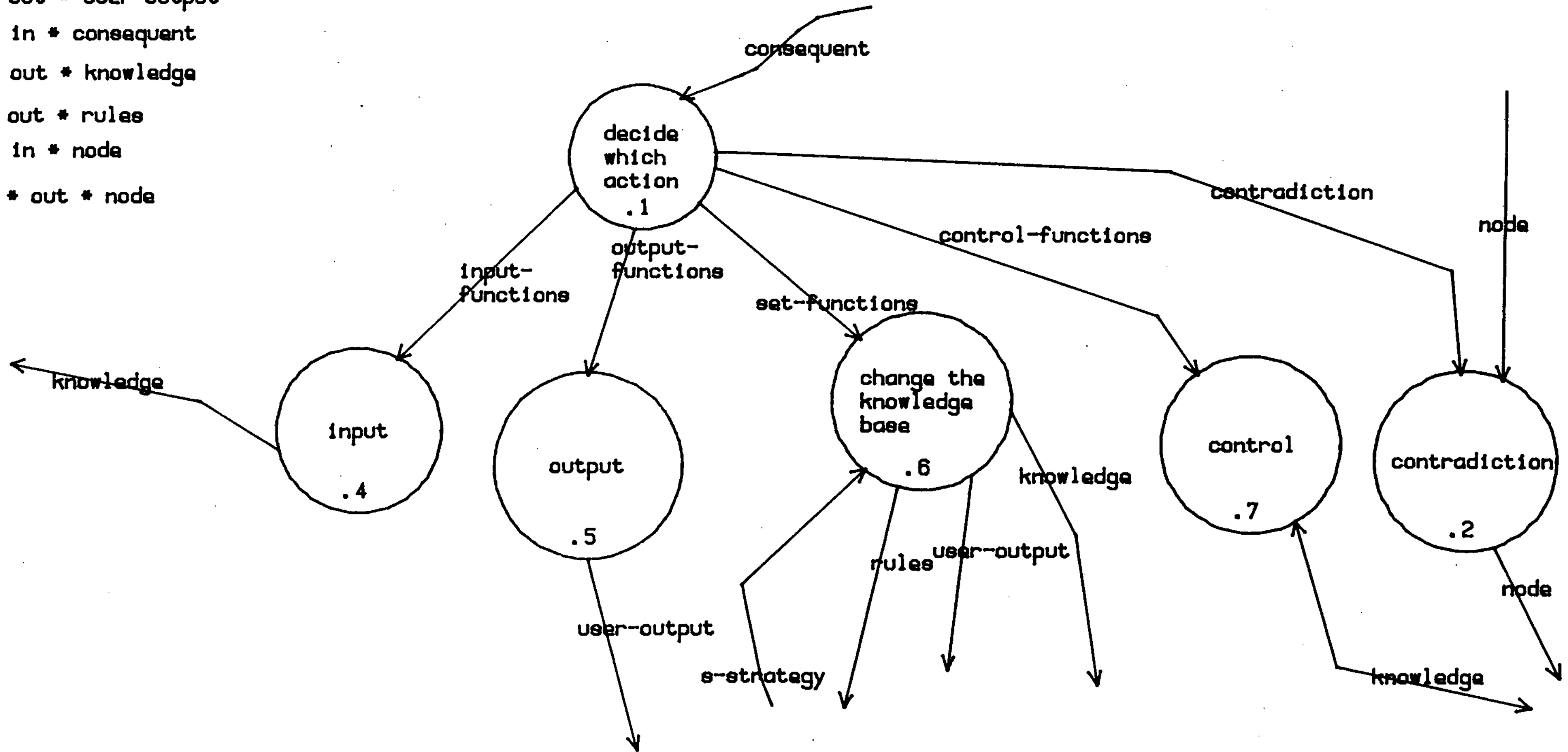
* out * consequent




3.3.5.2;8
put antecedants in network
* in * antecedents
* out * nodes
choice-frame
rule-type
antecedents
node
find support for antecedents
.1
nodes
choice-node
find control support for choice-node
.5
nodes
put in network
.3
update nodes context
.4
nodes
c-s-entry
nodes
choice-stack
Justification-network

Execute consequent and add to relevancy lis

* out * user-output
* in * consequent
* out * knowledge
* out * rules
* in * node
* out * node

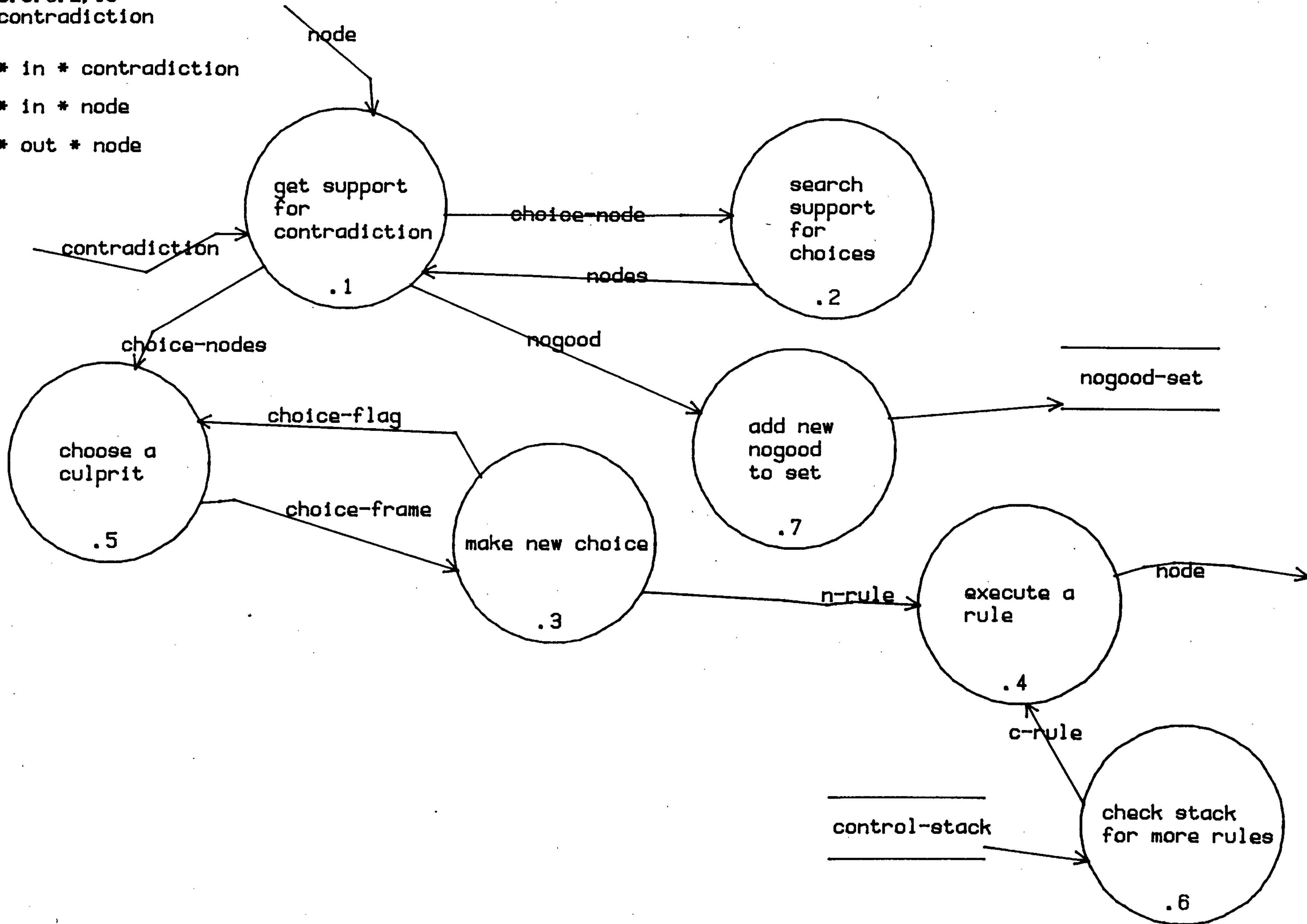


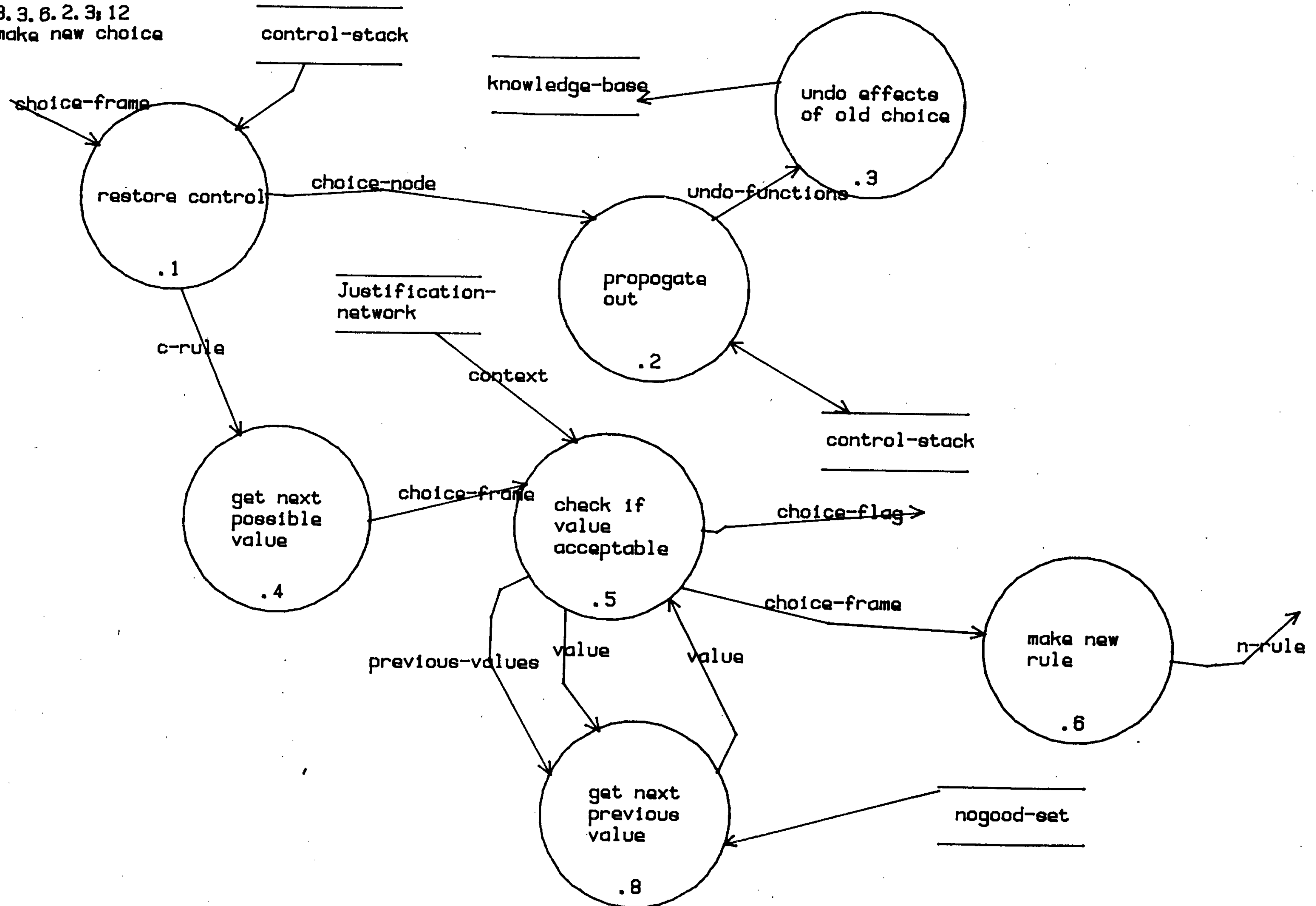
3.3.6.2;10
contradiction

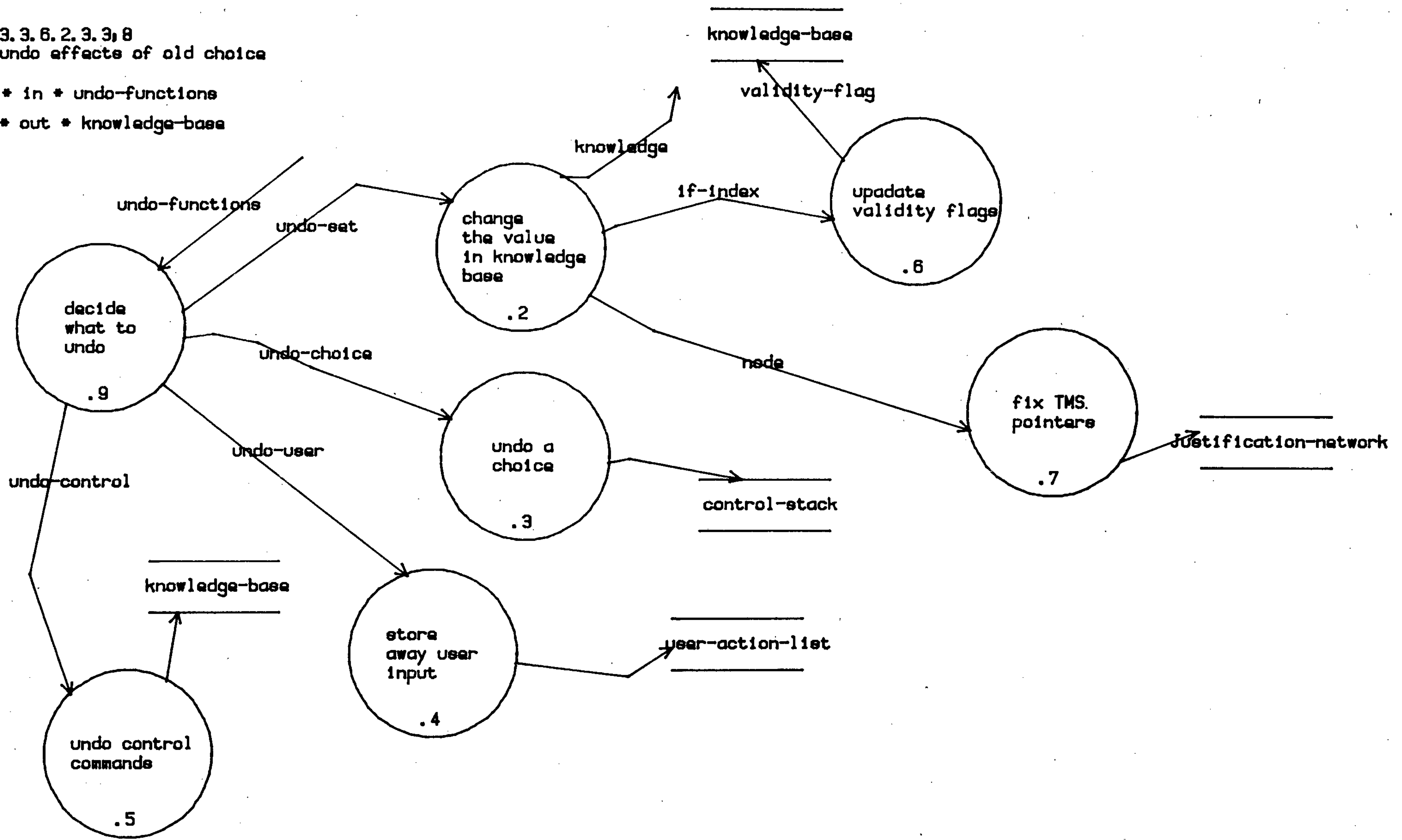
* in * contradiction

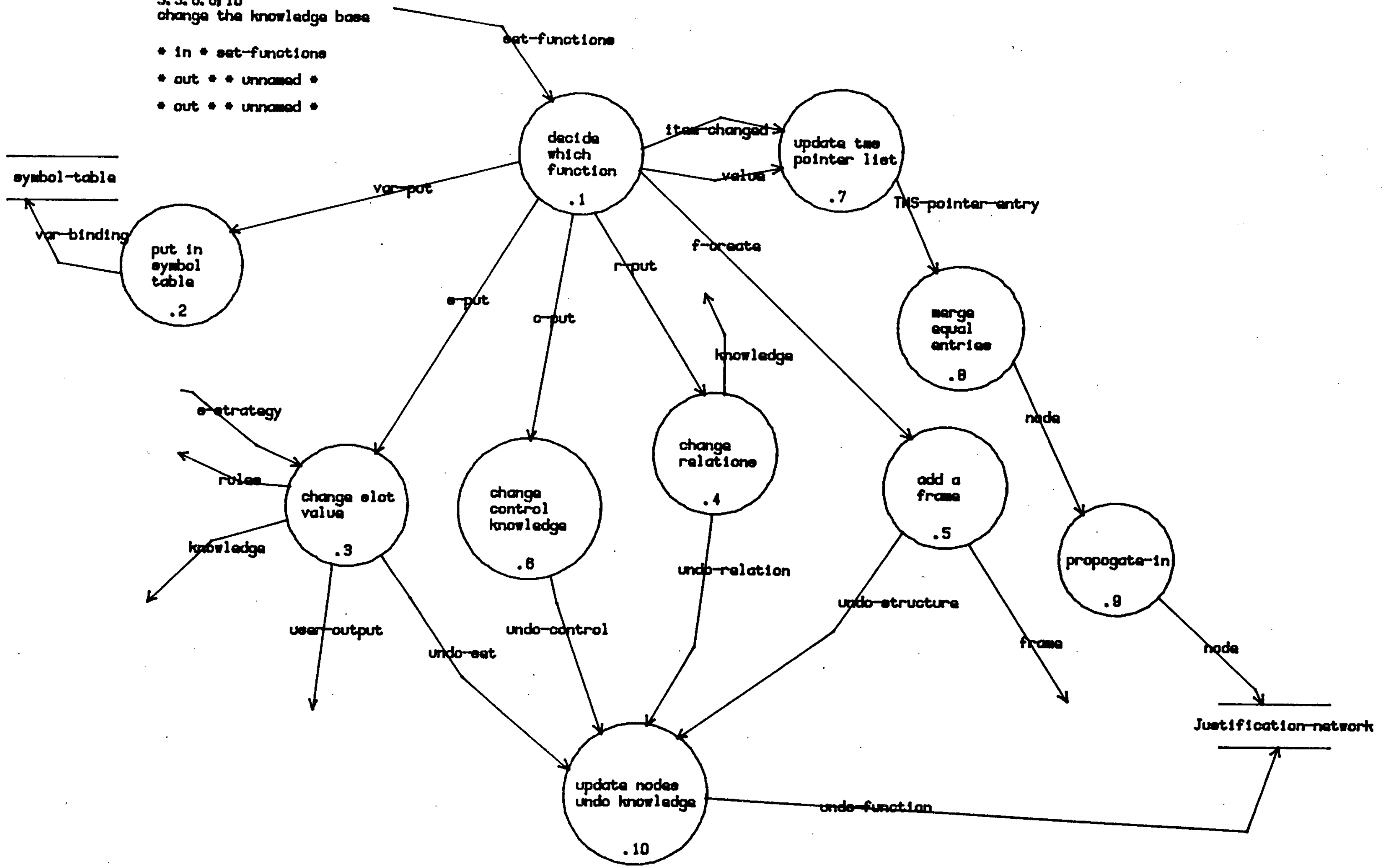
* in * node

* out * node




3.3.6.2.3.12
make new choice
control-stack
knowledge-base
undo effects of old choice
.3
choice-frame
restore control
.1
choice-node
undo-functions
propogate out
.2
Justification-network
c-rule
context
control-stack
get next possible value
.4
choice-frame
check if value acceptable
.5
choice-flag
choice-frame
make new rule
.6
n-rule
previous-values
value
value
get next previous value
.8
nogood-set


3.3.6.2.3.3.8
undo effects of old choice
* in * undo-functions
* out * knowledge-base
knowledge-base
validity-flag
knowledge
undo-functions
undo-set
change the value in knowledge base
.2
if-index
upadate validity flags
.6
decide what to undo
.9
undo-choice
node
fix TMS. pointers
.7
justification-network
undo-user
undo a choice
.3
control-stack
undo-control
knowledge-base
store away user input
.4
user-action-list
undo control commands
.5


3.3.6.6;10
change the knowledge base
* in * set-functions
* out * * unnamed *
* out * * unnamed *
set-functions
decide which function .1
symbol-table
var-put
var-binding
put in symbol table .2
item-changed
value
update tms pointer list .7
TMS-pointer-entry
merge equal entries .8
f-create
r-put
s-put
c-put
knowledge
s-strategy
roles
change slot value .3
knowledge
user-output
undo-set
change control knowledge .6
undo-control
change relations .4
undo-relation
add a frame .5
undo-structure
frame
node
propogate-in .9
node
Justification-network
update nodes undo knowledge .10
undo-function

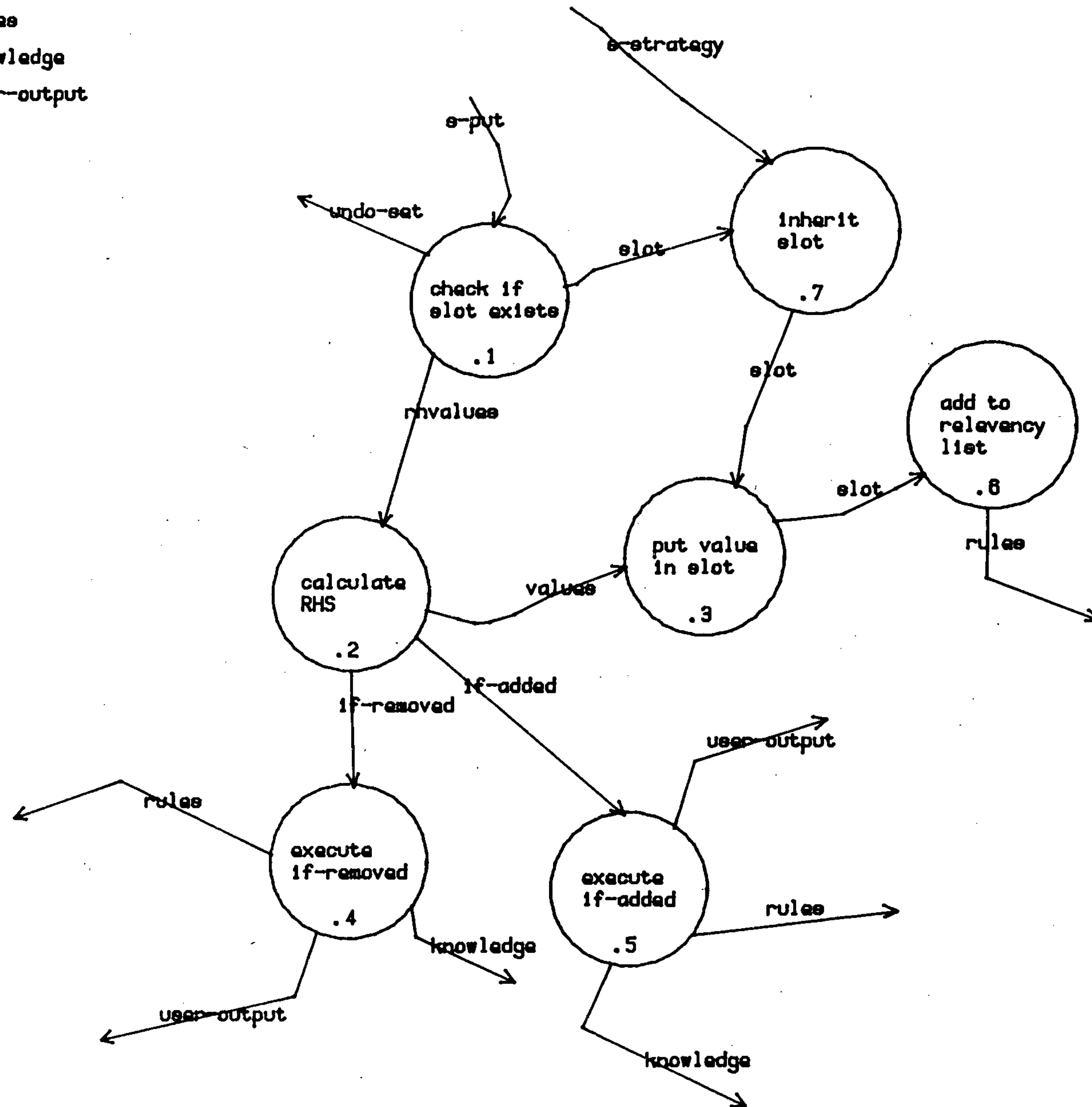
3.3.6.6.3;6
change slot value

* in * s-put

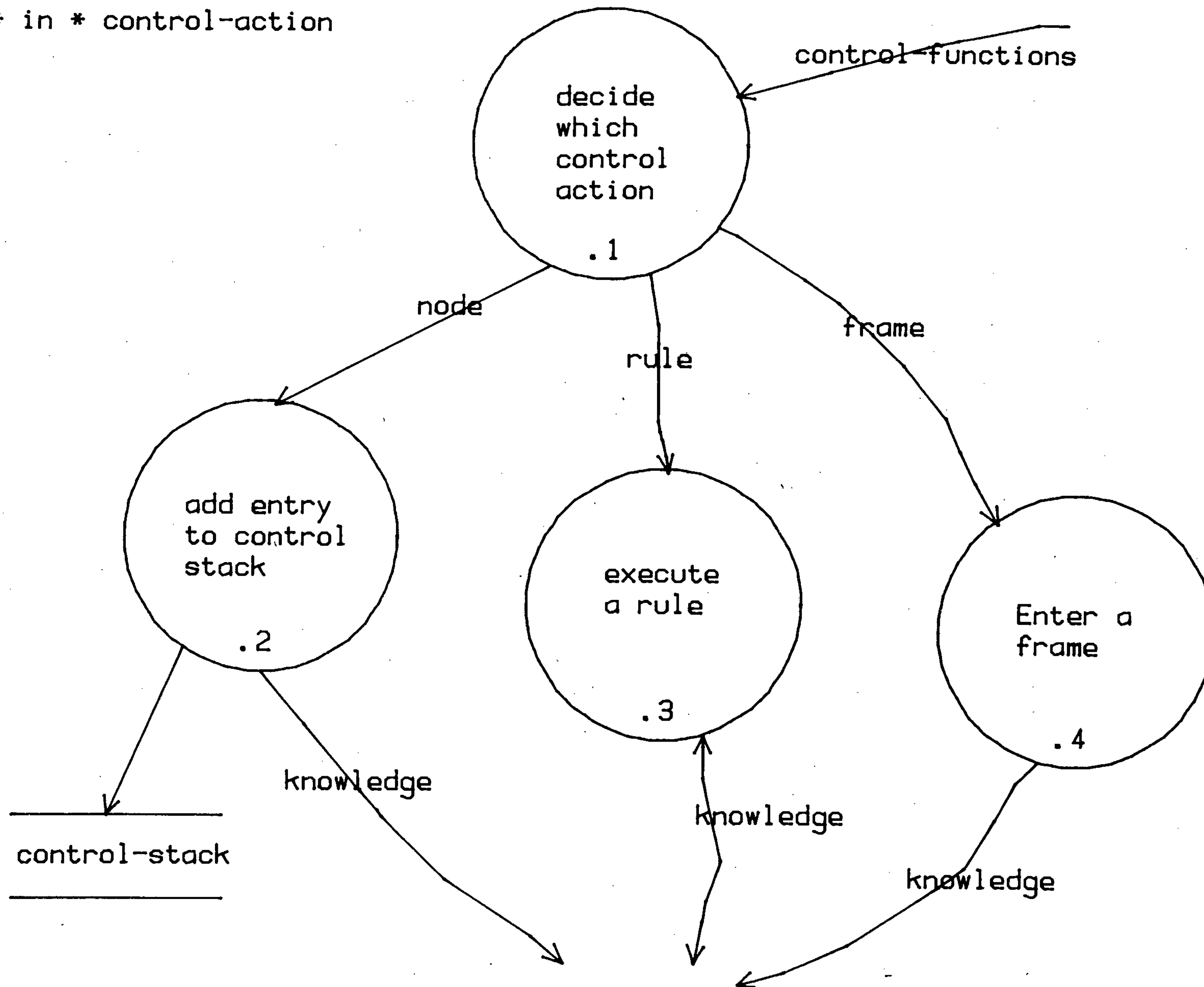
* out * rules

* out * knowledge

* out * user-output

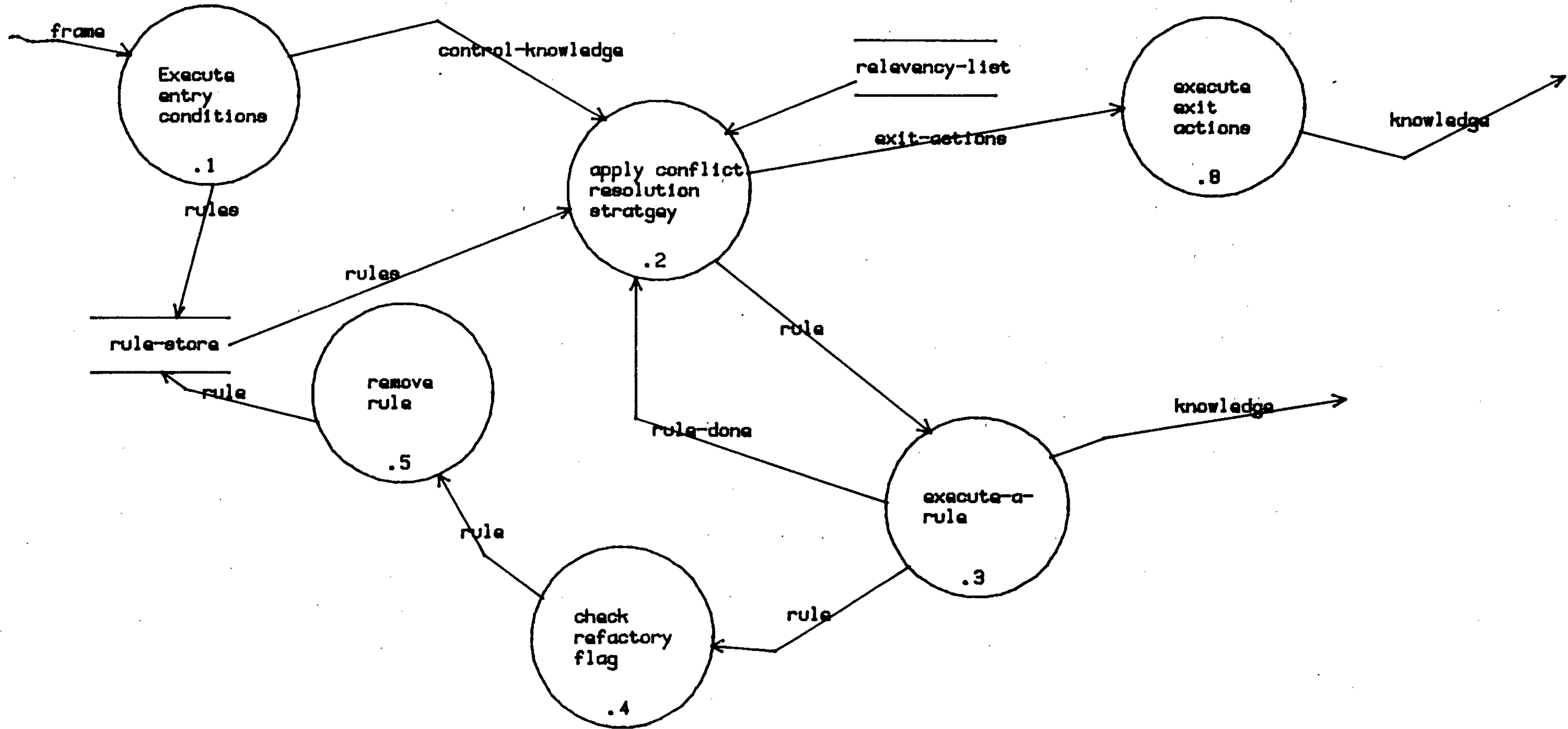


3.3.6.7; 4
control



3.3.6.7.4:5
Enter a frame

* in * frame



272

22 NOV 1988