Thes 1005

AN OBJECT-CENTRED INTERPRETATION OF THE BLACKBOARD ARCHITECTURE FOR KNOWLEDGE-BASED PROGRAMMING

by

David McArthur

Submitted to the University of Stirling
in partial fulfillment of requirements for
the degree of Doctor of Philosophy

ABSTRACT

Progress with knowledge-based approaches to the production of reliable software has been slow but intense over the last decade. From individual efforts, environments have emerged to support these investigations with tools for knowledge representation and search. However, this support technology threatens to be overtaken as investigators, largely inspired by cognitive models, prefer to work with multiple representations of systems.

The blackboard architecture offers a number of principles for reasoning amongst many representations of systems and might provide a framework in which knowledge-based programming could be supported. A suitable interpretation of the architecture must offer transparency to complex reasoning processes and include a high-level language for describing knowledge. In short the architecture must be made accessible for applications in the domain of programming.

This thesis describes an object-centred interpretation of the blackboard architecture for knowledge-based programming. Solutions are developed on blackboards : described as assemblages of objects. Programming knowledge for particular applications and strategies for applying that knowledge are represented as rules in a high-level language known as APPEAL. Rules are themselves objects and occupy the blackboard. This interpretation, embodied in the ENCORES system, brings transparency with a previously unattainable flexibility for engineering programming knowledge.

ACKNOWLEDGEMENTS

My thanks to:

Chic Rattray, without whose encouragement, ideas, criticism and enthusiasm this work would not have been started, continued or finished.

Muffy Thomas and Jean McInnes for providing a dynamic research team in which to generate ideas (both on and off blackboards).

The University of Stirling, who provided both the funding and opportunity to pursue this research.

The Department of Computer Science for providing the machine resources and printing facilities on which this thesis was produced.

The Alvey Directorate, and Tony Dignan in particular, for supporting me financially during the latter period of this research.

My mother, who urged me on with this work when financial support ran out.

My friends in Stockbridge, Edinburgh, especially Pete Sykes, Dave Patterson, Dave Pack and Allan Ferguson who provided alternative intellectual stimulation in the evenings over many a pint of Belhaven.

My flat-mates Dave Thomas, Tim Rowsell, Mike Hall, Steve Jackson and Farren Gainer to whom I owe many excellent parties, Sunday feasts and teabags.

CONTENTS

CONTENTS

CONTENTS

LIST OF FIGURES

CHAPTER 1.
INTRODUCTION.

Programs are surely one of the most complex artifacts of human reasoning. As static objects they are no more than logical expressions : in execution they simulate our mental model of the problem domain. Yet they often fail us. Our models and programming knowledge are seldom to blame. Rather, as programmers, we do not have the mental apparatus to manage the complexity of large systems [Parnas85a]. Programs grow to become Frankensteins which we no longer understand, whose details we forget and which disobey us at will.

Calls for a more scientific or engineering-based approach (eg. [Naur&Randell69],[Buxton&Randell70]) led to the emergence of "Software Engineering" as a discipline which includes methods to structure software production [Sommerville82]. Our models of the problem domain may be recorded succinctly using high-level representations and many methods may be employed to refine these to programs. However, formal methods of establishing program correctness are in their infancy [Parnas85b]. Only experience of building similar systems and extensive testing [Parnas85c] give confidence our software products will work most of the time. The quest for reliability must take new directions if demands for complex, reliable systems to be applied in areas such as business, defence and medicine are to be met.

Mechanisation of software production is seen as one way forward [Balzer73]. The program knowledge we employ successfully at the small scale might be employed mechanically to construct large scale systems if we can capture and harness this knowledge.

Techniques of knowledge-based systems (eg. those described in [Hayes-Roth etal83]) may offer a way forward towards mechanisation. However, a few observations on the domain of programming should give us reason to pause before rushing to engineer this knowledge. The techniques normally find application in domains where knowledge is ill-structured and implicit [Newall69]. The programming domain contains a large body of explicit knowledge which is richly structured. As programmers we bring well rehearsed methods to bear on problems using a variety of abstract representations and notations. This organisation should be captured in the automation of software production.

This thesis describes an attempt to design and implement tools for representing programming knowledge within a particular organisation known as the blackboard architecture. It will be shown that a suitable interpretation of the architecture provides an organisation well suited to the domain of programming : many abstract and concrete descriptions of software systems may coexist and programming knowledge may be organised around these.

In this chapter, we describe knowledge-based programming, introduce the blackboard architecture and reveal its potential for applications in software development.

1.1. Knowledge-based Programming.

The early successes with intelligent knowledge-based systems (IKBS) coincided with the realisation that large software systems could not be built reliably (the so called 'software crisis'). Therefore efforts to apply IKBS techniques for the automation of software production, partial or otherwise, have been intense amongst the research community.

Concerned with the representation and use of specialised problem-solving expertise, IKBS typically find solutions by a process of "search". The specific problem and knowledge base together define a hypothetical "search tree" over which this search is conducted [Nilsson80].

The aim of most knowledge-based approaches to programming has been to support the synthesis of efficient representations of formal specifications: high-level formalisms in which software systems can be specified abstractly [Liskov&Zilles75]. Some of these efforts will be described along with their chosen specification technique. Given the limited success of early work in this field, the most persistent of these efforts have given more attention to knowledge representation issues leading to the emergence of knowledge-based programming environments.

Thus a new generation of software tools which encode knowledge have emerged. Such tools are still mostly at the research stage, have limited applicability and are reported to have various degrees of success. The outlook however is optimistic. They fall roughly into four categories which we will call transformation systems, tools based on formal logics, tools supporting the conventional software life-cycle and knowledge-based programming environments.

1.1.1 Transformation Systems.

The rule representation of knowledge has become popular amongst the wider IKBS community and soon found application in the program domain. It has been suggested that programs may be generated from formal specifications by a sequence of rules, each of which are provably correct. This approach has been termed 'inferential programming' [Scherlis&Scott83]. Without regard to their correctness, a more modest goal is to find the rules then formalise them later. Lenat [Lenat82] has pointed out that knowledge made explicit for use in a knowledge-based system can be studied and formalised to new science.

Typical of the rule-based approach are systems in which the rules are wholely transformations (ie. each rule takes one program fragment and replaces it with another). The hope is that rules constrained to this form may be proved valid if the specification language has a formal semantics.

Early work on transformations demonstrated that functional programs could be optimised by selective application of transformations [Burstall&Darlington77]. The TI and APE systems are typical of the efforts to transform specifications to efficient representations so our attention will be confined to these.

The Transformational Implementation (TI) approach to programming [Balzer81a] involves refining and optimising an operational specification of a system written in a high-level language named CIST [Balzer81b]. The operational specification technique defines an application in terms of an initial configuration of objects, their relationships and any constraints on their behaviour.

Transformation rules are applied in a semi-automatic fashion in that the programmer must choose the transformation to apply and the context in which to apply it. Earlier contexts in the development sequence can be chosen from which point the programmer can explore various alternative lines of reasoning.

An interesting result from the investigations with TI is that transformations tend to fall into two categories according to their effect on the program. So called 'high-level' transformations result in a large refinement or optimisation of the program. Examples are changing a control structure from iterative to recursive or merging a number of loops into one. In contrast, 'low-level' transformations are used to massage the program to meet the specialised applicability conditions of high-level transformations. Examples of low-level transformations are commuting two adjacent statements or unfolding nested blocks. Consequently, these low-level transformations were gathered into a separate subsystem known as a 'jittering' system [Fickas80] which would apply them automatically.

The designers of the Automatic Programming Expert (APE) [Bartel etal81] made no pretence at using rules which were provably correct. However, the goal with this system was more ambitious in that algebraic specification of systems were to form the input and transformations were to be chosen and applied automatically for the production of LISP programs.

The algebraic specification technique defines the properties of operations by giving equations relating them, as in the ubiquitous stack example:

pop(new-stack) = error

pop(push(element,stack)) = element

where new-stack and push are the "constructor" operations for the stack type, and are considered primitive. In practice, other operations are defined in terms of the constructors.

APE takes such specifications as input, parses them into an internal LISP form and then applies production rules in a number of phases to produce LISP programs which might satisfy the specifications. The 'functionality' and 'axiom' phases apply rules which infer relevant properties of the operations from their signatures and equations. Relevant properties are the 'action' the operation appears to make on the abstract object; any 'selection criteria' used to access elements from abstract objects; and whether the operation is recursive. The 'representation' phase applies rules which infer a suitable representation for the abstract object based on the operation properties. Given this choice of representation the 'compile' phase implements each operation of the abstract object in terms of those of the chosen representation. A 'clean-up' and 'LISP-phase' remove inefficiencies and integrate the resulting LISP code.

A large number of rules are used in APE, some of which are tactical i.e. serve to control the use of other rules, presumably ordering their use in the phases described above. All APE rules, like those of TI, are chunks of LISP code, so the knowledge is only explicit to the knowledge engineers. Not only is the knowledge in a poor form to be formalised, but the addition of extra rules presents a problem : can the engineers recall the behaviour of all existing rules without reading their code? Clearly, high-level knowledge representation languages are necessary for the painless acquisition of programming knowledge within knowledge bases.

Many more transformation systems have been developed over the last decade. A general survey of these will be found in [Partsch&Steinbruggen83].

1.1.2 Tools based on Systems of Logics.

Another common approach to automating the programming of specifications is to attempt to generate executable code within some formal reasoning system. Knowledge is represented as theorems (rather than rules) and verification of code to specification is assured.

This approach has a long history. Green [Green69] attempted to develop programs using predicate calculus. Manna and Waldinger [Manna&Waldinger80] attempted to combine transformation rules with theorem proving in a program synthesis system based also on predicate calculus.

Constructive type theory is now showing promise [Hamilton85] for this purpose but as yet no formal approach to program synthesis has received widespread acceptance. A difficulty of this approach is that unless strategic knowledge also finds a representation and use within the system then the search space tends to be quite large.

1.1.3 Tools supporting the Conventional Lifecycle.

Knowledge-based techniques are also finding application to support stages of the conventional software life-cycle. Intelligent software tools have been designed to aid the stages of requirements definition, design, implementation, debugging, testing and maintenance. The most exciting of these systems employ an abstract representation for programs known as 'plans' [Rich81] which originated from a long-standing effort to develop a knowledge-based support environment known as the Programmer's Apprentice (PA) project [Rich etal79].

Programs can be represented as plans, described as stereotypical program fragments, each of which fulfills an identifiable role in the wider context of code in which it is found. Figure 1.1 illustrates how a PASCAL program (in this case to read in numbers, ending with '99999', then output their average) can be considered as consisting of four plans. A 'counter' plan maintains a count within a loop, a 'running total variable' plan records an accumulation, a 'running total loop' plan inputs and sums a sequence of numbers and 'valid result skip guard' outputs a valid value only. Notice that the code fragments describing plans tend to interlace (eg. with statement 'Sum:=Sum+New').

A knowledge-based program editor known as KBEmacs [Waters85] is a major component of the Programmer's Apprentice project. The system builder can stipulate, in an interactive manner, which plans should be used to construct a program. The reusability of programs in terms of plans has been investigated by Parker [Parker&Hendley87].

```
                    PROGRAM Average( INPUT, OUTPUT);
                    VAR Sum, Count, New, Avg : REAL;
Counter             BEGIN
Plan     ---------->     Count:=0;
         |   ----->      Sum:=0;           Running Total Loop Plan
         |   |           READ( New);  <--------------------
Running  |   |           WHILE New <> 99999 DO  <---------|
Total    |   |           BEGIN                            |
Variable |   ------------>    Sum:=Sum + New; <---------|
Plan     ---------------->    Count := Count + 1;       |
                              READ( New);  <------------
                         END;                     Valid Result Skip Guard
                         If Count > 0 THEN BEGIN  <-----------
                                Avg := Sum/Count;  <-------|
                                WRITELN( Avg);  <----------|
                         END ELSE  <-----------------------|
                             WRITELN('no legal inputs'); <--|
                    END.
```

Figure 1.1 Plans within a PASCAL program.

The PROUST system [Johnson&Soloway85][Johnson86] debugs PASCAL programs and suggests corrections using a knowledge base of program plans, common faults and some description of the programming problem in terms of goals which are known to be attained by these plans. The last report of PROUST's success rate was 75% and the system is available commercially. An interesting feature of PROUST is that the knowledge base contains both low-level transformation rules in addition to the plans which do not find as obvious representation as transformations (because of the inter-lacing). Low-level transformations appear to be pervasive in knowledge-based software tools.

Other tools aimed at supporting stages of the conventional software life-cycle are the PIE and SAFE system. The Personal Information Environment (PIE) [Goldstein&Bobrow81] is a code control tool for the management of evolving software designs. PIE makes use of database systems, explored in artificial intelligence (AI) research, in which alternative world views may be represented. (A prototype of one such database system will be described in chapter 5). SAFE [Balzer etal75] was an ambitious attempt to construct a system which would produce formal specifications, in the GIST language, from informal specifications of a system. Success was very limited, however, investigations with SAFE revealed the manner in which constraints for a system become dispersed when it is specified in an informal notation. A system is reported which uses a knowledge base to devise tests then locates and corrects bugs automatically [Dershowitz&Lee87].

1.1.4 Knowledge-based Programming Environments.

Broadly speaking, the development of expert system "shells" and knowledge-based programming environments have followed the same evolutionary path. Confidence gained with the construction of a knowledge-based system often prompts its creator to abstract the control and knowledge representation scheme within a system building tool. Further intelligent systems can then be built with these by the acquisition of knowledge from related domains.

A number of knowledge-based programming environments have been constructed to serve as convenient test-beds for building and testing intelligent tools. These systems often provide a high-level language for the acquisition of programming knowledge. In-built search procedures are available for generating the space of alternative programs. An easy-to-use language for expressing programming knowledge and in-built search procedures facilitate rapid development of intelligent tools.

Three such environments are known to the author. The CHI system [Smith etal85] was developed at the Kestrel Institute, California. CAKE [Rich85] is the latest product from the PA project. The POPART system [Wile82a] is supporting transformational programming at Information Sciences Institute, also in California. CHI and CAKE will be described here; aspects of Wile's work will be described in later chapters.

The main feature of CHI is the provision of a "wide spectrum" language, known as V. The V language includes many of the constructs of both set-orientated specification languages and structured languages. Also included is pattern-matching constructs which allow the user to enter transformation rules in the knowledge base of CHI. The idea is that both rules and specifications of systems (in terms of sets) can be compiled to efficient forms within a single language.

Of particular interest is the sub-language of V (known as VRL) which permits the description of transformations. An example of a transformation which will merge two adjacent conditions is;

```
rule: 'if @A then @C ; if @B then @C'
        -> 'if @A or @B then @C'
```

where symbols preceded with '@' are pattern-match variables and patterns are based on the syntax of the V language. Predicates may be included within the rule to indicate constraints on the sorts that variables may match with (eg. 'procedure(@C)'). Such predicates are processed using a description of the V language which resides in the knowledge base of CHI.

Wide spectrum languages provide a single representation in which systems can be developed. However, this approach has its critics. Wile has pointed out (in [Lindsey86]) that "....if you expect a semantics which will ascribe a meaning to a program containing any arbitrary mixture of these constructs, and transformation rules which will operate between any pair of them, then you expect too much. Such a language is simply not designable." He goes on to suggest use of a multi-level approach, with each level having its own "local formalism" (ie. well delimited syntax and semantics) and inter-formalism transformations to provide the mapping of programs between adjacent levels.

CHI permits search through the space of transformed specifications and includes a context mechanism to record alternative refinements. The system supports Smith's research on algorithm design [Smith85], and the construction of tools for project management and communication support [Kedzierski83], and synthesis of efficient structures for concurrent computation [King84].

CAKE is described as an multi-layer environment which provides reasoning facilities for the Programmer's Apprentice. Eight levels of representation are employed, each with their own reasoning procedures. The bottom five layers of CAKE use a predicate calculus representation of program features. CAKE has been constructed to support the analyses, synthesis and verification of programs for the PA, mainly using plans.

The overall reasoning task is partitioned among a number of specialised reasoning components. Several distinct categories of reasoning can be attacked with specialised algorithms : symbolic evaluation (of plans), equality (input/output correspondencies), algebraic properties of operators and type inheritance. Separate layers exist for each of these kinds of reasoning. The author hopes to gain a reduction of the controllable size of the reasoning problem seen by each layer.

The CAKE system could be the first of many tools to provide multi-level architectures for knowledge-based programming. The use of abstract representations, such as plans, and issues of semantic integrity is likely to drive researchers towards multi-level approaches.

Whilst the automatic refinement of high-level specifications would represent considerable progress towards automatic software production other problems with formal specifications persist. They too grow in complexity [Gerhart&Yelowitz76]. The software designer can only ensure that they meet the requirements by replaying them using rewriting or prototyping techniques [Henderson84].

Knowledge-based techniques may find roles in supporting earlier design stages of system construction based on cognitive studies of the design process. The mental processes involved are revealed from protocol analyses of designers reporting their work [Adelson&Soloway85]. Each designer forms a mental model of the design in progress where the model is a representation capable of supporting mental simulations. The model is systematically expanded from abstract levels of representation to more concrete forms.

Analysis of the representations in use shows the meta-plan used by expert designers contains five distinct phases (see figure 1.2): first experts would describe a user view a system, then they stated various assumptions about the implementation, then they developed models of systems at various levels of generality (eg. information flow models, examples of similar systems worked on followed by the specific system at hand); finally the experts worked on the concrete design. The components of the model are developed to the same level of detail in a breadth-wise fashion before the next lower-level of representation is chosen. Notes or diagrams are often kept as each level is worked on and designers use similar design plans for similar problems.

```
Representation:   User...Assump-...Abstract models...concrete...wrap-up
                  model  ions      of system          design
```

Figure 1.2 Meta-plan used by Expert Software Designers.

Knowledge-based support tools can support the design process where unambiguous notations exist or are devised to describe the work in progress (eg. a tool centred on use of the MASCOT design language [Budgeon85]). Designers produce solutions more efficiently if they are acquainted with the problem domain, so we might expect the use of domain specific knowledge in automatic programming tools to support some of the design process(eg. automatic programming in oil well logging [Barstow85]).

Software construction is therefore a highly complex activity of which much is known and much is yet to be discovered. Between the most abstract design model and the final program the construction process is factored along many levels of representation to make the overall task manageable.

1.2 The Blackboard Architecture.

The blackboard architecture [Erman&Lesser75] prescribes an organisation for knowledge-based systems in which solutions to complex problems may be developed at various levels of abstraction and the knowledge factored into a number of knowledge sources which act at these levels. The architecture itself is very abstract but provides a general framework for representing many complex reasoning tasks [Hayes-Roth83].

The "hypothesis and test" approach to problem solving forms the basis of the architecture which consists of a shared data region (called the blackboard), a set of knowledge sources and a control mechanism (see figure 1.3). The blackboard is a data base which is shared by the knowledge sources as their communication medium. Containing rules and hypotheses which express the domain expertise of the system, the knowledge sources respond to each other through observed changes in the blackboard. The control mechanism schedules execution of these knowledge sources.

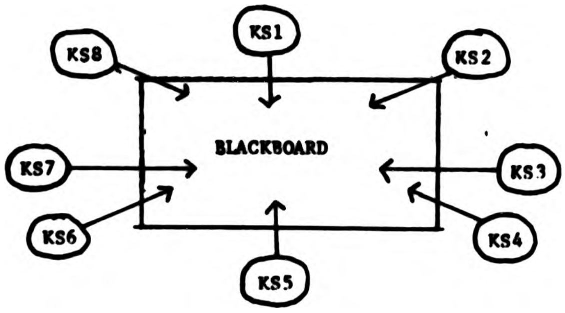


Figure 1.3. Overview of Blackboard Architecture.

Other important features of the blackboard architecture are :

(1) 'Opportunistic' approaches may be taken to problem solving. Knowledge sources can decide whether they can contribute anything to the current problem on the blackboard taking advantage of the prevailing situation.

(2) The blackboard is partitioned into distinct information levels, each consisting of a set of primitive elements for representing the problem at that level. The elements have a fixed set of attributes for all levels of the blackboard. The sequence of levels form a loose hierarchical structure in which the elements at each level can be described approximately as abstractions of elements at the next lower level.

(3) Proper separation of available knowledge according to the representations in use. When we consider knowledge sources to consult with specific representations, we have achieved a levelling in the blackboard system that has advantages in maintaining and understanding the system.

Originating with the Hearsay speech recognition experiment [Lesser etal75], the architecture was adopted in other attempts to interpret noisy data such as sonar signal interpretation in HASP [Nii etal82], protein crystallography in CRYSALIS [Engelmore&Nii77] and scene analysis [Hanson&Riseman79]. Its use has been investigated for multiple-task planning in the OPM system [Hayes-Roth etal79] and its generality exploited in system building tools such as HEARSAY-III [Erman etal82], AGE [Nii&Aiello79] and ART [Williams84].

Hayes-Roth [Hayes-Roth79] has suggested that more elaborate blackboard structures based on a number of abstraction hierarchies (known as 'planes') may be necessary for some applications in planning. Craig [Craig86] has investigated the dynamic generation of such planes in his ARIADNE-1 system.

Begg [Begg84] suggests that the architecture is ideal for automating VLSI design; a domain which shares many characteristics of software engineering. Four kinds of representation are typically used during the design of a VLSI chip, namely "formal specification" (of I/O behaviour), "floorplan" (describing gross physical structure), "logic diagram" and "layout" (describing fine physical structure). Problems of verifying layout to specification rest on the formality of the design languages and the correctness of the design rules in use. Begg believes VLSI design may be automated on a four-level blackboard as a process of transforming circuit specifications at higher levels to those of lower levels by action of design rules.

We noted in the last section that new formal representations of systems are advocated for software engineering and more abstract representations may emerge from cognitive studies of programmers. Clearly, use of multiple representations of systems, like VLSI design, is a current feature of software engineering. This suggests that investigations of its automation may be flexibly served by a knowledge-based programming environment based on the blackboard architecture. The potential of the architecture to support these studies is also recognised by Spohrer[Spohrer87] whose designs for a blackboard system are at an early stage, and Siddiqi and Sumiga [Siddiqi&Sumiga86] who describe their cognitive model of software design using a blackboard representation.

1.3 Summary.

As we have seen, techniques of artificial intelligence hold promise for releasing the programmer from much of the complexity of implementation thereby improving system reliability. The techniques may even support software design if suitable formalisms can be found for describing systems at the early design stages. Cognitive studies of designers at work are shedding light on the nature of the many representations they actually use.

The survey of the knowledge-based programming suggests a number of principles for designing a flexible environment for investigating knowledge-based approaches to software production. The environment should provide a multi-level organisation in which systems can be developed at various levels of abstraction. Language tools should support the representation of programming knowledge of the domain, including the methods in use, in the form of rules. The blackboard architecture suggests a framework in which these principles can be implemented.

This thesis describes the interpretation of the blackboard architecture manifested in the ENCORES ("Environment for Constructing or Reasoning with Engineered Software") system. Designed, in part, to support an Alvey research project*, the environment is the product of a second attempt to adapt the blackboard architecture for the programming domain. The main lesson learned from the first implementation [McArthur85] is that the architecture must be interpreted for this purpose with accessibility of users in mind. Blackboards must not only record complex reasoning processes but reflect them in a transparent manner. Similarly, knowledge of the particular programming domain and methods of deploying that knowledge should find a perspicuous representation in a blackboard system for purposes of acquisition and analysis.

The author shares the view of Noorzij [Noorzij87] that the blackboard architecture is "object orientated" in so far as there is the ability to implement the levelling of blackboards as abstract data types. Knowledge sources are capable of performing operations on these types or classes without specific knowledge of their contents. Thus objects may be implemented as suited.

The author extends this view of blackboards as assemblages of objects to encompass the knowledge sources. These may also be viewed as objects which may be located on blackboards and act on themselves or other objects to effect control or inference. Since the object concept plays a major role in software engineering, from 'object-centred' requirements definition [Greenspan84] and specification [Goguen&Meseguer86] to programming [Goldberg&Robson81], it might also

* This work, entitled "Automatic Programming Expert System for Implementing Abstract Specifications as Quality Structured Programs", is supported by both the Science & Engineering Research Council under grant GR/C/9604.3 and Alvey as project /PRJ/SE007.

serve in the role of interfacing engineers to blackboard systems for knowledge-based programming.

The remainder of this thesis describes the particular interpretation of the blackboard architecture chosen for the programming domain and its implementation in the ENCORES system. Since both interpretation and implementation involve choices for the blackboard, knowledge sources and control mechanism, the document is structured to highlight these choices.

The next three chapters describe how the architecture may be interpreted as the generation of objects in certain classes by the action of other objects. The generative process and common classes used to construct blackboards, are described in chapter 2. Knowledge of particular programming domains may be represented as rules of APPEAL ("A Pleasant Programming-expertise Acquisition Language"), which are themselves objects as described in chapter 3. Similarly, control knowledge may be represented as APPEAL rules, as shown in chapter 4. In each chapter we relate these concepts to previous work.

The ENCORES system embodies these ideas. Implemented in PROLOG for the NU7 interpreter, the system is currently running under UNIX* Version 7 on the department's PDP11/70. Chapter 5 describes a blackboard management system implemented in PROLOG. APPEAL rules are compiled to an internal form for efficient evaluation as shown in chapter 6. Control is effected by a small PROLOG program whose performance is analysed in chapter 7. Finally, chapter 8 compares ENCORES with the author's

* UNIX is a registered trademark of AT&T in the USA and other countries.

previous blackboard system, suggests improvements and proposes new roles for the blackboard architecture in the engineering of large software systems.

CHAPTER 2.

A BLACKBOARD STRUCTURE FOR ENGINEERING SOFTWARE.

In this chapter we will show that program development may be represented on a blackboard as a process of object generation. The structure of these objects, their common classes and relationships, in the author's choice of blackboard structure, are described. It will be shown that this blackboard structure is likely to reflect the development process more transparently than those structures in other blackboard systems.

Programming, like most complex processes, yields to a 'divide and conquer' analysis, from which we gain principles for its representation. In the next section we show that programming can be understood as a design activity. Software development, like the design of VLSI chips, can be decomposed to a series of small steps, each reflecting a single design decision. Thus programming, like VLSI design (as studied by Begg[Begg84]), may be represented formally within a blackboard system.

The blackboard structures available in some system building environments are then studied. This analysis provides criteria for choosing a blackboard structure for knowledge-based programming and a yardstick from which to compare the author's choice of structure. The components of this structure are described and we show, with use of an example blackboard which might be generated during the automatic programming of an algebraic specification, that the development of the implementation is transparent.

2.1 Principles for Representing Program Development.

We first introduce concepts for factoring the software development process into smaller units which may be represented formally in a knowledge-based system.

2.1.1 Step-wise Refinement

A key principle is the formal use of stepwise refinement [Wirth71]. The idea in stepwise refinement is to start with an abstract description of the system under development which might be called loosely a 'specification'. At each step a part of this specification is decomposed or refined into more detailed parts as a result of a single design decision. For example, one might choose an efficient data type for another, or optimise some algorithm. The step results in a new, usually, more efficient specification of the system which is refined in turn.

Typically there exist alternative refinements that can be applied to any partially refined specification. They give rise to a tree-shaped 'search space' where each node of this tree represents a more or less refined specification with the root representing the initial specification [Smith etal85]. Arcs between nodes represent refinements. Alternative refinements are represented by the directed arcs from a node to the nodes representing the refinements of that node. For example, leaves of the tree might represent programs in the target language.

The process of synthesising code from a given specification can be viewed as a search through this tree for a leaf representing a program with satisfactory performance characteristics. Subtrees represent families of programs that share a common heritage (sequence of design decisions) and, thus, have common characteristics. This fact is useful in constructing heuristics for guiding the search.

It may be difficult to foresee the effects of a series of design decisions. If a partially redefined specification has unacceptable characteristics, then the synthesis process should back up and explore another sequence of refinements. If the space of possible implementations is large enough, then a programmer will not be able to manually explore many alternatives. For this reason automatic search within a knowledge-based system may find better implementations of a specification than can humans.

2.1.2 Multi-level Organisation.

We argued in the first chapter that use of multiple representations of systems play a major role in the design of software systems. (eg. Adelson's results, page 13). Thus a second principle for representing software development is the use of a multi-level organisation to clearly distinguish the representations used in the specification of the system.

In general, refinements of a specification will result in a new specification of the system at a more concrete level of representation. Figure 2.1 illustrates how the refinement process might be recorded on a blackboard with levelling suitable for the various representations under development.

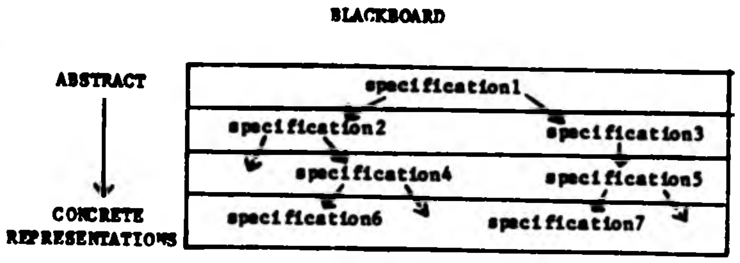


Figure 2.1 Program Development on a Blackboard.

The formalisation of step-wise refinement within a multi-level organisation provides three important factorisations of the programming process :

(1) There is the factorisation of the programming process into formulating specifications and implementing them via refinements.

(2) There is the factorisation of refinements and specifications according to the representations in use.

(3) When refinements are represented explicitly, there is a factorisation of programming knowledge. Since different sequences of refinements generally result in different implementations of the same initial specification, a relatively small number of refinements may be able to compactly encode a wide range of implementations.

Depending on the levels of abstraction employed within a blackboard system, some refinements may generate equivalent specifications within a single level. Local optimisations or decompositions may be desired. For example, programs often have to be massaged or 'conditioned' to meet the triggering conditions of knowledge sources [Wile82b]. Categories of refinement can be defined according to their effects at the same or adjacent levels, as proposed by Begg. However, new definitions serve no purpose other than to confuse. The term 'refinement' is taken here to mean that which generates any new specification regardless of the representations in use.

Refinements can be formalised in terms of rules that embody a single design decision and which generally have the effect of rewriting one piece of code into another. These rules will constitute the knowledge sources of the blackboard system.

Since programs may be altered by refinements, information will be retracted during the development process. Thus, by definition, the reasoning process will be 'non-monotonic' and have the character of knowledge-based planning or design [Tate85]. In contrast, monotonic systems reason wholly by asserting new information (eg. for diagnosis or interpretation).

2.2 Blackboard Structures available in some System Building Tools.

Of interest here are the data structures available in some general-purpose environments for building blackboard systems. At minimum, these environments provide tools to build blackboards with:

(1) The desired levels of abstraction.

(2) Suitable data elements to describe partial solutions.

(3) Pointers both within and across blackboard levels to model the logical support between these elements.

(4) Tools to represent domain knowledge as knowledge sources.

(5) Tools to represent and apply strategic or control knowledge.

The tools and data structures to build blackboards are of interest here. Knowledge representation in these systems will be described in later chapters.

There have been three well-documented attempts to abstract the blackboard architecture in system building environments. The HEARSAY-III designers [Erman etal82] provided a flexible environment based on a relational database and in which sophisticated control regimes could be represented. In contrast, the AGE system [Nii&Aiello79] was designed to provide great flexibility for representing complex reasoning tasks by virtue of the provision of numerous software components from which various blackboafd systems could be constructed. The ART system [Williams84] brings sophisticated architectural features not found in the other systems. Probably 'state of the art' (no pun intended) in blackboard technology, the ART system is available commercially. A brief description will be given of HEARSAY-III, AGE and ART. Particular attention will be given to those concepts which were influential in the author's choice of blackboard structure.

2.2.1 HEARSAY-III.

HEARSAY-III is a domain-independent framework for building knowledge-based systems. Founded on a relational database, the language of which is called AP3 [Goldman78], HEARSAY-III makes use of a context mechanism such that contexts play a key role in the reasoning mechanisms mode available to an application.

HEARSAY-III generates a tree of contexts during inference like those of figure 2.2. Each context is generated by the action of a knowledge source and contains the information contributed by that rule to the emerging solution. Where more than one knowledge source is applicable, alternative contexts are generated on alternative branches. Information which remains unchanged is inherited from father to daughter contexts and modifications are recorded as sets of assertions and/or retractions. Thus HEARSAY-III allows reasoning along independent paths, which may arise both from a choice among several competing knowledge sources and from a choice among several competing partial solutions.

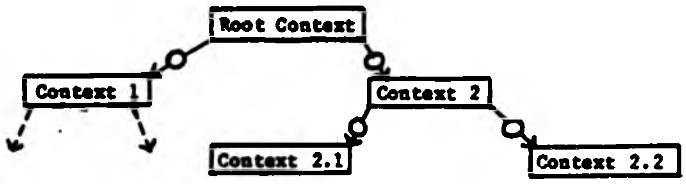


Figure 2.2. A Context Tree (boxes contain assertions; discs retractions).

The underlying AP3 database is used by an application program as a repository for a domain schema, representation of partial solutions and pending activities. HEARSAY-III supports the representation on the blackboard of graph structures. These consist of :

(1) 'Units' which are structured nodes of the graph, implemented directly as AP3 objects having a specified type. AP3 types have a hierarchical class structure, thus access can be restricted to a given blackboard level simply by using type-restricted AP3 database

retrievals.

(2) 'Roles' are labelled arcs linking units and are represented as typed relations in AP3. Type-restricted retrievals of roles allow suppression of detail along chosen dimensions when examining the blackboard. Role are placed in classes, called 'rolesets', which mimic the levelling of blackboards.

2.2.2 AGE (Attempt to Generalise).

The AGE project is described as an attempt to provide a 'software laboratory' in which system builders can experiment with a number of alternative problem-solving techniques applied to their particular applications. Interesting features of AGE are its description of its blackboard data type and its library of software components from which various search techniques or knowledge representation schemes can be implemented.

AGE provides a common representation for blackboards as a network of data elements, known as a 'hypothesis structure', built upon a database package known as UNITS [Stefik78]. The structural details of AGE blackboards follow closely the description of the architecture expounded by Erman and Lesser [Erman&Lesser75] and consist of three components:

(1) 'Hypothesis elements', are a named set of attribute-value pairs. Elements are arranged in levels, each describing partial solutions at a level of representation.

(2) Knowledge sources establish pointers (known as 'links') from elements which activated it to new elements which are generated. The links therefore model the logical support between elements and come in two varieties : 'expectation links' are directed to new elements from elements describing more abstract representations, and model deductive or theoretical support from above; 'reduction links' are directed to new elements from lower-level elements and model inductive or evidential support from below. Links can be further categorised as 'AND' or 'OR' to model support based on boolean combinations of elements.

(3) A separate structure, known as a 'changeset', records summary information of recent changes on the blackboard and is updated as inference proceeds. The changeset is used by the control mechanism of ACE to maintain focus of attention within the system.

Figure 2.3a shows a possible element from an application of ACE to crystal structure interpretation, known as CRYSALIS [Engelmore&Nii77]. Such an element may be generated in the hypothesis structure depicted in Figure 2.3b. In this figure 'r' label reduction links and 'e', expectation links.

Hypothesis Level: ATOMS
Hypothesis element name: ATOM-10
Inferred attribute-value pairs:

Atom.name	Sulfur
Location	(12.3 13.6 24.2)
Partof	(OR CYS14 CYS17)
Bonded to	HEME-1
Bondtype	Hydrogen

$$\begin{tikzcd}
 & & \text{Molecule} \arrow[dr, "e"] & \\
 & \text{superatom-1} \arrow[ur, "r"] & & \text{superatom-2} \arrow[d, "e"] \\
\text{atom-1} \arrow[ur, "r"] & & \text{atom-2} \arrow[ul, "r"'] & \text{atom-3}
\end{tikzcd}$$

Figure 2.3 CRYSALIS Hypothesis Element (a) and Structure (b).

2.2.3 ART (Automatic Reasoning Tool).

The ART system includes a number of advanced features not found in AGE nor HEARSAY-III. Firstly, techniques of truth maintenance are incorporated in ART to guide the search process by permitting selective back-up to earlier states of the blackboard. Secondly, a general purpose man-machine interface has been devised for the system based on the concept of 'viewpoint'.

A model of truth maintenance has been developed for the system by Williams. It can be used by labelling certain clauses of ART rules as assumptions from which a number of alternative lines of reasoning can follow. As inference proceeds along one of these alternatives a situation of stalemate or contradiction might be detected. Rather than back-up to the previous state at which the last rule applied, the system can backup selectively to the point where assumptions were made. Essentially, reasoning based on guesswork is given an explicit representation.

The 'viewpoints' mechanism of ART provides a graphic presentation of the reasoning process at any time the consultation is suspended. The user can peruse a network of viewpoints, within any single level of the blackboard, where each viewpoint is presented as a box containing data deduced by execution of a rule. Similar to the contexts in figure 2.2, each viewpoint inherits data from its ancestor, if not retracted by application of the rule. Retractions are annotated to each viewpoint.

From any given viewpoint, the user can locate viewpoints at other blackboard levels by following 'pathways' which link consistent descriptions of partial solutions at adjacent blackboard levels. Pathways linking any viewpoint at the highest level to one at the lowest level can be scanned to give a 'total view' of the emerging solution.

The viewpoints mechanism serves many purposes since it can be used to:

(1) Reveal erroneous deductions and thus aid rule debugging.

(2) Reveal the alternative lines of reasoning being explored along alternative branches of the viewpoints structure.

(3) Reveal more abstract or specialised descriptions of a viewpoint on adjacent blackboard levels.

(4) Trace the complete development of the solution.

In all other respects, ART shares some of the best features of AGE and HEARSAY III. A further bonus is that a clear distinction is made between uses of the system for plan-based reasoning (non-monotonic) and analytic (monotonic) reasoning. Further, the rules for combining both types of reasoning (albeit at different) blackboard levels are made clear.

The blackboard structures available in these three systems can be analysed along a number of dimensions to determine their suitability for applications in knowledge-based programming :

(1) Dependence on underlying representations. The three systems, in the sequence described, show increasing use of data structures intended to reflect the reasoning process to users. Blackboards of HEARSAY-

III are understood entirely in terms of the data types of the underlying package. AGE uses framelike structures [Minsky75] which are common in AI work.

ART blackboard components are not only metaphors for everyday concepts (eg. 'views' and 'pathways') but are visible through a graphical interface. There is clear advantage in describing blackboard structures in terms that are familiar to intended users of the system.

(2) Use of contexts. HEARSAY-III and ART both provide a context mechanism for reasoning amongst alternatives; links of AGE model logical support between elements directly. Consider the resulting change in blackboards of these systems by execution of a rule. Both HEARSAY-III and ART establish a single link to elements in the new context. All the elements which trigger the knowledge source in AGE are linked to all new elements generated. For this reason pointers are likely to be profuse in AGE blackboards, information in alternative lines of reasoning less visible and consistent reasoning more difficult to manage. Contexts model logical support economically and visibly as demonstrated by the success of the ART viewpoints mechanism.

(3) Use of Inheritance. ART maintains different graph structures to model monotonic reasoning and non-monotonic reasoning. For applications in design the user has to scan back through the tree of viewpoints, taking note of retractions on the way, to ascertain all information inherited to the viewpoint of interest. The viewpoint mechanism is therefore less convenient for non-monotonic

reasoning. In contrast, the elements of AGE are visible 'in toto', although the underlying UNITS package may make use of inheritance. One may therefore conclude that inheritance should be an implementation feature for applications in programming (particularly as programs are large structured objects). It should be avoided in the blackboard structure as presented to users.

(4) Use of truth maintenance. Truth maintenance techniques can save considerable duplication of effort and therefore may find a role in knowledge-based programming (eg. in CAKE, page 12). However the techniques are difficult to implement well so no attempt has been made to incorporate them within the author's blackboard structure.

2.3 An Object-centred Blackboard Structure.

The concept of object plays a key role in the author's interpretation of the blackboard architecture. All descriptions of a program under development or knowledge to effect this development are represented as objects. Objects have an identity and attributes, like the elements of AGE. All objects are considered to reside in a stack and have the following properties:

(1) they, or their attributes, may be in the class GLOBAL.
GLOBAL objects, or those attributes in GLOBAL, do not change their values during problem solving.

(2) any object not in GLOBAL belongs to one or more CONTEXT classes. Each CONTEXT describes a partial solution to the problem after some rule is applied.

(3) each CONTEXT belongs exclusively to a class of LEVEL. LEVELS have a name and precedence. Contexts in higher precedence levels represent more abstract descriptions of one or more contexts at lower precedence levels.

(4) each level belongs exclusively to a BLACKBOARD class. Blackboards are used to develop solutions to problems. A distinguished blackboard, known as the DOMAIN blackboard, is used to develop programs. (Another blackboard, known as CONTROL, plays a major role in problem solving and is described in chapter 4).

Blackboards are constructed using LINKS (ie. pointers) of two sorts:

(1) ABSTRACTION links are directed between two contexts on adjacent levels of a blackboard. The contexts so linked provide consistent descriptions of a partial solution, albeit at different levels of detail. Generated with the context to which they are directed, they model support across blackboard levels.

(2) REFINEMENT links are directed between two contexts on any single level of the blackboard. Generated with the context to which they point they model logical support amongst local optimisations of a

program. Objects are not inherited along these links, however the relationships established by abstraction links are inherited down the lineage (until a new abstraction link is established).

These classes and links fulfill similar functions to components of AGE, ART and HEARSAY-III. OBJECTS, CONTEXTS and LEVELS have obvious counterparts in these systems. The concept of OBJECTS residing in a stack is both a feature of the interpretation and implementation (described in chapter 5). The stack, like the changeset of AGE, is used to maintain focus of attention in the system. ABSTRACTION links function like the pathways of ART. We have not, as yet, considered the addition of blackboard 'planes' or multiple abstraction hierarchies to our blackboard structure.

2.4 Software Development as Object Generation : an example.

To give substance to these ideas we present a small example of a blackboard which might be used in the automatic programming of an algebraic specification of a system to a functional program.

2.4.1 Algebraic Specification of a Stack.

Readers will recall that the algebraic specification technique permits the precise, formal definition of the functional behaviour of abstract objects as sets of algebraic equations. Figure 2.4 is an example for a simple stack, expressed in the HOPE functional language [Burstall etal80]. (My comments are enclosed with '/*..*/').

```
module anystack;
pubtype stack;                 /*'stack', is the object being specified*/
pubconst pop,top,empty,push,newstack;   /*operations of 'stack/*
typevar alpha;     /*'alpha' is a type variable*/
data stack(alpha)==newstack++push(stack(alpha)#alpha);
                                 /*'newstack' and 'push' construct stacks*/
dec pop : stack(alpha) -> stack(alpha);        /*operation signatures*/
dec top : stack(alpha) -> alpha;
dec empty : stack(alpha) -> truval;
--- pop(newstack) <= undef;                  /*operation axioms*/
--- pop(push(s,item)) <= s;
--- top(newstack) <= undef;
--- top(push(s,item)) <= item;
--- empty(newstack) <= true;
--- empty(push(s,item)) <= false;
end
```

Figure 2.4 Algebraic Specification of a Stack.

Essential points to be noted in this example are:

(1) The 'stack' object is not represented by any explicit data structure.

(2) Equations with 'dec' symbol describe the operation 'signatures', that is, the input sorts and output sort of each operation.

(3) The 'stack' object is defined via its operations, the meaning of which are implicit in the axioms.

(4) Operations 'newstack' and 'push' build stacks and are thus called the 'constructor' operations.

(5) An axiom exists for each operation applied to each constructor operation, with result shown right of '<=' symbol.

(6) Symbol 'undef' means the application of the operation is undefined

(7) The 'stack' type is composed of elements of a type which is unspecified. The specification is said to be 'parameterised' and the type variable, 'alpha' may be instantiated to the chosen type of the element.

2.4.2 Representing the Problem.

Choice of levels should be convenient for the application. For example, the programming of the stack specification might be conducted conveniently on a three level blackboard. The abstract datatype is described by its operations which in turn are defined by its axioms or equations, thus the levels might be named DATATYPE, OPERATION and EQUATION respectively. (This choice of levels simplifies the discussion and leads to a simple decomposition of the problem but does not demonstrate the full power of abstraction).

The programming of the stack specification can be approached in the following manner:

(1) Analyse the axioms to characterise the action each operation makes with respect to the abstract object (in the style of APE, page 5).

(2) Choose representing type(s) for the abstract object based on this analysis.

(3) Generate equations which relate operations of the abstract object with those of the representing type(s).

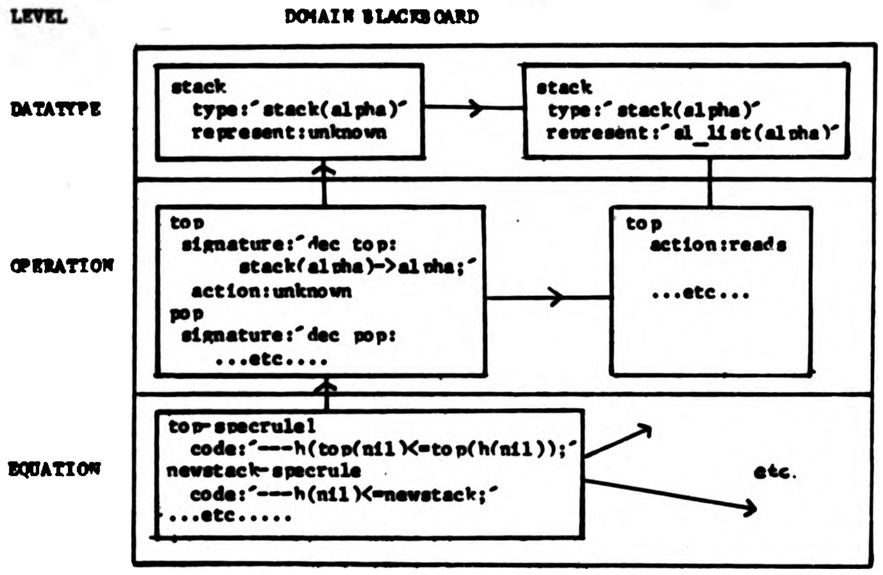
(4) Transform these new equations to equations of the functional program.

The four steps form a metaplan for solving this problem. Steps 1 and 2 are essentially those taken in the APE system Full details of steps 3 and 4 are saved for later chapters, suffice to say that they follow a procedure recommended by Kapur and Srivas [Kapur&Srivas85].

In a fully automatic system the steps would be accomplished by knowledge sources. Indeed, the order of the steps describe a strategy for solving the problem and may also be represented by knowledge sources to effect this order. The representation of this knowledge forms the subject of the next two chapters. For the moment, we illustrate how the solution would unfold on the blackboard.

2.4.3 A 'Snapshot' of the Blackboard.

Figure 2.5 illustrates how a three-level blackboard might look after the first three steps have been taken with the 'top' operation. Vertical lines depict the ABSTRACTION links; horizontal lines the INHERITANCE links. Each box is a CONTEXT. The blackboard would be initialised with the root contexts shown at the DATATYPE and OPERATION levels. The three new contexts shown would result from these steps.



```
stack
  type:'stack(alpha)'
  represent:unknown

stack
  type:'stack(alpha)'
  represent:'al_list(alpha)'

top
 signature:'dec top:
      stack(alpha)->alpha;'
  action:unknown
pop
 signature:'dec pop:
    ...etc....

top
  action:reads

  ...etc...

top-specrule1
  code:'---h(top(nil)<=top(h(nil));'
newstack-specrule
  code:'---h(nil)<=newstack;'
...etc.....
```

Figure 2.5. Blackboard for Automatic Programming.

Ideally, this blackboard would be shown via a graphical interface, like that of ART. At second best, a list of each context's contents might be obtained, including a description of the links to and from it. In either case, the development of the program can be followed at each level of representation.

2.5 Summary.

Some principles were developed for representing programming within a knowledge-based system. In particular it was shown that Begg's description of design within a multi-level organisation can be adapted for programming since it is essentially a design activity.

We analysed the blackboard structures used by some domain-independent blackboard systems. This analyses showed that there is advantage in choosing a structure which reflects the reasoning process using concepts familiar to the user.

The object concept is familiar to the software engineering community and should form the basis of a blackboard structure for knowledge-based programming. One such structure, that in the author's interpretation of the blackboard architecture, was described and an example given to illustrate its transparency.

Although the components of the author's blackboard structure are essentially those adopted in the ART system, they are placed entirely within a class hierarchy of objects. The blackboard structure of the ARIADNE-1 system [Craig86] is described in terms of objects and classes, however the system does not use contexts and the object concept is not reported as central to the user interface. The author is unaware of any other blackboard structure, besides his own, which gives comparable prominence to the object concept as a descriptive aid to its intended users.

CHAPTER 3.

AN OBJECT-CENTRED LANGUAGE FOR REPRESENTING PROGRAMMING KNOWLEDGE.

The representation of programming knowledge in knowledge-based systems presents special difficulties. In many domains, experts have collections of terms which define key concepts, objects or relationships within their specialism. A knowledge engineer can frequently encode this knowledge as simple patterns of symbols for these terms, in some pattern-directed inference system. Programmers have their own definitions and jargon but the focus of their activity is normally on programs - objects which have elaborate structure and syntax. Consequently, any representation language for expressing programming knowledge must also include some metalanguage facility for describing programs.

One such representation language, known as APPEAL, is described in this chapter. An attempt has been made to reconcile many requirements in the design of this language. Some of these requirements appear to be contradictory. Of particular concern was the need for accessibility. An earlier representation language, designed for use with the blackboard architecture, was not built with access in mind [McArthur86]. As a result it was difficult to use. Experience gained in building a knowledge-base [McArthur85], suggested that knowledge acquisition was difficult enough without the added task of translation to an obscure language. The title chosen for the new language, "A Pleasant Programming Expertise Acquisition Language", reflects the high priority given to ease-of-use in its design.

The chapter proceeds as follows. Some requirements for representing programming knowledge are described. We then review some languages developed for use with software tools. We will see that the design of high-level languages within flexible environments is now common at the outset of major investigations of knowledge-based programming. These environments frequently use an internal representation for both programs and programming knowledge which brings efficiency with some degree of independence from the particular programming language in use. One such representation, based on attributed trees, will be described. Finally, APPEAL, a rule-based language for manipulating objects of the sort described in the last chapter is described. It will be seen that APPEAL shares the best features of other representation languages.

3.1 Requirements for Representing Programming Knowledge.

Knowledge representation is now receiving greater attention as a topic of research in its own right. These investigations focus on the representation and organisation of knowledge into a coherent structure which captures the semantics of some slice of reality while being understandable by its designers [Greenspan etal85].

Rule-based representations have attracted most interest from AI investigators following their successful use in the MYCIN system [Shortliffe76]. It has been shown that rules can be used to represent strategic knowledge [Davis80a], support explanations of consultations [Clancey83], tutor students [Sleeman81], represent knowledge induced within machine learning systems [Bundy etal85][Boyle85]. and be used with many types of search strategy [Nilsson80][Davis&King76].

Despite the popularity of rules for representing programming knowledge in the TI, APE and CHI system (described in chapter 1), other representation schemes have been used. The PROUST system (page 9) has knowledge of program plans represented within frames [Minsky75]. Frames are object representations with 'slots' in which attribute information is held. Slots may include pointers to other frames and frequently to a frame which describes the class to which the object belongs. Reasoning is simulated by attempting to 'fill-in' missing slot information.

Regardless of representation scheme chosen for an application, one requires a formal language to describe the knowledge. There have been attempts to design general-purpose representation languages (eg. RLL, described in [Barstow etal83]). However, these do not appear to be used for applications in programming.

The decomposition of programming to specification and their refinement, described in the last chapter, presents us with the problem of representing refinements. Investigators in the knowledge-based programming field offer a number of suggestions for representing programming knowledge:

(1) Represent refinements as rules. The Kestrel Institute recommend use of rules for representing refinements [Smith etal85]. Each rule should embody one design decision and should generally have the effect of rewriting fragments of code. They suggest that the language for expressing refinements make use of metalanguage expressions, where the varying parts of the pattern are represented by metavariables. Premise and conclusion parts of such rules will be instantiated when matched against suitable programs and effect code substitution.

The design of the component of the V language used to describe rules is based on these principles. (An example of such a rule is given on page 11).

(2) Use a language-independent representation. Programmers produce programs expressed in the language syntax which they are familiar. The knowledge employed, however, comprises that of software design, algorithms and use of programming language constructs which could be used with a class of languages, not just the language with which the programmer is familiar. For this reason, Rich [Rich81], recommends that knowledge representation should be independent of particular languages. The 'plan' representation of programs, introduced by him, shows promise as being a language independent representation. Code fragments describing plans can be mapped into closely related languages given a precise knowledge of their execution models.
Consider, the frame describing the 'sentinel-process-read-while' plan shown in figure 3.1. The 'template' slot describes the form PASCAL code, implementing this plan, should take. (The '?*' symbol will match any PASCAL statements and is a place holder, otherwise '?' precedes metavariables. SUBGOALs in the template are goals that must be implemented using other plans.) The template could be replaced with similar metalanguage expressions for other languages.

```
(Plan-Definition Sentinel-Process-Read-While
     Constants            (?Stop)
     Variables            (?Input)
     Template             ((SUBGOAL (Input  ?Input))
                              (WHILE (?Input <> ?Stop)
                                  (BEGIN
                                   ?*
                                  (SUBGOAL (Input ?Input)))))
```

Figure 3.1   Frame describing a Plan.

(3) Use a representation which is 'reversible'. Another interesting feature of plans, as described by Rich, is that they have no particular orientation towards program analysis or synthesis. Whatever their intended use in an application, they can be used reversibly. He recommends this feature for any representation scheme.

To these requirements, we add that the representation should be powerful, in that knowledge from many applications to programming should find their representation in it. The representation language by which programming knowledge is expressed should be perspicuous to all users of a system, not just the knowledge engineer.

However, the requirement for perspicuity appears to be at odds with the requirement for language independence. We saw that rules expressed in the V language of CHI appear particularly accessible but this is because rule patterns are based on the syntax of the application language. Must program knowledge be expressed in terms of abstractions (eg.plans), which are potentially language independent? This dilemma might be resolved by compilation of knowledge from friendly representations (which use V-like patterns) to internal representations which are language independent or potentially so.

Since a considerable number of rule-based systems have been built over the last decade, there are many rule languages from which one may draw concepts for designing a language. A brief survey will describe those which have influenced the design of APPEAL.

3.2 Knowledge Representation Languages in some System Building Tools.

Efforts to abstract knowledge-based systems for use in other domains have given particular attention to the design of languages for representing knowledge. Some of these efforts will be described with examples of rules, many of which are drawn or inspired from a survey of such tools [Waterman&Hayes-Roth83] and a comparison based on their application to the management of spills of toxic liquids [Johnson&Jordon83].

One language which greatly influenced the author's work is ARL (Abbreviated Rule Language), the main language tool of EMYCIN (Extended MYCIN [Van Melle etal79]), the environment abstracted from MYCIN.

ARL is a terse, stylised, but easily understood language for writing rules. Described as being similar to a shorthand some domain experts use to sketch out rules, it uses simple operators to relate parameters and their values in both premise and conclusion parts. Figure 3.2 shows a rule which might be used in an application, built with EMYCIN, to advise on the management of toxic liquids. Note the symmetry of rules arising from the dual use of "=" for matching (in rule premises) and generating data (in rule conclusions).

```
     if   name = MATERIAL  AND
          colour = colourless  AND
          class  = acid  AND
          ph-value < 3
    then  toxicity = hazardous
```

Figure 3.2 Example of ARL rule.

ARL includes constructs for building rule antecedents from arbitrary Boolean combinations of predicates of associative triples (ie. object-attribute-value triples). Conjunction and disjunction of triples is achieved using AND and OR respectively. Predicates, including negation, can be applied to these combinations of triples and the user can stipulate an alternative consequent (using the ELSE construct) should the rule antecedent be false.

To accommodate uncertainty, a 'certainty factor' may be associated with every associative triple in EMYCIN. This number, a normalised probability, ranges from -1 (when the triple is false) through zero (no opinion) to +1 (unquestionably true). A certainty factor may also be associated with each ARL rule for the purposes of updating the certainty of the consequent in the light of evidence from the antecedent according to certainty-theory formulas of MYCIN [Shortliffe and Buchanan75].

KAS (Knowledge Acquisition System) is a knowledge engineering environment abstracted from PROSPECTOR, a consultation program developed for diagnosis problems that arise in mineral exploration [Duda etal79].

Viewed abstractly, the rule languages employed by EMYCIN and KAS are very similar. Rule antecedents and consequents may be arbitrary

Boolean combinations of propositional statements. However these statements are a generalisation of object-attribute-value triples to semantic networks in which hypothetical worlds may be described [Hendrix75]. This allows a statement to represent a situation involving n-ary relations among any number of objects.

To accommodate uncertainty, KAS associates a probability value with every statement. This number measures the degree to which the statement is currently believed to be true.

The probability of Boolean combinations of statements are computed using Zadeh's rules for fuzzy sets [Zadeh65]. Using these formulas, the probability of a hypothesis that is defined as the logical conjunction (AND) of several pieces of evidence equals the minimum of the probability values associated with this evidence. Similarly, a logical disjunction (OR) of evidence pieces is assigned a probability value equal to the maximum of these values.

Each rule includes two numerical strengths used to update the probability of its consequent through the use of Bayes' Rule [Duda etal76]. The first one is used if the antecedent is determined to be true, while the second is used if the antecedent is determined to be false. When the probability of the antecedent is somewhere between zero and one, an appropriate intermediate rule strength is used.

An example of a rule in the language described for KAS is shown in figure 3.3. Such a rule might be used by a diagnosis program advising on the best strategy for locating a spill (E1) of some toxic liquid. The measures of rule sufficiency (eg. 500) and necessity (eg. 0.3) are both ratios of probabilities and the first may be arbitrary large. The user expresses her certainty about the antecedent on an arbitrary -5 (definitely absent) to 5 (definitely present) scale.

```
    if (OR (AND (composed-of E1 oil)
                (size-of     E1 large))
           (AND (composed-of E1 acid)
                (ph-value-of E1 strong))
       )
  then (to degree 500,0.3)(priority-of E1 emergency)
```

Figure 3.3. Example of rule in language of KAS.

OPS5 is a rule-based programming language [Forgy81] descended from earlier OPS languages designed for AI and cognitive psychology applications. The language incorporates general purpose control and representation schemes.

OPS5 provides a single, global data base called 'working memory' consisting of 'vectors' (lists of symbolic values) and/or objects with associated attribute-value pairs. A typical attribute-value element (with carats distinguishing attribute names from values) might look like,

```
(MATERIAL ^NAME H2SO4 ^COLOUR COLOURLESS ^CLASS ACID).
```

The statement describes H_2SO_4 as a colourless acid.

Elements in working memory may vary dynamically when OPS5 rules are applied. Such rules consist of Boolean combinations of 'patterns' in the antecedents which are partial descriptions of data elements. Variables (symbols enclosed with angle brackets) in patterns are matched with components of the data element with multiple occurrences of a variable being bound to the same value.

Rule consequents may MAKE a new data element and add it to working memory; REMOVE one or more elements from working memory; MODIFY one or more subelements of an existing element or WRITE information on the user terminal.

An example of a rule in the language described for OPS5 is shown in figure 3.4. This rule may be used in a spill-management diagnostic application built with OPS5 and is named "beware-colourless-acids". The action of the rule will be to modify the subelement of the pattern matched and having attribute 'status' to take the new value 'hazardous'.

```
(P  "beware-colourless-acids"
 (material  ^name <MAT> ^colour colourless ^class acid ^status <X>)
->
(MODIFY ^status hazardous)
```

Figure 3.4 Example of rule in OPS5 language.

Similarly, efforts to abstract the blackboard architecture and, more recently, knowledge-based software tools as general-purpose environments, give language development particular attention. Some combination of language concepts from these disparate systems are necessary for our purposes.

3.2.1 Languages for Blackboard Systems.

Environments based on the blackboard architecture provide tools to represent knowledge sources as rules. These range from simple pattern-matching facilities, in the case of HEARSAY-III; a language for describing rules and constraints, in the case of ART; to a library of rule evaluation algorithms, in AGE, by use of which languages may be custom built for particular applications.

Each knowledge source of HEARSAY-III (page 27) is LISP code, which can be thought of as a large grained production rule. It consists of :
(1) a triggering pattern or predicate whose primitives can be AP3 fact templates composed with AND and OR operators and arbitrary LISP predicates;
(2) immediate code which associates information with the instantiation of the rule when triggered;
(3) a body which is run by the control mechanism in the triggering context with these instantiated variables.

For example, a Hearsay-III knowledge source from an application to advise on pollution management might look like the rule of figure 3.5. Note that the rule antecedent and consequent comprise patterns of triples with mixed-case symbols representing variables. This KS will trigger on the discovery of a caustic and volatile pollutant. The KS body will execute at a very high priority and send an appropriate warning to the user.

```
(declare-KS Warn-Caustic-Volatile (Material Observation Observer)
 Trigger:    (AND (MATERIAL Observation Material)
                  (CHEMICAL-ATTRIBUTE Material CAUSTIC)
                  (CHEMICAL-ATTRIBUTE Material VOLATILE)
                  (OBSERVER Observation Observer))
 Sched-level:  Emergency-Level
 Action:       (COMMUNICATE Observer
                "Warning:" Material
                " is caustic and volatile-DON'T BREATHE IT!"))
```

Figure 3.5. Example of HEARSAY-III knowledge source.

The language of ART uses similar predicates and patterns as

HEARSAY-III, but is syntactically sugared to remove much of the bracketing. Both HEARSAY-III and ART permit reference to specific contexts (described as viewpoints in ART) and the representation of 'constraints'. Constraints are sets of predicates which must not succeed together in a context. Contexts which do not comply with constraints are deemed contradictory or false and are 'poisoned', that is, removed from the search process.

An example of a constraint in the language described for ART is given in figure 3.6. This constraint prevents consideration of lime as a treatment for acid spills. Note that ART blackboards are represented with use of frames and 'Schema' indicates that attributes of the 'spill' object are to be matched.

```
(DefContradiction  lime-treatment-on-alkali
        "lime doesn't neutralise alkali"
        (Schema spill
                (material  alkali))
        (Schema spill
                (neutralise lime)))
```

Figure 3.6. Example of an ART constraint.

In contrast, a knowledge source of AGE consists of a labelled set of rules which are considered to belong together and describe a large chunk of knowledge. Associated with each KS are a common precondition of the rules and other information helpful in guiding their selection.

Since AGE elements are large frame-like structures (eg. see figure 2.3a, page 30), rules are used to add new elements, modify existing elements or establish links in the hypothesis structure. Rule premises are arbitrary LISP predicates but rule conclusions may have one or more PROPOSE statements of form,

PROPOSE <Changetype> <Name> <List of Attribute-Value Pairs>

where 'Changetype' is ADD or MODIFY and 'Name' is the element's identity. The attribute pairs describe a new element, for addition to the hypothesis structure, otherwise they describe alterations for modifying an element.

Evaluators can be assembled for rules having clauses joined with boolean operators AND, OR and NOT, with evaluation based on this logic. Alternatively, evaluation can be based on some subset of clauses of a rule premise succeeding, as for the EXPERT system [Weiss&Kulikowski79].

For example, the knowledge that "Oil spilling into water causes a sheen" would be represented with the rule of figure 3.7.

```
If      ($VALUE 'DISCOVER DESCRIPTION LATEST) = 'SHEEN
then    (PROPOSE change.type MODIFY
                 hypo-element 'DISCOVER
                 attribute-value (INITIAL-ID 'OIL)
                 support DISCOVERY
                 event.type OA3
                 comment If a sheen is observed,
                         then oil may have been spilled)
```

3.7. Example of an AGE knowledge source.

These languages are used to represent knowledge of unstructured objects in some domain. Since programs are large structured objects, language features for matching and generating program fragments are required.

3.2.2. Languages for Representing Programming Knowledge.

Centres with long term research programmes in knowledge-based programming place heavy emphasis on development of representation languages. The V language and POPART (Producer of Parsers and Related Tools) were developed at Kestrel Institute and Information Sciences Institute, respectively. Both V and POPART are, properly, tools supporting language development. However, both include constructs for representing programming knowledge, and we describe these here.

Knowledge is represented as rules in both the rule sub-language of V and POPART. Figure 3.8 shows equivalent rules (to remove a redundant conditional) in the syntax described for these languages. Other similarities are:

(1) All transformation rules are given an unique name to facilitate direct selection and application by users.

(2) Both use metalanguage expressions, based on the language of the program being manipulated.

(3) Both use an internal representation for programs based on attributed trees of an abstract grammar.

(4) Metavariables are used to match with varying parts of the program. Such variables have form '@<Identifier>' in VRL, or '!<Sort><Identifier>' in POPART where 'Sort' constrains the match to specific constructs of the programming language.

(5) Symbols of the abstract grammar can be used with metalanguage expressions to constrain the matching to particular constructs of

the programming language. These symbols are used within metavariables in POPART and used as predicates in VRL.

(6) Pattern substitution in context is achieved by use of the 'Match' command in POPART. Patterns, conjoined with a single '..', in VRL, results in a similar search through the code for matches.

(7) The extent of the pattern substitution can be quantified. POPART command 'Replace-All' is used to indicate that all occurrences of matching code is to be replaced. The universal quantifier, V, is available in VRL for this purpose.

```
rule replace-with-else(S1)
  'if false then @S else @S1'
  ^ statement(@S1)
  ->
  @S1
```

(a) VRL Rule

```
command ReplaceWithElse()
begin
    Match  if false
           then !Statement
           else !Statement#
  Replace !Statement#
 end
```

(b) POPART Rule

Figure 3.8. Equivalent rules in two representation languages.

We see that these languages use pattern substitution as the basis of their description. Token strings, and patterns based on these, are a familiar representation to the programmer but are too inefficient to provide a basis for a machine implementation. Representations based on abstract descriptions of programming languages can provide this efficiency whilst being potentially language independent. Attributed trees are the current choice of the CHI and POPART designers (although future versions of CHI will use internal descriptions based on logic [Westfold84]).

3.3. An Attributed-tree representation for Programs.

An attributed-tree representation of programs, based on an abstract grammar, can be used to retain program structure. Nodes of the trees correspond with sorts of construct defined for the programming language. Terminals are tokens of the surface syntax.

For example, the HOPE statement,

'dec top : stack(alpha) -> alpha;'

can be represented by the tree of figure 3.9 based on the abstract syntax defined in figure 3.10.

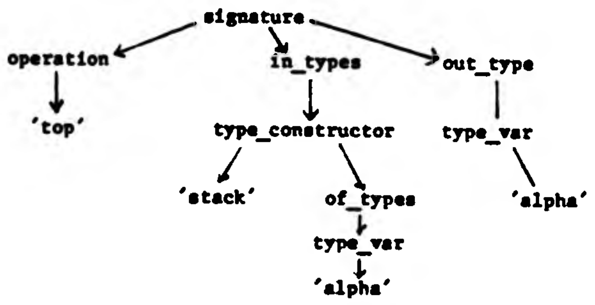


Figure 3.9 Abstract Syntax Tree

```
1. signature => as_oper : operation,
                as_in : in_types,
                as_out : out_type.
2. operation => as_identifier : id.
3. in_types => as_list : seq_of cl_type.
4. out_type => as_type  : cl_type.
5. cl_type ::= cl_primitive ^ typevar ^ type_expression ^
               type_constructor.
```

Figure 3.10 Sample of Abstract Syntax for HOPE.

This abstract syntax is defined using a notation, known as IDL [Nestor etal81], which is well suited for defining tree structures. An IDL statement defines the sort of some program construct by use of an equation with the sort to be defined on the left-hand-side and a number of attributes on the right-hand-side.

Each attribute must be of the sort specified after the colon. In our example, a signature sort has three attributes (descendent nodes) named 'as_oper', 'as_in' and 'as_out'. Each attribute is either constrained to be a specific sort (eg. as_oper must be an 'operation' sort); or a sort from some class (eg. the attribute of 'out_type' must be a sort from the class 'cl_type'); or may even be a sequence of sorts (eg. 'in_types' must be a sequence ('seq_of') sorts from class 'cl_type').

Classes are defined in IDL statements using the '::=' symbol and include a list of exclusive sorts (eg 'cl_type' must be either a 'cl_primitive' or 'typevar' or a 'type_constructor'). (Note that the '^' character is used to indicate disjunction in this example).

Abstract syntax trees are now used in a variety of advanced software tools. They provide a convenient intermediate representation within compilers. Code generators can be written for particular hardware while the parser is reused. For example, DIANA [Goos&Wulf83] is an abstract syntax defined for the ADA language. APPENDIX-C contains an abstract grammar for a useful subset of the HOPE language in this notation.

Efforts have been made to translate programs of one language into another language making use of abstract syntax trees (eg. ADA to PASCAL [Albrecht etal80]; and FORTRAN to ADA [Slape&Wallis83]). Common constructs of the two languages should have common syntax defined for them. Translation of other language constructs will involve a process of transformation. Abstract syntax trees are language independent in so far as they provide a convenient representation for a core of constructs from any particular class of broadly similar programming languages.

Powerful editors use tree representations of programs to hide details of the code whilst users establish the structure [Leblang82]. Such tools encourage top-down design and have been integrated in environments such as MENTOR [Lang85]. The author's previous rule language used attributed tree representations directly. Programs were transformed as trees using templates which include metavariables for matching subtrees. For example, the template 'signature(operation(O),In,Out)', would match the tree of figure 3.4 binding 'top', 'out_type(type_var(alpha))' and 'in_types([type_constructor(stack,of_type([type_var(alpha)]))])' to variables O, Out and In respectively. However, the language demanded knowledge of the abstract grammar and the bracketing was painful in use.

3.4. APPEAL (A Pleasant Programming-Expertise Acquisition Language).

We saw in the last chapter that programs are represented on the blackboard as assemblages of objects. The author's representation language, APPEAL, is object-centred in that refinements are represented with rules to generate objects. These rules are themselves objects and may be located on a blackboard. (We will see in the next chapter that APPEAL rules can address themselves).

APPEAL includes constructs for matching objects and matching code within objects. A number of examples of APPEAL rules are given to illustrate these language features using the HOPE language and stack example of the last chapter. After showing rules to manipulate objects within a single blackboard level, we show rules to manipulate objects on multiple levels.

3.4.1 APPEAL Rules for use with a single representation.

Knowledge-based tools generally require low-level transformations to massage or condition programs into states which will match other rules. Low-level transformations typically regroup nested expressions. Figure 3.11 shows rules for (a) commuting terms, and (b) associating terms to be added in an expression.

For example, the expression '(x+y)+z' might be transformed to '(y+x)+z' by action of rule 3.11a, or 'x+(y+z)' by action of rule 3.11b. (Note that matching is performed on structural components of programs and not arbitrary partitions of tokens).

```
  if    an equation(1) has
  &      code = '__A+B__'
then    an equation(1) has
  &      code = '__B+A__'
```

(a) Commuting terms

```
  if    an equation(1) has
  &      code = '__A + (B + C)__'
then    an equation(1) has
  &      code = '__(A + B) + C__'
```

(b) Associating terms

Figure 3.11. Some low-level transformations.

APPEAL rules consist of a number of templates of objects in either left-hand or right-hand sides. The generic form of each template is

'<OBJECT>(<REFERENCE NO>) has'   <Attribute-Value Pairs>.

Use of existential quantifiers 'a' or 'an' is optional. Every object has a specific identity as its 'name' attribute but need not be used : local reference numbers (in brackets) denote the same object in the context of the rule. Attribute values are accessed by equations :
'<ATTRIBUTE VALUE> = <APPEAL EXPRESSION>'.

Portions of code may be located or substituted by the use of patterns within single quotes. Symbols with initial capital are pattern-match variables. If patterns are also enclosed within the wildcard symbols, '__', then code will be matched and substituted in context.

Use of local reference numbers outwith, and ARL-like syntax within object templates, provide similar syntax to describe rule premise and conclusion parts. Rules have a symmetrical appearance and no further descriptor is needed to emphasise that a modification to objects is intended (as for ACE rules). Similarly, objects in rule conclusions, not referred to in rule premises, are to be added to the blackboard.

More than one object may be matched or generated by an APPEAL rule at any given blackboard level. Figure 3.12 shows two transformations which have a more significant effect on functional equations. FOLD replaces code matching a function body with a call to it : UNFOLD performs the reverse transformation (but note that it is not the reverse of the FOLD rule). Hope syntax is apparent within the program patterns.

For example, the equation '---f(x)<=x+2;' might be transformed by FOLD to '---f(x)<=g(x)+2;' in conjunction with '---g(x)<=x;'.

```
 if    an equation(1) has                    if   an equation(1) has
  &     code = '---Lhs1<=Expr;'               &    code = '---Lhs1<=Rhs1;'
  &    an equation(2) has                     &    Rhs1 = '__Op(X)__'
  &     code = '---Lhs2<=Rhs;'                &   an equation(2) has
  &     Rhs = '__Expr__'                      &    code = '---Op(X)<=Rhs2;'
then      equation(2) has                   then     equation(1) has
  &     Rhs = '__Lhs1__'                      &     Rhs1 = '__Rhs2__'
  &     code = '---Lhs2<=Rhs;'                &    code = '---Lhs1<=Rhs1;'

          (a) FOLD                                 (b) UNFOLD
```

Figure 3.12 Some Higher level Transformations.

After each transformation is read by the system a table is presented of pattern-match variables and the sorts of the abstract syntax they will match. The user then has three options : either accept these sorts, force a reparse of the rule (eg. to parse a pattern variable to match with one parameter instead of a list of parameters) or name more specific sorts which the pattern variable must match with. This procedure, we name TYPE QUALIFICATION, allows the user to resolve ambiguities arising from patterns which have more than one interpretation. The mechanism introduces use of internal descriptions of the programming language and fulfills the same function as the use of sort information in VRL and POPART rules.

For example, once the APPEAL editor has read the FOLD transformation, the table of figure 3.13 is presented to the user. Sorts associated with each variable are the most general which can be matched, given the pattern in which variables were found. If he replies 'yes' then the metavariables would match with the sorts shown. Typing 'reparse' will change those variables matching sequences of sorts (eg.parameter lists) to some single sort. The final choice is to enter more specific sorts to be matched by the variable.

Variable	Match Type
Lhs1	sequence_of cl_expression
Expr	cl_expression
Lhs2	sequence_of cl_expression
Rhs	cl_expression

Figure 3.13 Table of Variables and Matching Types.

Matching and substitution with all occurrences of objects or code is achieved with use of the universal quantifier 'all'. For example, the rule in figure 3.14 is a version of INSTANTIATION used to substitute some variable identifier with another throughout an equation. Similar to 'Replace-all' in POPART, the 'all' construct is used in this rule to signify that substitutions will be made everywhere the pattern is located in the equation. A rule may also be quantified over all objects in some blackboard context using syntax 'all <OBJECT>(s) have'. Note also use of negation with keyword 'not' in figure 3.14.

For example, the equation '---f(x)<=x+2;' might be transformed by INSTANTIATION to '---f(y)<=y+2;' in conjunction with equation '---h(y)<=z;'.

```
  if    an equation(1) has
   &     code = '---Op1(X)<=Rhs1;'
   &     all(Rhs1 = '_X_')
   &    an equation(2) has
   &     code = '---Op2(Y)<=Rhs2;'
   &     not( Y = X )
then       equation(1) has
   &     all(Rhs1 = '_Y_')
   &     code = '---Op1(Y)<=Rhs1;'
```

Figure 3.14 Version of INSTANTIATION in APPEAL.

Referring to the stack example of the last chapter, the rules thus far described can be used to transform equations relating operations of the abstract and representing types (so called 'abstraction mappings') to equations of an implementation. Referring to the equations in figure 2.5 (page 40), 'h(top(nil))<=top(h(nil));' can be transformed with the axioms to the equation of the implementation in the following sequence:

```
h(top(nil))<= top(h(nil)

           <= top(newstack)     by UNFOLD

           <= undef             by UNFOLD

           <= h(error)          by FOLD
```

Finally drop 'h' both sides,

'---top(nil)<= error;'.

3.4.2 APPEAL Rules for use with multiple representations.

Two final examples of rules are given to illustrate how a representation might be chosen for an abstract object specified algebraically. The APE system (page 5) attempts to characterise the actions of operations on the abstract data type (eg.'stack') then chooses a data type with a similar set of operations as the representation. Operations typically read, write, delete or take other actions with respect to elements at some location within an abstract object and these can be inferred by pattern-matching. Figure 3.15(a) identifies READ's operations. Figure 3.15(b) is a rule borrowed from the APE system to choose a linked list as a representation. Note that these rules must match with objects on more than one blackboard level and that pattern 'D(P)' will match with 'stack(alpha)' for the stack data type (see figure 2.5, page 40).

```
  if    a datatype(1) has
   &     type = 'D(P)'
   &    an operation(1) has
   &     signature = 'dec Op:Sorts->P;'
   &     Sorts = '__D(P)__'
then    an operation(1) has
   &     action = reads
```

(a) READS rule.

```
  if    a datatype(1) has
   &     type = 'D(P)'
   &    all operation(s) have
   &     signature = '__D(P)__'
   &     actposn = front
then    an object(1) has
   &    represent = 'sl_list(P)'
```

(b) Choosing a representation.

Figure 3.15. Examples of APE rules in APPEAL.

A formal definition of the rule language in Extended Backus-Naur Form is given in figure 3.16. Readers are reminded that the metasymbols have the following meaning : '|' can be read as 'or'; '{...} enclose items which may be repeated or not be present; "" enclose string literals. Terminals 'object-class' and 'program-syntax' are the blackboard level name and token strings of the application language, respectively.

```
rule = "if" premise "then" conclusion.
premise = object-template {"&" object-template}.
conclusion = object-template {"&" object-template}.
object-template = universal-template | existential-template.
universal-template = universal-quantifier object-class
                          "(s)" "have" clauses.
existential-template = existential-quantifier object-class
                          object-refno "has" clauses.
clauses = "&" clause {"&" clause}.
universal-quantifier = "all".
existential-quantifier = "a"|"an"| .
object-refno = "(" integer ")".
clause = program "=" expression |
         "all(" program = expression ")" |
         "not(" program = expression ")".
program = object-attribute | metavariable.
expression = "'" pattern "'" | "'__" pattern "__'".
pattern = program-syntax {pattern} |
          metavariable {pattern}.
```

Figure 3.16 EBNF Definition of APPEAL Syntax.

This definition excludes extensions of the APPEAL language used to refer to or control the application of, APPEAL rules. Issues of control and its representation are addressed in the next chapter.

3.5 Summary.

Some requirements of a representation for programming knowledge were noted. The representation should be based on rules which use patterns to effect code substitution. The representation should be language independent and neutral with respect to analysis and synthesis. The author's prior experience of using rules with programming tools suggests that they should also be accessible and powerful enough to represent many programming problems. Rule representations are also attractive given the accumulated experience of their use in knowledge-based systems and their potential for machine-driven explanation, teaching and learning.

A study of some representation languages showed that languages using metalanguage expressions to match token strings of programs were perspicuous. This is not surprising given that programming languages provide the basis of communication of programs between programmers and their machines. Abstract descriptions of programs need not be prominent in such representation languages since rules can be compiled to an internal form which is potentially language independent. We noted also that some languages such as ARL have simple, easy-to-use syntax for describing rule predicates.

The APPEAL language has been designed to meet these requirements and incorporate the best features of the representation languages studied. Some equivalent language constructs of VRL and POPART are adopted, particularly the use of metalanguage expressions to effect code substitution. However, APPEAL is unique in its treatment of programs as objects. Use of ARL-like syntax within locally numbered templates give the language a symmetry not observed in VRL nor POPART. In no other knowledge representation language, known to the author, are constraints on matching sorts of metavariables expressed by a separate process to preserve symmetry. The symmetry of some transformation rules is apparent when expressed in APPEAL and they could be evaluated for the purposes of analysis or synthesis. Again, use of the object concept brings transparency to the representation with no loss in sophistication.

CHAPTER 4.

A TASK-ORIENTATED CONTROL MECHANISM FOR PROGRAM REFINEMENT.

The discipline of software engineering not only provides high-level notations for describing systems but numerous methods for constructing systems using these. Methods complement our knowledge of programming 'specifics' by providing a framework in which to order problem solving activities. Concerned with strategy or control, methods bring efficiency to programming and, in addition, reduce the scope for error.

The representation of strategic or control knowledge within knowledge-based software tools appears to be particularly relevant. According to Wile [Wile82b], "Enormous chains of primitive transformation applications are necessary to optimise even the most trivial specification." The application of refinements must be controlled, since "for realistic applications, enormous numbers of transformations, transformation applications, intermediate program states, etc., must be dealt with quickly. This makes size the most crucial problem to be solved."

We noted in chapter 1 that application of low-level transformations, to condition programs for major refinement, has merited particular attention (eg. in PROUST and TI). A difficulty is that these transformations can be applied to programs in a multitude of ways (eg. consider the many possibilities for commuting, re-associating or re-distributing terms of expression '2x+3(y+x)'). Thus many programs may be generated from any given program, or in tree-search terms, the 'branching factor' [Nilsson80] is high.

The choice of control mechanism for our interpretation of the blackboard architecture should, therefore, be orientated towards the representation and use of control knowledge. Of particular concern is that this knowledge be explicit and accessible within a blackboard system. A high-level, object-centred language was developed for expressing programming knowledge. Similar aims should guide us towards a usable language for expressing control in a knowledge-based programming environment.

To begin this chapter, we examine some representations for control knowledge in knowledge-based systems. Attempts to control use of MYCIN's medical knowledge base for purposes of explanation and tutoring have been particularly fruitful of ideas. Since control knowledge is complementary to domain knowledge, techniques for representing the former mirror those used with domain knowledge. Use of rules is the favoured approach. We describe these efforts and subsequent work to provide knowledge-based control within blackboard systems.

Hitherto, designers of knowledge-based programming tools have tended to use any device available within their systems to control the search for solutions. However, the relevance of control issues to the programming domain is clearly seen by Wile who has developed a language, known as PADDLE, for representing strategic knowledge of programming. We examine this language closely.

The author's interpretation of the blackboard architecture is complete with a choice of control mechanism. A control mechanism has been developed which uses four classes of object. Methods are represented as sets of control rules, each expressed via an extended

APPEAL language. Control rules generate networks of objects describing 'tasks' within a separate control blackboard. Tasks are used to direct activity on the domain blackboard. Control will be seen to be explicit and accessible using methods, control rules, tasks and a control blackboard.

4.1 Knowledge-based Control in some Knowledge-based Systems.

Much of the pioneering work in representing control knowledge was aimed at improving the efficiency of MYCIN and making its knowledge base available for other purposes. We examine the control mechanisms of two developments of MYCIN : the TEIRESIAS and NEOMYCIN systems.

The realisation that strategies are essentially hierarchical motivated investigations of their representation in blackboard systems. The concept of 'control blackboard' introduced for HEARSAY-III, is taken further in OPM, where a general-purpose control mechanism has been developed.

4.1.1 TEIRESIAS.

Explicit representation and use of control knowledge has been investigated by Davis [Davis80a], in the Teiresias system. He suggests that its role is to decide which knowledge (of MYCIN) to invoke next in a situation where more than one chunk of knowledge may be applicable. Faced with a set of alternatives, possibly so large that exhaustive invocation becomes infeasible, some decision must be made about which should be chosen.

Davis represents control knowledge as 'metarules' which have the following properties:

(1) They refine the set of applicable domain rules. The set is pruned or reordered to indicate the preferred rules to be applied. (Viewed in tree-search terms, metarules either prune the search space or reorder branches of the tree).

(2) They contain descriptions of domain rules. Metarules select amongst applicable domain rules by referring to their properties.

(3) They should be encoded in constructs accessible to the program. Further, if a uniform encoding scheme is adopted for both control and domain knowledge, then the same access, inference, acquisition and explanation procedures, developed for domain rules, can be extended to metarules as well.

(4) They may also reason about control at the metalevel. That is, the system retrieves the list (L) of domain level rules. Before invoking these, it checks for a list (L′) of first order metarules to reorder or prune L. But before invoking this, it checks for second order metarules to prune L′, etc. Recursion stops when there is no rule set of the next higher order, and the process unwinds, each level of strategies advising on the use of the next lower level.

In an elaboration of (2), Davis suggests that metarules select amongst plausibly useful domain level rules by direct examination of their content. The technique, named 'content referencing' [Davis80b], requires that the source code of domain rules be examined at execution time. That is, the metarule "goes in and looks" for the relevant characteristic in these rules. The technique is particularly suited for large, evolving knowledge bases since new metarules and domain rules can be added without reviewing all existing rules.

In contrast, reference to rules via 'external descriptors' (ie. names, main effects, side effects, etc.), attached to them, compromises the ability of the knowledge base to accommodate changes. After a domain level rule has been edited, for instance, we would have to check all the metarules referring to its descriptors to ensure all such references are applicable to the revised rule. Similar problems arise with the modification of metarules. Further, the set of external descriptors is potentially large and difficult to define a priori.

Metarules bring both a conceptually cleaner organisation and conceptually clearer use of strategic knowledge. Davis argues, "..metarules offer a means of explicit representation of the decision criteria used by the system to select its course of action. Subsequent 'playback' of this criteria can provide a form of explanation of the motivation for system behaviour. That behaviour is also more easily modified, since the information on which it is based is both clear (since it is explicit) and retrievable (since it is accessible). Finally, more of the systems behaviour becomes open to examination, especially by the system itself".

Metarules are, however, deficient in a number of respects. Firstly, this representation views strategies wholly as operations on domain rules. Thus 'focus of attention' on particular partial solutions cannot be controlled. The system cannot be directed to work with contexts or data items worthy of attention. Secondly, metarules generally encode 'fixed' strategies for solving some problems. Domain rules tend to be refined by metarules in a regular fashion irrespective of the particular problem. Both limitations make the representation inflexible and unresponsive to new data arising during inference.

4.1.2 NEOMYCIN.

The aim with NEOMYCIN was to reconfigure the knowledge base of MYCIN (including the metaknowledge contributed by Davis) for application to teaching [Clancey&Letsinger81]. However, Clancey found the knowledge too narrow for this purpose. Rather, it was necessary to articulate the hierarchical organisations of knowledge and search strategies that humans found useful.

Of interest here, is NEOMYCIN's control mechanism, described as "..forward, non-exhaustive reasoning of a space of hypotheses that makes it reason more like a human". The diagnostic strategy of MYCIN, revealed by protocol analysis of medical experts, is recast as a task posing activity. NEOMYCIN's strategy is structured in terms of "tasks", which correspond to metalevel goals and subgoals. An ordered collection of metarules constitutes a procedure for achieving a task. NEOMYCIN uses hierarchical metarules with the following properties :

(1) Metarules encode methods of achieving tasks by invoking other tasks, or invoke the base-level interpreter to apply selected domain rules.

(2) The metarules associated with a task may describe the sequence of steps used to achieve the task (in which case the applicable rules are applied once in order.)

(3) Metarules may represent alternate strategies for achieving the goal (in which case the preferentially ordered rules are executed until the goal of the task is achieved). Figure 4.1 (from [Hasling etal84]) shows an abstraction of a task and its metarules.

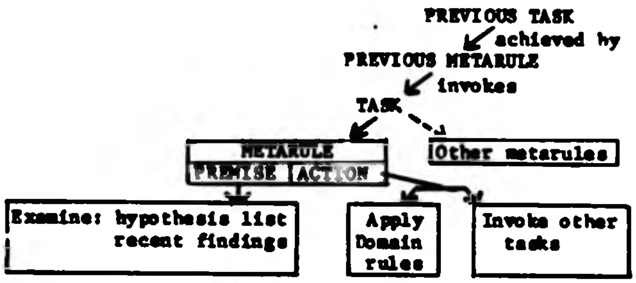


Figure 4.1 Abstraction of a task and its metarules.

(4) The rules for a given task are treated as a pure production system (ie. they are repeatedly tried in order, returning to the head of the list when one succeeds, stopping when no rule succeeds or an end condition is true).

(5) The main use of this feature is to allow refocusing when new data changes the state of the search space, as well as non-exhaustive consideration of hypotheses.

NEOMYCIN's control mechanism offers a number of advantages over that of TEIRESIAS:

(1) The task structure generated by NEOMYCIN metarules is an explicit representation of the decisions, or plan, for problem solving since application of selected domain rules is conditional on tasks generated. No such plan is generated by TEIRESIAS metarules.

(2) NEOMYCIN's metarules can be used to focus on particular objects in the hypothesis space (ie search tree). Thus domain rules can be directed to work with a particular part of a problem.

(3) As a corollary of (2), and the feature which returns metarule invocation to the head of the list, once the hypothesis space is changed, the strategy can react opportunistically to promising new data.

(4) Since metarules are used to invoke domain rules directly, not all domain rules may be tried. Therefore a solution may never be found (given adequate domain knowledge) if the strategy is inadequate.

Implementation details of NEOMYCIN's control mechanism are sketchy but metarules appear to be indexed with sets of domain rules. Thus rule dependencies, of the sort frowned on by Davis, will be in existence. The refocusing behaviour of the control mechanism requires that the system frequently move its attention among alternative methods of achieving a task. This suggests that the task structure be recorded using a context mechanism. There is no indication that NEOMYCIN uses one.

4.2 Knowledge-based Control in some Blackboard Systems.

The control mechanism of AGE is factored into a number of steps. System builders assemble a control mechanism for their particular application by supplying rules or code to guide control in each step. The four steps of the cycle are;

(1) A recent change in the hypothesis structure is chosen as focus.

(2) A knowledge source is selected which has its precondition satisfied.

(3) The activated knowledge source is executed or 'fired', resulting in a change to the hypothesis structure.

(4) Finally, a test is made to determine if the solution is found.

An AGE-like control scheme was adopted in the author's previous blackboard systems. A rule interpreter was supplied to implement the third step; hooks were provided to attach code for the first, second and last steps.

Control is effected in HEARSAY-III using a 'scheduling blackboard'. Scheduling knowledge sources establish priorities, recorded as units, for selecting instantiations of knowledge sources. On each cycle, the highest priority knowledge source is executed.

OPM, a multiple-task planning system, was designed to investigate the explicit representation and use of control knowledge within the blackboard architecture. A model of control [Hayes-Roth85] was developed for the system whereby knowledge sources, representing control knowledge, could generate a plan on a control blackboard for guiding the development of the solution on the domain blackboard.

Solution elements are generated on a six-level control blackboard. These describe the priorities that should be given to applying certain rules (both domain and control), to certain elements, at periods during the problem solving process. Figure 4.2 illustrates the control blackboard of OPM with horizontal axis representing time (in control cycles) devoted to the problem. The network of elements describe an aggregation hierarchy of tasks on these levels such that PROBLEM elements summarise the problem to be solved; STRATEGY elements describe general guidelines; FOCUS elements describe levels of the domain blackboard which should receive attention during certain problem-solving episodes; POLICY elements describe rules or domain elements which should receive attention during particular cycles; AGENDA and KSAR elements describe instantiations of rules and which of these fired, respectively.

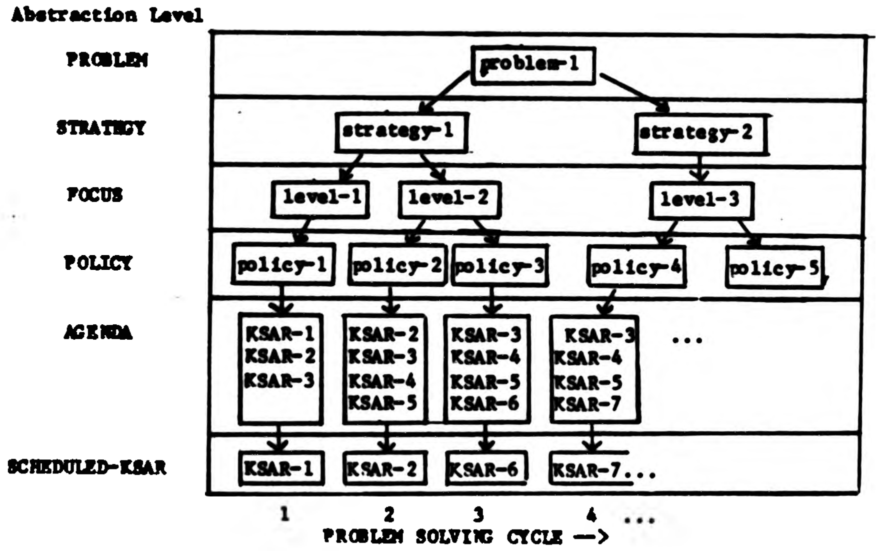


Figure 4.2. The Control Blackboard for Controlling OPM.

The control cycle is established by three knowledge sources acting in rota. An AGENDA-UPDATER finds rules which can be instantiated with recent blackboard changes and records these instantiations (also known as 'activation records' or 'KSARS') in a list at the AGENDA level of the control blackboard. A second knowledge source, the SCHEDULER, selects an activation using an evaluation function invoked with information contained in elements at the FOCUS and POLICY level for that period of problem solving. Finally, an INTERPRETER executes the activation and records it at the lowest, SCHEDULED-KSAR, level with pointers to resulting changes on either blackboard.

The control model is general purpose and highly adaptive. Control rules can change the priorities used for their own selection during problem solving (ie. new strategies can be tried if no solution is emerging or time to find one is running out). Since control rules change agendas, structures vital to the systems own control, OPM is a self-programming system. Control rules can also induce new plans based on observations about which rules actually produce solutions.

The most abstract of the control mechanisms so far studied, its adaptability and power derives from manipulation of its own data structures. Plans generated on the control blackboard are recipes for modifying agendas.

4.3 PADDLE : A Language for Controlling Program Refinement.

PADDLE is a language for expressing the optimisation strategy (or 'development structure') to be pursued in transforming a specification to an implementation [Wile82b]. The development structure, applied to a specification to produce an implementation, gives rise to a 'program development' which is a formal document explaining that implementation.

Thus PADDLE is not only used to control the optimisation process but also record the development in a form that can be replayed on other specifications or, if possible, changed to accommodate other specifications.

PADDLE is used for structuring goals which capture the strategies and design decisions made in the implementation. To be useful such goal structures are parameterizable. A goal may be defined using a "command" statement. Like procedures of structured languages, commands consist of a name, a set of parameters and a body.

Goals may be invoked within the body of commands using the following mechanisms to convey dependencies between them:

(1) sequential dependency (ie. composition) is expressed by enclosing the sequence of goals or stubs within a begin/end pair. For example, 'begin firstgoal; secondgoal end.'

(2) stubs are placeholders for other goal structures. Their refinement is defined with use of the reserved word 'by', (eg. 'mystub by <PADDLF command>).

(3) Goal independence is indicated using the 'and' reserved word. For example, 'A and B' declares that goal A may be achievable in parallel with goal B.

(4) goal choice is indicated by enclosing a list of goals in the 'choose from ...end' construct. The implementor decides in each situation what the appropriate selection should be.

(5) Conditional structures, expressed with the 'first of' construct, is used to make automatic choices. For example, 'first of A;B;C end', will try each goal in turn until one of A, B or C succeed, this being also the one chosen.

(6) Repetitive Goals are established using a loop goal structure that enables the body to be executed (achieved) repeatedly. For example, 'while A do {B, C}' will repeatedly execute B and C while A succeeds.

(7) leaves of goal structures are 'primitive' commands to execute transformations on programs.

The overall model of program manipulation used by PADDLE is as follows: there is at all times a specification/program affected by PADDLE expressions. This specification/program, together with the active goal structure(s), forms the data and control portion of the "abstract PADDLE machine" state. The development structure is applied to an initial state to produce a new state. That application is a relatively straightforward interpretation of the development language as though it itself were a programming language. In particular, it is a depth-first, left-to-right tree traversal of the goal structure represented by the development.

PADDLE goal structures therefore resemble the task structures generated within NEOMYCIN. However, there is no provision within PADDLE to focus the transformation process on particular portions of the specification. According to Wile, this shortcoming has given rise to errors when attempting to replay developments on specifications.

4.4 A Task-orientated Mechanism for Blackboard Control.

The issue of accessibility determined the choice of concepts (from the above control mechanisms) to be included in the author's control mechanism. An object-centred control mechanism was devised with the overall aim of incorporating 'development structures', similar to Wile's, within a flexible task-orientated framework suggested by NEOMYCIN.

The ability to focus, provided by the NEOMYCIN mechanism, is attractive for our purposes. The refinement process, as envisaged in chapter 2, will produce a tree of specifications (see figure 2.2, page 23) on the blackboard. Thus control knowledge should be used, not only for selecting the refinement(s) to apply, but also select the specification to refined next.

While the OPM control mechanism includes all these features and more, its use of internal data structures, mitigates against its complete adoption. Explicit reference must be made to data structures supporting control (eg. rule activations, agendas, evaluation functions and control cycles). These structures are generally unfamiliar to software engineers. In contrast, control is made accessible if framed entirely in terms of objects contributed by users.

Agendas are also inappropriate given that the branching of search trees is likely to be wide. (Considerable evaluation will be necessary to fill agendas at each step). Direct, non-exhaustive application of domain rules, as exhibited by NEOMYCIN and PADDLE, is desirable for our purpose.

The basic objects of the author's control mechanism are therefore tasks and control rules. Since task structures, representing problem-solving plans, are hierarchical, they are conveniently recorded on a control blackboard. Control rules, representing strategic knowledge, both generate and execute tasks on this blackboard. We describe the blackboard and control rules with their associated object classes.

4.4.1 The Control Blackboard.

The basic element of the control blackboard is the task. (Like Clancey, we believe the concept conveyed by word 'task' is more familiar to domain experts than that of 'goal'. It is also more general since it can convey the notion of 'action to be taken'). Like elements of the domain blackboard, tasks have an arbitrary number of attributes (one of which is the 'name'). They can be used to record information appropriate to the particular problem, so mimic the parameterisation of PADDLE goal structures. Plans for applying programming knowledge consist of tasks in the following classes:

(1) CONTEXTS. Like goal composition in PADDLE, all tasks in a context must be achieved. (There is no construct to indicate that tasks are achievable in parallel).

(2) LEVELS. This class distinguishes the level of representation within the aggregation hierarchy of tasks. Like levels of the domain blackboard, they have a name and precedence.

(3) CONTROL BLACKBOARD. Consisting of one or more levels, the control blackboard also records the development structure. No fixed levelling is in force (as with OPM), but is chosen for convenience.

However, the bottom-most level (lowest precedence) is named 'rule' and is considered to be the domain knowledge base.

For purposes of access and uniformity, each domain rule is considered to be an object with attributes which depend on the particular domain blackboard in use. Each attribute describes clauses of the rule intended to match with, or act on objects at a given level of the domain blackboard and the naming convention is '<LEVELNAME>-conds' or '<LEVELNAME>-actions' respectively. For example, the simple rule of figure 3.11a (page 60) contains a single clause in each attribute 'equation-conds' and 'equation-actions'. Other attributes of this rule (eg. 'object-conds') are empty lists.

Task structures are maintained using the same ABSTRACTION and REFINEMENT links as available for the domain blackboard. Abstraction links are directed from a context on one blackboard level, to a context at the next lower level. They indicate decomposition of tasks between the contexts. Task refinement normally takes this form, thus REFINEMENT links will be used rarely.

4.4.2 Programming Methods.

Programming methods are represented as sets of control rules which both generate the task structure and execute tasks contained therein. Rather than index domain rules with tasks (as for NEOMYCIN), we have adopted TEIRESIAS' technique of content referencing to determine domain rules to be applied for some task. There is therefore two flavours of control rules : TASK GENERATORS are used to grow the task structure; TASK EXECUTORS are used to apply domain rules to complete some task.

The underlying stack organisation plays a key role in this control mechanism. Both preferential ordering of control rules and system focus is achieved using it.

Since APPEAL was designed for manipulating objects and rules are objects, it is well suited for expressing control knowledge. Metalanguage expressions, within APPEAL control rules, are used to reference APPEAL domain rules by their content. (Minor extensions are necessary in APPEAL syntax to describe rule components and invoke rule application).

The automatic programming problem described in chapter 2 (page 36) can be used to illustrate the representation of methods by APPEAL rules. A four step metaplan for implementing the stack specification was described (page 38). In brief, we have to characterise the operations, choose representing types for the stack, generate specification rules relating operations of the stack to those of the representation, then transform these to equations of the implementation.

The TASK GENERATORS of figure 4.3 (a) and (c) will grow a task structure on a four-level control blackboard with levels named PROBLEM, STRATEGY, TACTIC and of course RULE. (Note that tasks so generated will not be parameterised since task names alone are used. This simple example is chosen for clarity but note that chaining of rules together by name is not in the spirit of the blackboard architecture).

The task generators of figure 4.3 (b) and (d) describe an alternative method for solving the problem, namely, that of pursuing a "blind" depth-first search. Note that the alternative strategies expressed in rules 4.3(a) and (b) mimic the 'choose from' construct in PADDLE. Further, goal execution is similar if rule 4.3(a) is higher in the stack.

```
 if    a problem(1) has                    if    a problem(1) has
  &    name=implement_object                &     name = implement_object
then   a strategy(1) has                  then   a strategy(1) has
  &    name = analyse&represent             &     name = tryanyrule

             (a)                                        (b)

 if    a strategy(1) has                   if    a strategy(1) has
  &    name = analyse&represent             &     name = tryanyrule
then   a tactic(1) has                    then   a tactic(1) has
  &     name = findactions                  &     name = depthfirstsearch
  &    a tactic(2) has
  &     name = represent                                (d)
  &    a tactic(3) has
  &     name = implement

             (c)
```

Figure 4.3 Task Generators of alternative methods.

Control rules which are task executors both specify the domain object(s) that should receive attention and the form of domain rules that might be applied. As a stack organisation is employed, the first matching transformation rule will be tried with the most recent matching object added to the stack.

Metalanguage expressions are used to reference these rules by use of patterns intended to match clauses in the relevant attribute. For example, the pattern, '__action = reads__', is to be found in the 'operation-actions' attribute of rule 3.15(a) (page 65) so it will be tried by rule 4.4(b).

The 'depth-first' method (ie. rules 4.3(b),(d) and 4.4(b)) might be called an uniformed method for finding a solution. Only if no cycles of identical contexts are permitted and no randomly named objects generated, will the search be exhaustive. However, refinement generally requires object generation at new blackboard levels so this will be difficult to arrange. The rule of figure 4.4(b) might direct depth-first search on the operation level, but similar task executors will be necessary for other levels of the domain blackboard which use expressions guaranteed to match all relevant domain rules.

```
  if   a tactic(l) has
   &    name = findactions
   &   an operation(l) has
   &    action = unknown
   &   a  rule(l)  has
   &   operation-actions='__action=A__'
then    tryrule(l)

                 (a)
```

```
  if   a tactic(l) has
   &    name = depthfirstsearch
   &   an operation(l) has
   &    signature = S
   &   a  rule(l) has
   &    operation-conds='__signature=S__'
then      tryrule(l)

                 (b)
```

Figure 4.4 Examples of task executors in alternative methods.

4.4.3 Control Cycle of the APPEAL machine.

The control cycle of the abstract APPEAL machine attempts to combine the depth-first, left-to-right tree traversal of the PADDLE machine with the refocusing behaviour of NEOMYCIN. Rather than employ a pure production system of individual control rules, as in NEOMYCIN, we use a similar organisation with sets of control rules, where rules in each set take their action at a given level of the control blackboard. The intention is that the control 'drives' the generation of tasks on levels of descending precedence. Task executors then change the domain blackboard and the cycle restarts.

A control cycle so implemented might work with one method (the preferably ordered control rules) before considering another. In practice, it has been necessary to augment this procedure by allowing back up to previous levels if task generation fails. Without back up, the system attends to an alternative method before the first is fully explored. Further details of the control cycle are reserved for chapter 7, where a formal description will be given.

Figure 4.5 shows the net of tasks that might be created in a control blackboard by the rules of figure 4.3. The curved line indicates the order in which tasks receive attention after the domain blackboard is altered.

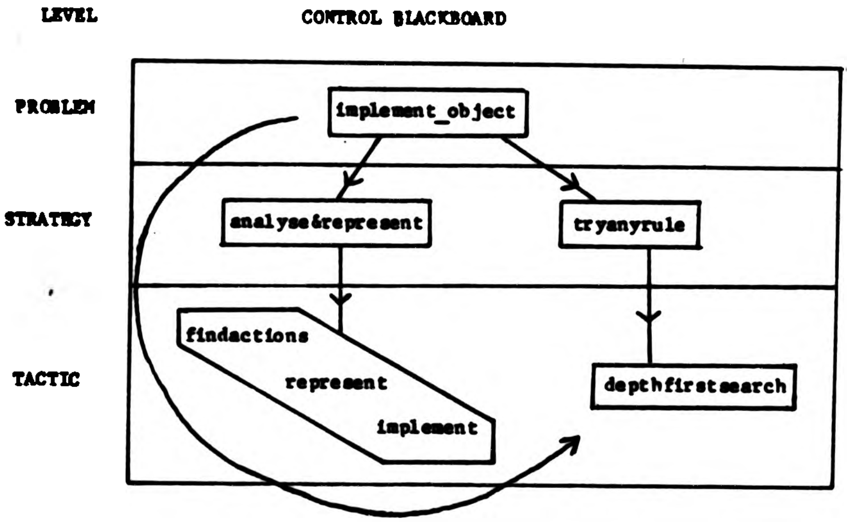


Figure 4.5. Control Blackboard for Programming Tool.

In contrast to the control of OPM, the model developed here for applications in programming is not suited for machine learning of rules or reprogramming its own control structures. Control rules are considered to exist outwith the blackboards and do not reference each other. Nonetheless, one sort of opportunism, provided within OPM, is available by virtue of the control cycles refocusing behaviour. The syntax of APPEAL is extended for expressing control knowledge with the EBNF grammar rules of figure 4.6, which replace corresponding rules of figure 3.16 (page 66).

```
conclusion = object-template {"&" object-template} |
             "tryrule" object-refno.
program = object-attribute | metavariable | rule-attribute.
rule-attribute = object-class "-conds" | object-class "-actions".
```

Figure 4.6. Supplementary grammar rules of APPEAL.

4.5 Summary.

It is desirable that a knowledge-based programming environment provide a representation for strategic knowledge[D given the enormity of search trees. We surveyed representations for control knowledge used in the wider body of knowledge-based systems. Many concepts can be drawn from these systems in the design of a control mechanism for knowledge-based programming within a blackboard system.

The primary aim in the design of this mechanism is that it be accessible for knowledge-based control by software engineers. Thus a guiding principle in the design was that only objects contributed by the engineer be used to express control. The abstract control model adopted in OPM was therefore not adopted in toto. Rather, concepts compatible with this principle, such as content referencing, control rules, tasks and a multi-level blackboard, provide a framework for a new control mechanism.

A novel control mechanism for knowledge-based programming within blackboard systems was described. This is the first control scheme for blackboard systems known to the author which permits knowledge-based control of search and opportunism without use of agendas. Nor is the author aware of any high-level knowledge representation language for use in blackboard systems which permits content referencing.

The new control mechanism is compatible with the object representations developed for other components of the blackboard architecture. Indeed, the object description of rules devised for content referencing brings uniformity to these representations. The language for expressing domain knowledge is of immediate use for describing control knowledge and the control blackboard mirrors the problem-solving behaviour of the domain blackboard.

CHAPTER 5.

A PROLOG IMPLEMENTATION OF THE BLACKBOARD ARCHITECTURE.

In this chapter a PROLOG implementation of the blackboard architecture is described. Since there have been few attempts to use PROLOG as the implementation language of a blackboard system, this effort is likely to be of academic interest so a detailed account is given of the design. As an implementation of the architecture (rather than an interpretation for a particular application) this software is designed to have wider applicability. Thus the management of blackboards are the principal concern, not the representation of knowledge sources nor search strategies since a variety of these can be interfaced to the system.

The blackboard manager of ENCOPES is a complete re-implementation of the author's earlier work [McArthur86]. The previous system was designed with the assumption that many sophisticated reasoning processes could be recorded with networks of frame-like objects, similar to those of the ACE system (see page 29). However, pointers were too dense in ACE-like networks and it was difficult to manage alternative lines of reasoning.

The multi-attribute objects of ACE are also too 'large-grained' for reasoning with large structured items such as programs. We do not wish to record programs in their entirety every time a small change is made. Rather, blackboards should find a representation as small-grained data units and include an inheritance mechanism for economical storage of information.

Details of the ART system (see page 31) inspired the current work. A clear distinction is made in ART between monotonic reasoning (for analytical applications) and non-monotonic reasoning (for applications in design). A mechanism is also included for locating consistent descriptions of partial solutions at different blackboard levels. The system described here uses similar representations for non-monotonic reasoning but no attempt has been made to replicate ART's truth maintenance mechanism.

The chapter provides a bottom-up description of the implementation. The next section discusses the major design choices and problems faced by the knowledge engineer when choosing PROLOG as the implementation language for large IKBS. A PROLOG tool is described which permits modular development of large systems. As the blackboard data type is composed from a number of simpler types, these simpler types are introduced before the blackboard manager itself. Finally an interface to the manager, known as ACSYS ('Acquisition Subsystem'), for defining blackboard schemas, is described, as is the construction of configuration tools for particular applications in knowledge-based programming.

5.1 PROLOG and Knowledge-Based Systems.

PROLOG is a declarative language based on predicate logic [Kowalski79]. The language provides constructs for manipulating symbolic data structures and a powerful pattern-match facility based on unification [Robinson65]. Recent publication of numerous language texts [Bratko86][Sterling&Shapiro86] testifies to PROLOG's growing popularity and emergence from the research laboratory for serious commercial use.

Evaluations of PROLOG as a language for building IKBS have been published in recent years [Hammond80][Clark&McCabe82]. Attempts to implement general-purpose blackboard systems in PROLOG are reported by [Jones etal86] and [Noordzij87]. Two approaches to system design using PROLOG are discernible : the first might be termed the 'logic programming approach'; the second, termed the 'symbolic programming approach'.

The logic programming approach is to represent knowledge as rules in the clausal form expected of all PROLOG procedures and use the theorem proving evaluation of these rules with some query to perform the search. The resulting back-chaining of PROLOG rules is similar to the search behaviour of the MYCIN system [Shortliffe76]. Indeed, facilities to provide explanation [Hammond&Sergot83], and question the user [Sergot83] about subgoals , have been developed to effectively transform PROLOG into an expert system shell (for example, the APES system [Hammond83]).

The symbolic programming approach is to ignore the niceties of logic programming, for which PROLOG is in any case deficient, and write PROLOG systems in the style of LISP. Knowledge is represented as some appropriate PROLOG data structure and an inference engine, using some chosen search method, is written. This approach is typically used where the format of PROLOG clauses is inappropriate for encoding domain knowledge, or where goal-directed search is not desired.

The symbolic programming approach is preferred for PROLOG implementations of the blackboard architecture. A major consideration is that the blackboard should be a global data structure. The logic programming approach frowns on the alteration of global data and would require that the structure representing the blackboard be passed to virtually every procedure as a parameter. The symbolic programming approach also permits the flexibility of knowledge representation and search strategy inherent in the architecture.

Another problem arises with PROLOG when attempting to use it for implementing large systems. With few exceptions, most PROLOG systems do not support the 'module' construct. Some substitute was required so a special PROLOG tool has been developed which permits overlaying of small programs written in the common or 'core' PROLOG [Clocksin&Mellish81]. This tool influences considerably the PROLOG writer's style of programming so it is briefly introduced as a prelude to the description of the blackboard implementation.

5.2 MODULE : a Tool for Constructing Large Modular PROLOG Systems.

The MODULE tool provides a means for stating usage dependencies between software components and a means of loading or unloading a component with all its dependents. Further, loading or unloading can be invoked at runtime for the purposes of overlaying. A full treatment of the MODULE is given in APPENDIX-A.

In brief, MODULE requires that each PROLOG module has an assertion as to the predicates 'exported', or available for use in other modules, and assertions naming modules that are 'used' by this module. Modules have the format of Figure 5.1. Note that the module name is synonymous with the name of the file on which the module resides.

```
export(<MODULE NAME>,[<FUNCTOR1(ARITY1),FUNCTOR2(ARITY2),..>]).

uses(<MODULE NAME>,<OTHER MODULE1>).
uses(<MODULE NAME>,<OTHER MODULE2>).
...etc..
uses(<MODULE NAME>,<OTHER MODULEn>).

<PROLOG CODE ASSOCIATED WITH FUNCTOR1>.
<PROLOG CODE ASSOCIATED WITH FUNCTOR2>
..ETC....
```

Figure 5.1. Module Format.

To use the tool, MODULE is first consulted into the PROLOG workspace. A module (and its dependents) can now be loaded with the query,

'load([<MODULE NAME>]).'.

The code in file '<MODULE NAME>' will be 'consulted' as will all other files which are used by these, etc., until the complete source of the application is 'consulted' or loaded.

The code for an application need not all reside in the PROLOG workspace at once. The 'load' procedure may be called from within the body of some PPOLOG procedure at run-time thereby loading further code dynamically. Calls may be made to this new code followed by a call to 'unload' with the named module(s). All the predicates will be 'retracted' but the workspace is not freed until we backtrack past the first 'load' call. This is achieved by enclosing the 'calling' code in a 'repeat,...,fail.' loop. A 'uses' assertion is not required to reference a module which is overlayed.

Figure 5.2 illustrates how 'module1' may overlay and execute the code of 'module2', followed by the code in 'module3'. Notice that any data computed by 'module2' may be returned as an assertion which is retracted by the calling procedure. This example is trivially small but overlaying is effective if the modules are large subsystems and the PROLOG workspace is inadequate.

```
/* code for module1 */
export(module1,[process1(0), go2(1), go3(1)]).
process1:- go2(Result), go3(Result).
go2(Result):-repeat,
        ( retract(ok2(Result));
         (load([module2]), process2, unload([module2]), fail)).
go3(Result):-repeat,
        ( retract(ok3);
         (load([module3]), process3(Result), unload([module3]),fail)).

/* code for module2 */
export(module2,[process2(0)]).
process2:-asserta( ok2('prolog is fab')).

/* code for module3 */
export(module3,[process3(1)]).
process3(Result):-write(Result), asserta(ok3).
```

Figure 5.2. Overlaying Modules.

Admittedly, use of the 'module' tool leads to a coding style that departs somewhat from the style advocated by logic programming purists. However, the advantages of this tool are : large systems may be constructed from small reusable software components which can be individually tested and preferably written as abstract data types or algorithms 'using' these; and that PROLOG applications may be run on machines that severely restrict the PROLOG workspace.

With the addition of user supplied rules, a prototype tool built with ENCORES will be a very large PROLOG system. During one consultation with a tool (described in chapter 8, page 152), a large language parser (some 200 PROLOG clauses) is overlayed, before the main consultation software (about 300 clauses) is loaded. The small address space of the DEC PDP11/70 cannot accommodate both programs together, which testifies to the effectiveness of the MODULE tool.

5.3. MULTIFACTS : A Tool for Managing Triples in Multiple Databases.

The basic unit of information in ENCORES is contained in small-grained structures known as 'facts'. This unit is universal in ENCORES in so far as blackboards, blackboard schemas and even knowledge bases ultimately find their representation as facts.

Module MULTIFACTS was developed from a PROLOG prototype of the FACT machine developed at Strathclyde University [McGregor&Malone83]. Both systems record information as 4-tuples of the form,

<FACTID>,<SUBJECT>,<RELATION>,<OBJECT>.

Whereas FACT maintains these labelled triples in a single database, MULTIFACTS maintains them in multiple databases with use of a supplementary triple to designate each fact's database ownership. Both systems require the schema describing the database(s) to be represented as facts and permit deduction of facts not explicitly recorded by propagating certain properties of a set over each member.

MULTIFACTS has proved a useful tool for building other complex data types used in ENCORES. The complete code is given in APPENDIX-B. Here we describe the main operations and their behaviour using an exemplar database describing staff membership of our department. Tuples describing facts find an (obvious) assertional representation and are maintained in a stack organisation with use of PROLOG primitives 'asserta' and 'retract'. Fact identifiers are returned by operation 'next_fact' and are PROLOG terms of form 'f(<INTEGER>)'.

The main constructor operation is 'insert_fact' which takes six parameters of form,

```
insert_fact(<CODE>,<DBASEID>,[<FACTID>,<SUBJECT>,<RELATION>,<OBJECT>]).
```

The operation adds a fact to the database according to the value of CODE. FACTID is then instantiated : values of other parameters are normally supplied. Possibilities are :

(a) CODE has value 'rule'. The fact to be added is part of the database schema. The system records this by labelling the fact as a 'semantic rule'.

eg. goal, insert_fact(rule,_,[_,academic,clever_with,subject]),

results in two new assertions being pushed on the stack,

```
fact( f(2), f(1), is_a,semantic_rule).
fact( f(1), academic,clever_with,subject).
```

We can now add other facts naming specific academics as being good at specific subjects.

(b) CODE has value 'explicit'. Normal usage is implied so an identifier for the database must be supplied. For example, we might add facts to a distinguished database known as 'global' with goals,

```
insert_fact(explicit,global,[_,cs_lecturer,is_a,academic]),
insert_fact(explicit,global,[_,computer_science,is_a,subject]).
```

The result will be new assertions,

```
fact(f(6),f(5),in_dbase,global).
   fact(f(5),computer_science,is_a,subject).
fact(f(4),f(3),in_dbase,global).
fact(f(3),cs_lecturer,is_a,academic).
```

Notice that supplementary tuples are created to designate ownership of facts in particular databases. These designations alone permit ownership to multiple databases without duplicating the fact. For example the goal,

```
        insert_fact(explicit,otherdb,[_,computer_science,is_a,subject]),
```

results in one assertion being added,

```
        fact(f(7),f(5),in_dbase,otherdb).
```

This feature underlies the basic inheritance mechanism of more complex data types which will be introduced shortly.

Further set membership relations can be added such as names of individual lecturers and academic topics. The semantic rule then permits entry of facts such as 'chic, clever_with, formal_specification'. The reader might enquire if all 'cs_lecturers' are clever at a topic, say programming, then must facts linking every individual to the topic be inserted? Fortunately the retrieval mechanism assumes facts are universally quantified if either subject or object are sets and the fact is in the 'global' database.

The retrieval operation, 'fetch_fact', has the form,

```
fetch_fact(<CODE>,<DBASEID>,[<FACTID>,<SUBJECT>,<RELATION>,<OBJECT>]).
```

Most combination of bound and unbound parameters are allowed with the call to this operation. Two forms of retrieval may be used depending on the value of the first parameter.

(a) CODE has value 'rule'.

The binding of the first parameter to 'rule' is used to retrieve a fact which is a semantic rule. Note that this form of retrieval is ignored if CODE is uninstantiated.

(b) CODE has value 'explicit'.

Retrieval is based in simple unification with facts which were explicitly inserted. All parameters which were unbound become bound and one can invoke failure to fetch further matches. For example the goal, fetch_fact(explicit,_,[_,chic,is_a,_]), succeeds as,

fetch_fact(explicit,global,[f(20),chic,is_a,cs_lecturer]).

(c) CODE has value 'implicit'.

Suppose we have issued the goals,

insert_fact(explicit,global,[_,cs_lecturer,clever_with,programming]),

fetch_fact(implicit,_,[_,chic,clever_with,programming]).

The retrieval mechanism searches for a set which includes 'chic' as member then attempts to find an explicit fact in database 'global' relating this set to 'programming'. The fact we just inserted satisfies this search so fetch_fact succeeds with, 'chic,clever_with,programming'.

The search through set member relations recurses to any depth so a large number of facts can be implicit in the database.

Use of multiple databases and certain restrictions on the retrieval mechanism are the only significant differences between MULTIFACTS and the simpler FACT model.

Finally, if the value for CODE is left unbound then both explicit and implicit retrieval are utilised in the search.

Other operations provided by MULTIFACTS allow deletion or storage of facts. For example, 'delete_fact' will remove the fact supplied as parameter from the named database provided it is not designated as belonging to another database. Otherwise the designation is just removed.

The performance of MULTIFACTS can be tuned to the particular application. The schema (semantic rules) are normally checked before operations are permitted to access the databases, however, goal 'checking(off)' will stop these tests and access times will improve noticeably. Set member relations may also be omitted if checking is off. Performance also improves if all facts are inserted and fetched explicitly.

5.4 CONTEXT-FRAME : Maintaining World Views.

Module 'context-frame' contains operations to manage sets of facts, each of which provides an independent description of the world being modelled. Similar to the ART 'viewpoint', each context-frame describes some partial solution to a problem generated in the course of problem solving. The module uses the MULTIFACTS module and provides operations to record certain information about context-frames such as pointers between them or whether they represent an erroneous description of a problem.

Context-frames are represented as individual databases in MULTIFACTS. Operations of this type take parameters which should be lists of triples ie. fact identifiers are not used. Context frames are generated during problem solving and record the search space which has been explored. The manager of such a search space will be introduced after we examine some operations of CONTEXT-FRAME.

Most of the operations are simple extrapolations of MULTIFACT operations in that they recurse through a list of facts, invoking a

MULTIFACT operation on each cycle. For example, the code of 'put_cframe', shown in figure 5.3, makes use of 'insert_fact'.

```
put_cframe(Cframeid,[Fact|Restfacts]):-
        insert_fact(explicit,Cframeid,[Factid|Fact]),
        put_cframe(Cframeid,Restfacts).

put_cframe(_,[]).
```

Figure 5.3. Code of put_cframe.

Operations 'match_cframe' and 'retract_cframe' use similar code with operations 'fetch_fact' and 'delete_fact' respectively. Context frames have an identifier which is a PROLOG term of the form 'cf(<INTEGER>)', generated by operation 'next_cframe'.

Whereas 'match_cframe' attempts partial matches of lists of facts, operation 'get_cframe' retrieves all facts in a context frame. This operation (shown in figure 5.4) makes use of fact identifiers to prohibit retrieval of the same fact more than once.

```
get_cframe(Cframeid,[Fact|Restfacts],Found):-
        fetch_fact(explicit,Cframeid,[Factid|Fact]),
        not( member(Factid,Found) ), !,
        get_cframe(Cframeid,Restfacts,[Factid|Found]).

get_cframe(_,[],_).
```

Figure 5.4. Code of get_cframe.

The operation is invoked by a goal such as 'get_cframe(cf(1),Facts,[])'.

Operation 'copy_cframe' simply copies facts from one context frame to another. Since MULTIFACTS records one copy of a fact only, the effect of 'copy_cframe' is to designate each fact of the first context-frame as belonging to the second context-frame. Rather than use 'get_cframe' then 'put_cframe', the code of figure 5.5 is faster.

```
copy_cframe(Cframeid1,Cframeid2):-
        fetch_fact(explicit,Cframeid1,[_|Fact]),
        insert_fact(explicit,CFrameid2,[_|Fact]),
        fail.

copy_cframe(_,_).
```

Figure 5.5. Code of copy_cframe.

Important meta-information are father-daughter relationships between context-frames, managed with operations such as 'relaté_cframes' (figure 5.6).

```
relate_cframes(Fathercf,Daughtercf):-
        insert_fact(explicit,global,[_,Fathercf,in_dbase,Daughtercf]).
```

Figure 5.6 Code of relate_frames.

A context-frame can be 'poisoned', ie. labelled as invalid, using the poison_cframe operation (figure 5.7).

```
poison_cframe(Cframeid):-
        insert_fact(explicit,global,[_,Cframeid,is_poisoned,true]).
```

Figure 5.7. Code of poison_cframe.

5.5 CONJECTURES : Context-sensitive Database Manager.

The CONJECTURES module is a tool to generate plans for simple IKBS applications in design or robotics. Modelled on the database manager described for CONNIVER [McDermott&Sussman72], CONJECTURES includes a context mechanism for managing the space of alternative solutions to the design problem.

The module may be used within systems where plan development is represented as data-directed application of knowledge sources which alter non-decomposable state descriptions of the problem. Each description resides in a database known as a CONTEXT. CONJECTURES maintains a tree of such contexts like those of figure 2.2 (page 28). Where more than one knowledge source is applicable the system generates alternative sub-contexts on alternative branches. The data structure is similar to the net of viewpoints maintained by the ART tool. State descriptions which remain unchanged are inherited from father to daughter contexts and modified descriptions are represented as sets of assertions and retractions.

Two methods of implementing inheritance of information for the tree structure are possible in terms of the data types introduced earlier. Trade offs are economy of storage versus computational cost of retrieval. A context-frame may describe each node of the tree and include facts which describe both added and retracted information. Alternatively, one can simply copy facts not retracted to the new database of the sub-context. The first method is more sparing of storage but note that all information in a context is only found by searching back to the root of the tree taking note of retractions along the way. No explicit retractions are recorded by the second method and access to all context information is possible with one call of 'get_cframe'. However, more tuples designating ownership must be propagated.

The application domain dictated use of the second design method for two reasons. Firstly, programs are large structured objects therefore facts describing program structures are likely to require much more store than tuples designating ownership. The storage cost of propagating the latter therefore is not comparatively high. Secondly, low-level transformations are typically reversible so subcontexts are likely to be generated which are identical to an ancestral context. Such cycles are less costly to detect with the second method of implementing inheritance and can be made automatic.

The ART tool appears to have inheritance implemented along the lines of the first method. As a consequence, the knowledge base must include explicit knowledge sources to detect and eliminate cycles. This requirement would conflict with the intended ease-of-use of ENCORES.

A CONTEXT is therefore represented directly as a context-frame and operations recording relationships between frames can be used to build the tree structure.

Important operations of CONJECTURES facilitate search through the tree. The root context is labelled by an assertion of form 'is_root_context(<ROOT>)'. Operation 'super_context' (figure 5.8) locates ancestral or child contexts according to the bindings of its parameters. The super context is the immediate ancestor or father context of its daughter, or ancestor of this father.

```
super_context(Father,Daughter):-related_cframes(Father,Daughter).

super_context(Ancestor,Daughter):-
        related_cframes(Ancestor,Father),
        super_context(Father,Daughter).
```

Figure 5.8. Code of super_context.

The main constructor operation, 'sprout_context', takes three parameters as follows, sprout_context(<FATHER>,<NEWDAUGHTER>,<FACTS>).

The first parameter should be bound with the identifier of the context from which the new daughter context is being sprouted. The third parameter should contain a list of facts to be retracted conjoined with a list of facts to be added, both lists being separated by the triple, 'lastfacts,retracted,true'. Some of the facts to be added (contained in parameter FACTS) may be inherited from those of the father context. Operation 'put_cframe', which uses 'insert_fact', ensures no duplication takes place. The operation also searches for an ancestral context identical to the new daughter and 'poisons' the daughter if this is the case to prevent cycles forming. The code is shown in figure 5.9.

```
sprout_context(Cxt,Newcxt,Facts):-
        next_cframe(Newcxt),
        relate_cframes(Cxt,Newcxt),
        get_cframe(Cxt,Oldfacts,[]),
        /* process retractions if any */
(( append(Retractions,[[lastfacts,retracted,true]|Newfacts],Facts),
           !,
           delete_all(Retractions,Oldfacts,Inheritance)  );
           /* no retractions */
           (Newfacts = Facts,
           Inheritance = Oldfacts)),
        append(Inheritance,Newfacts,Cxtfacts),
        put_cframe(Newcxt,Cxtfacts),
        /* now check no cycles are forming */
        (( super_context(Supercxt,Newcxt),
           get_cframe(Supercxt,Cxtfacts,[]),
           poison_cframe(Newcxt));
           true),
        !.
```

Figure 5.9. Code of sprout_context.

Operations 'append' and 'delete_all' are simple list processing operations to join two lists and remove all elements of one list from another, respectively.

Notice that 'sprout_context' need not take account of earlier retractions when backtracking through the ancestral chain since inheritance is implemented as propagation of fact labels. The code is designed for simplicity and in the interests of getting a working system. More complex implementations may check for subsumption more efficiently, however extensive testing and comparison of these was beyond the scope of the current work.

The main retrieval operation of CONJECTURES is 'fetch_context' which takes a set of partially instantiated triples and searches for a match in some context. Operation 'match_cframe' is used to locate as many facts as possible in the context. Any remaining triples must be matched in the global database if the retrieval is to succeed. Of course, the context must be valid ie. not poisoned. Figure 5.10 shows the code of 'fetch_context'.

```
fetch_context(Cxt,Facts):-
        match_cframe(Cxt,Facts,Restfacts),
        not( poisoned_cframe(Cxt) ),
        match_cframe(global,Restfacts,[]).
```

Figure 5.10. The code of fetch_context.

CONJECTURES is used in ENCORES to maintain programs under development at any one level of a blackboard. If components of the program (eg. modules, routines, declaration parts or program bodies) are distributed amongst a number of attributes of some blackboard object then those attributes, not changed by a refinement, will be inherited. Attributes should therefore be chosen with economy of store in mind.

Further, more than one root context may be created by the action of

APPEAL rules at any one level of a blackboard. The reader may enquire how proliferating descriptions of systems at various blackboard levels are related. How does one move between blackboard levels to locate consistent abstract descriptions of refined programs?

5.6 ABSTRACTION : linking inter-level state descriptions.

The ABSTRACTION module provides operations for the management of pointers linking any two contexts on adjacent blackboard levels which represent consistent descriptions of the plan at any stage of its development. The pointers provide a basis for reasoning at various levels of abstraction : any context describing a problem state on a low level of the blackboard may have more abstract descriptions on higher levels and the contexts describing these can be located by searching along the pointers. We therefore name the pointers ABSTRACTION LINKS.

Like the father-daughter pointers used by CONJECTURES, abstraction links are directed from the context which triggered the knowledge source, to the new context which it generated. Links can be directed up or down blackboard levels : the former are represented as facts with relation 'abstracts_to', the latter by facts with relation 'specialises_to'.

Higher-level contexts usually represent major stages in the development of a plan. Each has a more detailed description amongst contexts on a lower blackboard level. Rather than have a complete set of abstraction links to each, a single link is maintained to the first lower context only, for economy of storage. This link is considered to be inherited by its descendents. Computation cost of retrieval is traded for economy of store, so not surprisingly the retrieval operation is more complex and encodes rules for locating links along any blackboard level.

Constructor operations simply insert the appropriate relations in the global database of MULTIFACTS. Operation 'put_abstraction' (figure 5.11) records vertically directed links.

```
put_abstraction(Abstractcxt,Specialcxt):-
  insert_fact(explicit,global,
              [_,Specialcxt,abstracts_to,Abstractcxt]).
```

Figure 5.11. The code of 'put_abstraction.

Operation 'put_specialism' works similarly but uses relation 'specialises_to'. Operation 'is_abstraction' determines if one context is an abstraction of another by searching for either an 'abstracts_to' or 'specialises_to' relation.

Operation 'valid_abstraction' (figure 5.12) encodes the rules for searching through the pointers to locate more abstract (or specialised) descriptions described by other contexts.

```
/* Valid abstractions can be found by following the links directly*/
        valid_abstraction(Abstate,Specstate):-
                is_abstraction(Abstate,Specstate).

/*Alternatively, one can search amongst descendents*/
/*a context at one end of an abstraction link */
        valid_abstraction(Highercxt,Lowercxt2):-
                is_abstraction(Highercxt,Lowercxt1),
                super_context(Lowercxt1,Lowercxt2).

/* In some circumstances, reasoning may have been carried*/
/* on at a higher level without regard to lower-level contexts. */
/*The refined descriptions at higher levels are still assumed */
/*to be abstractions of the last lower-level description.*/

        valid_abstraction(Highercxt2,Lowercxt):-
                super_context(Highercxt1,Highercxt2),
                is_abstraction(Highercxt1,Lowercxt).
```

Figure 5.12. The code of 'valid_abstraction'.

The validity of the rules rest on the assumption that refinements acting at a single level of representation are used to generate equivalent programs.

5.7 The Blackboard Manager of ENCORES.

The blackboard data type of ENCORES is a multi-level database in which trees of the sort managed by CONJECTURES are found at each level. An important point about reasoning within blackboards is that, in theory, inferences may be drawn both from and to many of these levels, at any one time, by the action of knowledge sources. Some simplifying assumptions have been made about these processes to contain the complexity of the implementation. Firstly, given information to be matched on various blackboard levels, we assume that retrieval can be performed down descending levels amongst contexts which are valid abstractions (as defined above). Secondly, we assume that available knowledge can be factored into knowledge sources, each of which will generate objects at one level only.

Both the retrieval and constructor operations of the blackboard type take parameters which are pairs of context-level identifiers. Operation 'match_blackboard' performs retrieval given information as to the last context and level where facts were found and the level where remaining facts are yet to be matched. The clauses needed to retrieve facts which are rules or in contexts at the same or adjacent blackboard levels are shown in figure 5.13.

```
/*Rules are blackboard objects and are matched*/
/*in a database named 'rules' on a level named 'rule'.*/
        match_blackboard(_,[rules,rule],Facts):-
                !, match_cframe(rules,Facts,[]).

/*If retrieval is to be performed at the same level where the */
/*previous set of facts were matched, then a search */
/*is made in the same context*/
        match_blackboard([Highercxt,Higherlevel],[Cxt,Level],Facts):-
                Higherlevel=Level, Highercxt=Cxt,
                fetch_context(Cxt,Facts).

/*Normally, the next group of facts are to be matched on a */
/*lower blackboard level*/
        match_blackboard([Highercxt,Higherlevel],[Cxt,Level],Facts):-
                valid_abstraction(Highercxt,Cxt),
                fetch_context(Cxt,Facts).
```

Figure 5.13. The code of 'match_blackboard'.

The reader should note that the treatment of rules as objects continues to a low level of implementation. The internal representation of rules will be described in the next chapter.

The main constructor operation, 'modify_blackboard', takes a list of context-level pairs, which triggered a knowledge source, and a list of new facts with the level on which they are to be recorded. A new context is generated with appropriate linkages from these triggering contexts. Figure 5.14 shows the code.

If one of the triggering contexts was at the same level as the context to be generated, then we must retract previous values of object attributes from the father context which are receiving new values. This is done simply by unbinding the values of each fact, then appending the lot, with triple 'lastfacts,retracted,true' to the list of facts. Operation 'sprout_context' makes these retractions from those inherited from the father context.

List processing operations, 'select' and 'append_all', select elements from lists and conjoins lists of lists, respectively. Operation 'put_abstractions' cycles through each of the triggering context-level pairs and establishes an abstraction link in the appropriate direction for each.

Another clause for 'modify_blackboard' applies where all triggering contexts are at a different level from that where new facts are to be recorded (ie. the 'select' goal above fails). In this case a new root context is established.

```
/*case, a triggering context is on same level as that to be generated*/
        modify_blackboard(Trigcxtlevels,[Newcxt,Atlevel],Facts):-
                select([Cxt,Atlevel],Trigcxtlevels,Restcxtlevels), !,
                unbind_facts(Facts,Unboundfacts),
                append_all([Unboundfacts,[[lastfacts,retracted,true]],
                        Facts],Newfactlist),
                sprout_context(Cxt,Newcxt,Newfactlist),
                put_abstractions(Restcxtlevels,[Newcxt,Atlevel]), !.

/* Case, generate a new root context */.
        modify_blackboard(Trigcxtlevels,[Newcxt,Atlevel],Facts):-
                is_root_context(Newcxt,Facts),
                put_abstractions(Trigcxtlevels,[Newxct,Atlevel]).
```

Figure 5.14. The code for 'modify_blackboard'.

A further blackboard operation, 'write_blackboard', outputs all contexts for a given blackboard as tables of facts separated by those facts representing pointers between them. The operation provides an interface to the blackboards and is a primitive version of the 'viewpoints mechanism' of ART.

5.8 ACSYS : Acquisition Subsystem.

ACSYS is a friendly interface to receive blackboard schemas from users. Information can be entered about blackboards, levels, objects and attributes. ACSYS records the information as semantic rules and set-membership relations in MULTIFACTS. Invoked with PROLOG goal, 'acsys', the tool responds as follows:

```
  Acsys (Acquisition Subsystem)
  Level 1 of control blackboard
  Class of object :
```

The user enters the name of the object class (which is also the name of the top level of the control blackboard). Possibilities are 'problem', 'strategy' or 'tactic'. ACSYS reminds the user that the first attribute of any object is assumed to be its distinguished 'name' attribute. Subsequent attributes, and permissible values for each, are then requested from the user. Responding with end-of-file (eof) character, completes the entry of attributes and their valid values. ACSYS then proceeds to query the user about the next lower level of the blackboard. Similarly, eof causes exit from this and ACSYS repeats the sequence for definition of the domain blackboard. Essentially, schema information is requested of class-subclass hierarchies in a cyclic manner.

ACSYS records schema information as set-member relations and permits use of checking procedures supplied by the user. For example, the dialogue to define the domain blackboard used in figure 2.5 (page 40) would proceed as follows (user replies are underlined):

```
        Level 1 of domain blackboard
        class of object: datatype.
        (attribute 1 of datatype is its name)
        attribute 2 of datatype : type.
        valid value of type : predicate(is_ast).
```

These responses result in the following facts being added to the database,

```
        fact( f(11), datatype, level_of, domain).
        fact( f(13), datatype, level_num, 1).
        fact( f(15), datatype, type, is_ast).
```

The last of these is a semantic rule (others are global facts). If checking of new facts is permitted during the consultation, MULTIFACTS will invoke procedure 'is_ast' with a single parameter, the value to be checked. Users must ensure such procedures are available in the PROLOG workspace.

At the next lower level, that of OPERATION, ACSYS will place all valid values of the 'action' attribute in a set for checking purposes. For example, the dialogue,

```
        Level 2 of domain blackboard
        class of object : operation.
        (attribute 1 of operation is the name)
        attribute 2 of operation : action.
        valid value of action : reads.
        valid value of action : deletes.
```

will result in semantic rule,

```
        fact( f(21), operation, action, action-val)
```

and set-member relations,

```
        fact( f(23), reads, is_a, action_val)
        fact( f(25), deletes, is_a, action_val).
```

5.9 Blackboard Configuration Tools.

In general, a blackboard system using the management software must be supplied to parse problem statements to objects; distribute these objects amongst an initial configuration of the blackboard; reassemble objects of the solution and finally output this in a suitable form. For our purposes, these tools must translate programs to and from the tree representation (described in section 3.3, page 57). Therefore constructing configuration tools for an application of ENCORES involves working with abstract descriptions of a programming language.

APPENDIX-D describes the construction of a language parser, in PROLOG, to read user supplied programs and return an abstract syntax tree. Parsers so built also interface to the APPEAL language compiler (see next chapter) for parsing APPEAL metalanguage expressions. Parse trees are decomposed and inserted in the blackboard by a user supplied routine which asserts them as FACTS (eg. using procedure 'modify_blackboard'). The tree representing the solution is reassembled on completion of a consultation by user supplied software. Source code is regenerated using a tool known as a 'pretty-printer' supplied for the application. APPENDIX-E describes a tool available within ENCORES which can be used to compile pretty-printers.

In contrast, once knowledge acquisition is underway, the engineer works with tools supporting friendly object-centred descriptions such as the APPEAL editor. This engineer need rarely be burdened with abstract descriptions of systems and preferably be a separate individual from the person who undertook the configuration of the application

5.10 Summary.

The blackboard manager of ENCORES is composed from many simpler data types. Contexts are managed by the CONJECTURES tool at each blackboard level. Operations of the ABSTRACTION data type provide inter-level linkages between contexts. Nodes of this network are sets of facts managed by the CONTEXT-FRAME data type using operations of the MULTIFACT tool. All these abstract data types are separate files of PROLOG predicates, or 'modules', which are loaded together using the MODULE tool.

These tools are used within ENCORES as follows. The ACSYS subsystem records blackboard schemas as facts using operations of MULTIFACTS. In particular, facts which are semantic rules record valid values for object attributes. If the checking goal of MULTIFACTS is not asserted to be 'off' (page 100) then checking of many blackboard entries will be performed as they are added to the blackboard (ie. rules are effectively debugged at runtime). However, performance of the system falls drastically with these checks so semantic rules are seldom used. The blackboard operations are used during the evaluation of APPEAL rules. Rule premises are translated to retrievals using 'match_blackboard'; rule conclusions are translated to invocations of 'modify_blackboard'. The next chapter shows how these translations are made.

CHAPTER 6.

APPEAL LANGUAGE IMPLEMENTATION.

APPEAL rules are intended to be a transparent representation for programming knowledge and this transparency is based on use of patterns of the familiar string representation of programs. As was pointed out in chapter 3, however, attributed tree representations of programs facilitate more efficient pattern matching. If efficiency is to remain a goal, then APPEAL rules must also find an internal representation based on attributed trees.

The general utility of APPEAL rules derives from the provision of metavariables and this generality must be retained with their internal representation. One approach is to compile each rule to a set of tree-matching and -generating templates in which variables now have the role of matching with trees. However, tree representations of programs include type information which is absent or inexpressible in program patterns. Thus we augment APPEAL with a mechanism, named TYPE QUALIFICATION, by which the knowledge engineer can impose constraints on the matching types of metavariables.

The internal representation of rules and their evaluation must also reflect their representation as blackboard objects. In particular, control rules may reference attributes of domain rules, using patterns based on the APPEAL language. Implementation of this technique, known as 'content referencing' (page 71), involves complex trade-offs of economy of store versus efficiency of rule retrieval and evaluation. If rules are decomposed to collections of object-attribute-value triples, like other blackboard objects then referencing is efficient but retrieval and assembly of rules for evaluation will be quite costly. Alternatively, if rules find a representation as integral units then referencing will be complex and costly.

Finally, the products of rule evaluation and execution are not only new blackboard objects but also instantiations of rules themselves (better known as 'rule activations'). These may also be recorded for the purpose of preventing rules executing in an identical manner more than once. More importantly, these records provide a trace of the software development within the system and provide the raw material for investigations of explanation in knowledge-based programming. The possibilities are discussed further in chapter 8.

The following sections of this chapter describe how these issues were addressed in the implementation of the APPEAL language. Rules, like other blackboard objects are reducible to facts, of the sort managed by MULTIFACTS. The compilation process is then described. The object description of rules provides a natural hierarchy of abstract data types along which the compilation process is factored. The pattern matching facilities of PROLOG are supplemented with two operations to effect pattern matching with lists and trees. Evaluation code for both control and domain rules take a simple form. The representation and management of activation records of rules will be sketched as will a tool for editing APPEAL rules.

6.1 Internal Representation of APPEAL Rules.

Each APPEAL rule is compiled to two facts : one containing the rule premise, the other, the rule conclusion. The facts are PROLOG assertions of the form managed by MULTIFACTS. Figure 6.1(a) is a reproduction of the FOLD transformation. Figure 6.1(b) shows the two facts compiled from this rule. The term structure of these assertions are expanded and numbered (as is the APPEAL statements) to help explain how components of the rule are compiled.

```
 1.      if:  an equation(1) has
 2.       &     code = '---Lhs1<=Rhs1;'
 3.       &   an equation(2) has
 4.       &     code = '---Lhs2<=Rhs2;'
 5.       &     Rhs2 = '__Rhs1__'
 6.     then     equation(2) has
 7.       &     Rhs2 = '__Lhs1__'
 8.       &     code = '---Lhs2<=Rhs2;'
```

(a)

```
 1.      fact( f(10), rule(1), conds, [
 2.              vars([ [E1,equation(1)],[L1,lhs1],[R1,rhs1],
 3.                     [E2,equation(2)],[L2,lhs2],[R2,rhs2],
 4.                     [Tree,rhs(2)]     ]),
 5.              [a, [E1,is_a,equation],
 6.                 [E1,code, eqn(L1,R1)],
 7.                a,[E2,is_a,equation],
 8.                 [E2,code, eqn(L2,R2) ],
 9.                 transform(R1,R2,rhs2(2),Tree) ]  ] ).

10.      fact( f(12), rule(1), actions, [
11.              vars([ [E2,equation(2)],[L2,lhs2],[R2,rhs2],
12.                     [Tree,rhs2(2)],[L1,lhs1] ]),
13.              [ transform( rhs2(2),Tree,L1,R2),
14.                a,[E2,is_a,equation],
15.                 [E2,code, eqn(L2,R2)] ] ] ).
```

(b)

Figure 6.1. External(a) and Internal(b) forms of an APPEAL Rule.

A major problem arises with the treatment of variables within PROLOG. Most implementations of the language convert user supplied variables to an internal form which is used consistently within a single PROLOG clause. If the clause is decomposed and written to another medium then this consistency is lost. Thus when APPEAL rules are compiled to PROLOG assertions, some information must be included with these so that appropriate variables in each assertion can be made identical when the rule reassembled. Preferably, this extra information should not only provide a means of linking variables but also include the user-supplied symbols.

Figure 6.1(b) shows the two facts compiled from the APPEAL rule. This representation can be understood as follows:

(1) Each fact is a triple with subject of form 'rule(<N>)' where N is the rule number.

(2) One fact has relation 'conds', the other 'actions' to indicate that they contain the condition part and action part of the rule, respectively.

(3) Both facts have objects which are lists of PROLOG terms, the first of which has functor 'vars'. This term contains information for linking identical variables in both facts and for reproducing the original symbols for the metavariables. Remaining terms are the compiled instructions.

(4) All metavariables used in the appropriate rule part have an entry in the 'vars' term consisting of a pair : the first element is the PROLOG variable substituted by the system; the other the original symbol with initial letter lower-case. Thus we see, for example, that metavariable 'Lhs1' in lines 2 and 7 of figure 6.1(a) gives

rise to entries '<PROLOG VAR>,lhsl>' in each 'vars' term (lines 2 and 12 of figure 6.1(b)). The two entries are unified on reassembly of the rule. The PROLOG variable substituted for the metavariable in each fact is normally different and is certainly read differently when facts are input from secondary store.

(5) Two different objects from the class of equation are matched by this rule. Each has a specific name but no reference is made to names in the rule. Reference is indirect by use of reference numbers (eg. lines 1 and 3, figure 6.1(a)). Variables are introduced for the names of these objects and each gives rise to a pair in the 'vars' term of the form '<PROLOG VAR>,equation(<N>)', where N is the reference number.

(6) Other terms in the object of each fact are either three-element lists or goals. The former are templates for matching facts in some blackboard context; the latter procedures establish further conditions or transform these facts.

(7) Indirect references to objects by numbers (eg. line 1,figure 6.1(a)) give rise to templates for matching elements from the class of objects. Statements of form 'an <OBJECT>(<REFNO>) has' and 'all <OBJECT>(s) have' compile to template '[X,is_a,<OBJECT>]' preceded by symbol 'a' or 'all' as appropriate. (eg. line 5,figure 6.1(b)).

(8) References to object attributes, such as 'code' (line 2, figure 6.1(a)), are compiled to templates for retrieving the appropriate fact whose object unifies with the tree template compiled from the program pattern (line 6, figure 6.1(b)).

(9) APPEAL statements for locating and transforming a portion of code (eg. lines 5 and 7 of figure 6.1(a)) give rise to a 'transform' procedure call in each fact (lines 9 and 13, figure 6.1(b)). A call to a variant of this procedure, named 'transform_all', is compiled where the APPEAL statement has the form 'all(<ATTRIBUTE>='__<PATTERN>__')'. The functions of both 'transform' and 'transform_all' operations will be described shortly. Both are used to manipulate trees on both sides of the rule and information is passed using an entry in the 'vars' term (eg. lines 4 and 12, figure 6.1(b)).

The rule shown is a domain rule but control rules have an almost identical form internally except that patterns, to be matched with domain rules, are compiled to list operations. (Notice that the internal representation of rules is based on lists, not trees).

Other facts are compiled for the rule, besides those shown in figure 6.1(b). Two designate the rule as belonging to database 'rules',

```
     fact( f(11), f(10),in_dbase,rules)
     fact( f(13), f(12),in_dbase,rules).
```

Others record the object class (ie. blackboard level) on which the rule acts,

```
     fact( f(14), rule(1), acts_on, equation)
```

and the blackboard on which the rule makes its contribution,

```
     fact( f(16), rule(1), ks_of, domain).
```

6.2 Compilation of APPEAL Rules.

The object description of rules provides a natural decomposition of premise and conclusion parts for the purposes of compilation. Both parts of the rule can be considered as sets of object templates (eg. lines 1-2, 3-5, and 7-8, figure 6.1(a) describe three templates, two to be matched with equations, one to generate an equation). Each template is seen to consist of a quantifier (eg. 'a','an' or 'all) with the object class, followed by APPEAL clauses which access attribute or metavariable values. Thus operations for managing rules can be grouped round six data types which we call rule, object_templates, object_template, quantifier, clauses and clause. Further, each type may be programmed as a module (see Chapter 5,page 93).

The complete code of these modules is too large to reproduce here but the compilation process is characterised. Each data type has operations to parse APPEAL tokens and assemble the compiled instructions. Figure 6.2 shows the type hierarchy and names of relevant operations associated with each type. Vertical arrows depict the flow of control during compilation : tokens are passed down parsing operations of each type; instructions are assembled as control returns to higher-level types via the constructor operations.

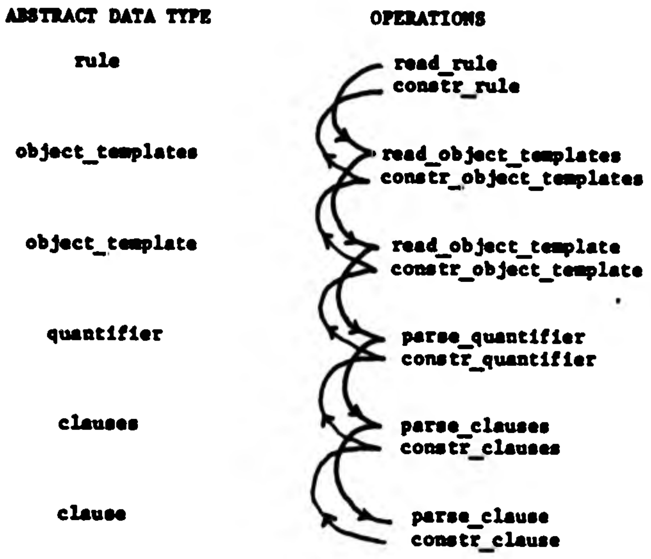


Figure 6.2. Rule Component Types and their Operations.

The most important consumers of tokens are operations 'parse_quantifier' and 'parse_clause'. The former creates two-element lists of the form,

    <QUANTIFIER>,[X,is_a,<OBJECT-CLASS>]

for matching instances of objects belonging to the appropriate class (eg. line 5, figure 6.1(b)). Operation 'parse_clause' parses APPEAL statements within the object template by calling the parser of the application language with tokens of program patterns. This parser must parse the pattern to a tree template and return information on the types metavariables will match with using this template. Operation 'parse_clause' then returns three-element lists of form,

[X,<ATTRIBUTE>,<TREE_TEMPLATE>]

(eg. line 6, figure 6.1(b)) with the matching types of metavariables.

A difficulty with parsing program patterns is that generally more than one parse is possible. For example, in the pattern of line 2, figure 6.1(a), metavariable 'Rhs1' will be substituted by a tree-matching variable which will match with 'cl_rhs_expr' (ie. any expression found on the right-hand-side of an equation). As used in line 5, figure 6.1(a), the same metavariable would parse to match any HOPE construct defined by the abstract grammer. Type matching constraints on metavariables must be consistent when the compiled instructions are assembled from different statements of the rule. For this reason, the type information returned by operation 'parse_clause' is accumulated and checked at all stages during the assembly of the instructions. Inconsistencies force backtracking to find alternative parses of patterns until the matching types of metavariables are identical.

6.2.1 Implementation of Type Qualification.

The constructor operations append the resulting lists of instructions together with identities of the original metavariables and those variables substituted by PROLOG. Information of matching types of all metavariables is then presented to the engineer as a table at the final stage of compilation. The matching type associated with each variable is the most general type returned by the parser for all occurrences of the variable in the rule. The engineer can now impose more specific types for matching with selected metavariables. This procedure, named TYPE QUALIFICATION, results in further constraints for matching the rule represented as further PROLOG goals to be assembled with the instructions.

For example, unqualified variable 'Rhs1' will match any right-hand-side expression when the rule is parsed, but if the engineer qualifies the matching type of this variable to be 'cond_clause', then only 'if__then__' expressions will be matched since goal 'R1=cond_clause(_,_)' is appended with the instructions.

The user can also force a reparse so that certain metavariables will match single instances of types rather then sequences of types. For example, pattern 'Op(Params)' within a rule might parse such that variable 'Params' will match a sequence of type 'cl_lhs_expr' (ie. any expression found on the left-hand-side of equations). At type qualification the engineer can force a number of alternative parses of the rule by issuing goal 'reparse'. One alternative will be a parse of 'Params' to match with a single occurrence of type 'cl_lhs_expr'.

When type qualification is complete, the final instructions are compiled to the two facts and asserted in the PROLOG workspace by operation 'put_rule'. No other record of the original APPEAL statements remain : ENCORES works entirely with the compiled form of rules.

6.2.2 Implementation of Content Referencing.

APPEAL rules are considered as objects with a set of attributes for the purposes of content referencing. Each rule has two attributes for every object class on the blackboard where it contributes its knowledge. Each attribute contains a list of clauses from object templates of the rule and each has a name of form, '<OBJECT-CLASS>-<PART>', where PART takes value 'conds' or 'actions' indicating that clauses are from the condition or action part of the rule. For example, the rule of figure 6.1(a) has two attributes named 'equation-conds' and 'equation-actions'. Attributes associated with other object classes (if any) are empty lists.

Control rules, which are task executors, reference domain rules with statements of the form,

```
     (if) a rule(1) has
          <OBJECT>-<PART> = '__<PATTERN>__'
     then tryrule(1)
```

where the PATTERN is intended to match with APPEAL statements. For example, pattern 'code = Any' can be used to match with either the 'equation-conds' or 'equation-actions' attributes of the rule in figure 6.1(a).

Since the internal representation of rules is based on lists, object templates for matching other rules are compiled to rule retrieval and list matching operations. For example, the APPEAL statement,

```
     equation-conds = '__code = Any__'
```

will be compiled to instructions,

```
     a,[R,is_a,rule],[R,conds,C],[R,actions,A],
     append(_,[[X,is_a,equation]|Otherinstructions],C),
     match_list(Otherinstructions,[--,[X,code,Any],--])
```

where the first three triples are used to retrieve facts comprising the rule (such as those of figure 6.1(b)). The append operation locates instructions associated with the relevant object class (eg.'equation') within the relevant fact (eg. that with relation 'conds'). Operation 'match_list' locates the instruction which accesses the relevant attribute (eg. 'code').

Other blackboard objects may be matched by the task executor before rules are sought which.match the pattern. Thus the pattern may be partly instantiated when matching with the rule takes place. Since matching is based on unification, the rule itself will become partly instantiated. Thus content referencing as implemented here not only matches rules but specialises them to the particular situation where they are to be tried.

6.2.3. ENCORES' Pattern-match Operations.

Operation 'match_list' extends the pattern-matching properties of unification for matching elements within lists. It can be used to unify a single term with another where both contain variables. For example, goal 'match_list([typevar(A),typevar(B)],[T1,T2])' succeeds binding T1 to 'typevar(A)' and T2 to 'typevar(B)'. The second parameter may include the symbol, '--', with variables to bind them to partitions of lists. For example, the goal,

```
match_list([id(pop),id(push),id(top)],[id(I),--,X,--])
```

will bind I to 'pop' and X to '[id(push),id(top)]'. A solitary '--' symbol will match with any partition. For example, goal

```
match_list([id(pop),id(push),id(top)],[id(I),--])
```

will just bind I as before. Operation 'transform' is a versatile predicate for both constructing and matching PROLOG terms embedded within PROLOG terms. Since syntax trees are represented as PROLOG term structures, 'transform' can be used to transform programs.

The operation takes four parameters of form,

```
transform(<NEWSUBTREE>,<NEWTREE>,<TREE-SPECN>,<OLDTREE>)'.
```

Given a tree in the fourth parameter, a search is made to find a descendant tree which matches the tree specification in the third parameter. The descendant tree is then substituted for the tree given in the first parameter. The transformed tree is returned in the second parameter. For example,

```
     transform( type_constructor(queue), Newtree,
         type_constructor(stack), in_types([type_constructor(stack)]))
```

will bind Newtree to 'in_types([type_constructor(queue)])'. A similar operation, 'transform_all', would make the substitution in all descendant trees which matched the specification.

An interesting feature of 'transform' is that it functions identically if values, supplied to the first pair of parameters, are inter-changed with the last pair. This symmetry is exploited in APPEAL rules for effecting transformations. Tree templates, compiled from program patterns, are matched and substituted with a label in the rule premise. The new code is substituted for the label by the transform call compiled from the rule conclusion. The variable bound to the labelled tree in the rule premise has to be unified with the variable representing the same tree in the conclusion when the rule is reassembled. Entries in the 'vars' term ensures this takes place (eg. lines 4 and 12, figure 6.1(b)).

6.3 Evaluation of APPEAL Rules.

We have seen that rules find an internal representation which is, in most respects, identical to that of other blackboard objects. As a consequence, a single pair of evaluating routines serve to process the premise and conclusion parts of both control and domain rules. One evaluator, contained in module 'lhs_eval', attempts to satisfy the rule premise. The other, in module 'rhs_eval', executes the conclusion part of the instantiated rule. Both evaluators access objects using operations of the blackboard manager.

Before evaluation, the two facts containing the compiled instructions of the rule are retrieved and the appropriate entries in their 'var' terms are unified so that variables are again consistent. The constructor operations can now be used in reverse (by virtue of the reversibility of the 'append' operation) to decompose these lists of triples and goals to those associated with each object template. Evaluation therefore reduces to matching and generating facts within the blackboard and invoking built-in procedures such as 'match_list' and 'transform'.

For the most part, evaluation consists of matching and generating facts within a single blackboard. However, the particular control mechanism developed for ENCORES (described in chapter 4) requires that each domain rule be executed as a result of evaluating a control rule which is a task executor. The premise of executors must match with objects of both control and domain blackboards (eg. tasks, rules and program objects). Also the conclusion part of an executor is the single statement 'tryrule', where the rule to be tried is that domain rule matched by the premise of the executor. Each evaluator then, must include code to handle two cases: one for evaluating task executors, the other for evaluating task generators and domain rules. Again, the code of each is large so we describe their behaviour.

6.3.1. Evaluation of Rule Premises.

Predicate 'test_object_templates', within module 'lhs_eval', determines if a set of object templates from a rule premise can be matched within some context of a blackboard. Given context and blackboard identifiers, with the list of compiled instructions from the premise, it processes these instructions recursively and returns remaining instructions which cannot be matched. Each cycle, in essence, consists of five steps:

(1) instructions compiled from the first object template are extracted

(2) these are decomposed to the quantifier, fact templates and goals

(3) these fact templates are matched with the first relevant object within the blackboard context

(4) the goals are now invoked (in which variables will now be partly instantiated)

(5) if quantification was universal then repeat steps (3) and (4) for all other objects in the context.
Backtracking is permitted at all times to permit the predicate to succeed. Also, due to the flexibility of binding variables within PROLOG, if no blackboard context is initially supplied then one will be found for the predicate to succeed.

Evaluation of rule premises for task generators and domain rules is straightforward using this predicate. Both match with objects within a single blackboard so a single invocation of 'test_object_templates' with the instructions constituting their premise completes the evaluation. Domain rules have a context chosen for them (by a task executor), whereas a context has to be found for evaluating task generators. The most recently generated tasks will be used to establish this context since the underlying organisation of the blackboard is a stack.

The premise of task executors has the form,

if <TASK OBJECT> & <DOMAIN OBJECT(S)> & <RULE OBJECT> ..

therefore evaluation consists of multiple invocations of 'test_object_templates' to the control, domain then control blackboard respectively, until all the compiled instructions are processed.

6.3.2. Evaluation of Rule Conclusions.

Predicate 'try_rule', within module 'rhs_eval', processes the conclusion part of APPEAL rules. The normal case is that objects are generated on the blackboard to represent the conclusion, however task executors conclude that a domain rule should be tried in a chosen context. The two cases are processed by two distinct clauses of 'try_rule'. A further clause is included to check records of rule activations previously executed to ensure that the rule instantiation currently being processed has not been encountered before.

The compiled instructions from the conclusion part of task generators and domain rules consist of fact templates and goals, some of which will have been instantiated during premise evaluation. Predicate 'try_rule' takes this list and the context(s) in which the premise of the rule was satisfied. Goals are separated from facts and invoked completing the instantiation of facts (eg. with transformed trees). Operation 'modify_blackboard' (chapter 5, page 111) is now called with these facts and contexts.

The conclusion part of task executors is processed in a simple manner. Since the compiled instructions describe the domain rule to be tried, 'try_rule' simply invokes both the premise evaluator and itself with the domain rule and the context where it is to be tried.

If 'try_rule' fails then backtracking into the premise evaluator is permitted to find an alternative instantiation of the rule. On each occasion 'try_rule' succeeds, the user is informed which rule fired and the record of its activation is stored in a form which will now be described.

6.4. Records of Rule Activations.

An activation record of a rule consists of three items of information:

(1) the number of the rule which was fired

(2) the blackboard contexts in which the rule was activated

(3) the list of instantiated fact templates and goals of the rule conclusion.

This information is deemed sufficient to reconstitute an instantiated form of the original APPEAL rule.

Activations are recorded as facts within MULTIFACTS. The three items of information are simply lumped into a single PROLOG term and deposited in a database named 'actvns'. Operations 'put_actvn' and 'get_actvn' are available to manage this database. All activations contain ground PROLOG terms, so the problem of unifying two terms, neither of which strictly subsumes the other, is avoided. The representation is inelegant but activations currently have a minor role within ENCORES. Their use is confined to avoiding repetitions of the same inference.

6.5 The APPEAL Editor.

The most important tool of ENCORES is the APPEAL editor. Rules are entered via a friendly interface which prompts for each clause of rules in turn. The editor parses and compiles these clauses, presents the table of metavariables to the user for type qualification then finally deposits the resulting facts in MULTIFACTS.

Invoked with goal 'read_blackboard_rules', the editor requests entry of control rules first and reminds the user of the rule number,

```
    Control rule 1
   if:
```

The user enters the first statement (eg. 'a tactic(1) has'). The editor prompts for the next statement with the ampersand, '&:'., to remind the user that all APPEAL clauses are and'ed together. The user enters end-of-file (eof) to complete entry of the rule premise. The editor now prompts with 'then:'. Statements of the rule conclusion are now entered in a similar fashion to those of the premise. If parsing of these clauses fails at any point then the user is invited to enter the rule again correctly.

The eof character completes entry of the rule and type qualification begins. A table like that of Figure 3.13 (page 63) is presented and the user responds along the lines described in page 62. The editor now prompts for the next control rule but the user can start entry of domain rules by responding with eof.

The current APPEAL editor permits entry of rules only. Deletion of rules or perusal of the knowledge base is accomplished with use of MULTIFACTS operations. Chapter 8 (page 158) describes editor functions which might be incorporated in future versions of ENCORES.

6.6 Summary.

A number of complex issues are addressed in the implementation of the APPEAL language. In particular, an internal representation for APPEAL rules has been chosen which has proved a good compromise in terms of economy of store and efficiency of retrieval and evaluation. The representation is almost identical to that of other blackboard objects giving uniformity to the rule evaluation software. The representation overcomes shortcomings with PROLOG's treatment of variables.

The symmetry of rule premise and conclusion parts has been exploited to factor compilation amongst a hierarchy of component data types. The same data types facilitate decomposition of the rule during evaluation. PROLOG's inherent backtracking has been exploited in both the compilation and evaluation software. APPEAL rules can be both parsed and evaluated in all possible ways.

CHAPTER 7.

IMPLEMENTATION OF THE TASK-ORIENTATED CONTROL MECHANISM.

The two basic objects of control in ENCORES are control rules and tasks. Chapter 4 described how individual control rules are compiled and evaluated to generate hierarchical task structures. In this short chapter we describe the control cycle of ENCORES and show, with an example, how it can take advantage of opportunistic strategies for problem solving.

The control cycle is maintained by an algorithm which directs activity on the control blackboard. Not only must it control the generation of task structure(s), but also execute them in a depth-first, left-to-right manner similar to that adopted for PADDLE goal structures (section 4.3, page 78). Clearly some scope is offered in the design of such an algorithm. One can combine growth of the task structure with its execution, or attempt to grow the task structure completely before executing it. The algorithm described here takes the former approach and is therefore one implementation of the control scheme described in chapter 4.

To begin this chapter, the control cycle is described informally with reference to the solution of a simple block-stacking problem. The problem finds frequent use in descriptions of knowledge-based planning (eg. [Nilsson80][Warren74][Sterling84][Kowalski79]). This serves better than any programming example since the domain knowledge base of three rules does not distract from control issues. It also provides greater insight into the operational details of the control mechanism; particularly the role of the stack organisation for backing up and the refocusing behaviour of the control cycle. We show how a simple depth-first strategy takes advantage of these back-up processes, then show how an opportunistic strategy takes advantage of refocusing to narrow the search considerably.

A formal description of the control cycle follows. The control algorithm is expressed cogently in PROLOG. Since ENCORES was first constructed a mechanism similar to the 'changeset' of AGE (page 30) has been added which will increase the efficiency of search for uninformed search strategies such as depth-first. We describe its implementation.

7.1 The Control Cycle of ENCORES : An Informal Description.

The control cycle of ENCORES is based on a pure production system (like the control mechanism of NEOMYCIN, page 73), but is modified to:

(1) Work with multiple strategies. Control rules may generate alternative task structures for a given problem describing alternative methods of solving it. The algorithm attempts to execute one task structure exhaustively before trying another.

(2) Work with multiple levels of representation. Since the task structure(s) are factored into levels of abstraction within the control blackboard, depth-first execution is assured by refocusing on successively lower levels.

The control mechanism takes advantage of ENCORES' stack organisation to drive the inference process. Control rules will be added to the stack with preferred methods at the top. At the heart of the control algorithm is a pure production system, that is, a loop consisting of two components:

(1) if find solution then TERMINATE,

(2) find first applicable control rule on the stack and apply it,

where no rule is permitted to execute in an identical manner more than once.

The problem with this simple algorithm is that it will generate tasks in a breadth-wise fashion where more than one tree of tasks can be generated. We want the control mechanism to work with one plán before trying others. It is also pointless to test for a solution until the domain blackboard has changed. (NEOMYCIN's control mechanism returns rule selection to the head of the list of rules only when the search space has changed and uses a rule to signal termination).

The control algorithm of ENCORES overcomes these problems with a loop which only tests the blackboard for a solution after a task executor has fired, and only tests rules for applicability which generate tasks on successively lower levels of the control blackboard.

The control algorithm is therefore a loop containing steps:

(1) if find solution then TERMINATE

(2) make the FOCUS-LEVEL the topmost level of the control blackboard,

(3) if find an applicable control rule which acts on the FOCUS-LEVEL
    then apply it,

(4) If a task executor has fired then repeat from one,

(5) focus on next lower level,

(6) repeat from (3).

The control model has four distinctive features arising from use of this algorithm:

(1) It is highly directed in its use of knowledge sources. Selection is established before rules are activated, not after, as with other control mechanisms. Rule activations are not recorded in an agenda (priority queue) : if the selected rule is applicable then it is immediately executed.

(2) Provision for back-up is essential. Alternative objects may be generated which meet the task specification. Other domain objects and/or domain rules might match the control rule currently driving inference.

(3) A key property of the control algorithm is that it fetches control rules from top of stack on each cycle. If high priority control rules (ie. those generating preferred plans) occupy top-most positions of the stack then the algorithm will attempt to fire these first on each cycle. Some will be task executors, possibly highly opportunistic in their specification of domain objects to be matched. If not executable during early cycles of the control mechanism, the execution of other tasks (of possibly other plans) may create the domain objects which will now trigger them. Thus the refocusing behaviour of the algorithm ensures that work will continue with high priority plans whenever possible and that the search is exhaustive.

(4) As a consequence of (1), back-up in the control mechanism is quite costly. All rules have to be activated with previously considered objects until one activation is found which generates new data. In

contrast, the control mechanism of OPM need not evaluate rules again during back-up. Since all activations of OPM rules have been recorded in an agenda (as described in page 77), the system need only select the next activation and execute this.
Essentially, the computational cost of repeated rule evaluation during back-up is traded for the direct application of rules in ENCORES. Such a trade-off is justified for the programming domain since search trees tend to wide (as argued in chapter 4 page 68) and rule evaluation costly.

A final remark about the control algorithm is that backtracking amongst the steps (specifically, steps 3 to 6) should be permitted as a corollary to feature (2). If no backtracking were allowed, control might pass to other plans before all alternatives for the first are pursued. These points will be made clear with an example before the complete algorithm is presented.

7.2. An Example : Opportunistic Stacking of Blocks.

For an illustration of the control mechanism applied to a simple block-stacking problem, consider an initial configuration of blocks described by figure 7.1(a). Let the configuration described in figure 7.1(b) be the goal state.

This classic problem is often used in discussions of goal-directed planning systems with reference to the problem of 'interacting subgoals'. Consider: if the goal is decomposed to subgoals 'put B on C' and 'put A on B', the first is not possible, while the latter leads further from a solution. Sophisticated devices have been introduced within goal-directed systems to overcome this limitation (eg. [Sacerdoti74][Tate85]).

The problem might be represented within a single level domain blackboard with level named BLOCK. Each BLOCK has two attributes : 'on' which names the object on which the block sits and 'clear', a boolean, used to indicate if no object is stacked on the block.

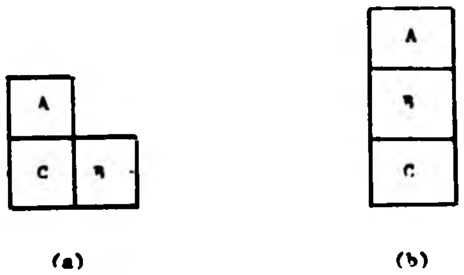


Figure 7.1 Initial State (a) and Goal State (b) of Blocks.

The only operations permitted are (a) move a block off another block onto table , (b) move a block off the table onto another block and, (c) move a block from one stack of blocks to another. These operations (properly named 'operators') can be represented in the APPEAL language as the three rules of figure 7.2. Note that the values of both 'clear' and 'on' have be adjusted when we move blocks. The right-hand-sides of rules record these adjustments.

```
  if  block(1) has
   &  clear = true
   &  on = X
   &  block(2) has
   &  name = X
then  block(1) has
   &  on = table
   &  block(2) has
   &  clear = true
```

(a)

```
  if  block(1) has
   &  clear = true
   &  on = table
   &  block(2) has
   &  name = Y
   &  clear = true
then  block(1) has
   &  on = Y
   &  block(2) has
   &  clear = false
```

(b)

```
  if  block(1) has
   &  clear = true
   &  on = Y
   &  block(2) has
   &  name = Y
   &  block(3) has
   &  name = Z
   &  clear = true
then  block(1) has
   &  on = Z
   &  block(2) has
   &  clear = true
   &  block(3) has
   &  clear = false
```

(c)

Figure 7.2 Three Operators of the Blocks Knowledge Base.

The knowledge base is complete when we specify the control regime or strategy to be employed when searching for a solution. Figure 7.3 shows three rules which describe depth-first search using classes and attributes defined within the blackboard schema. They find application with a four level control blackboard having levels PROBLEM, FOCUS, POLICY and RULE as described above.

Note that depth-first search can be specified for this example since:

(1) No new objects are generated. There are always three blocks A, B and C.

(2) Duplicate configuration of blocks are ignored by the system (since they are 'poisoned'.

```
                                                            if   policy(1) has
                                                             &   on = any
   if   problem(1) has          if   focus(1) has            &   block(1) has
    &    name = stackabc         &    name = block           &    on = X
 then    focus(1) has         then   policy(1) has           &   rule(1) has
    &    name = block            &    on = any               &   block-conds='_ _on=X_ _'
                                                          then   tryrule(1)
```

Figure 7.3. Control Rules for Depth-first Search.

The representation is complete when we initialise the control blackboard with a problem with single attribute 'name' having value 'stackabc', and three BLOCK objects describing the initial configuration (figure 7.2a). A specification of the goal state, figure 7.2(b), is also needed to ensure termination.

With reference to the six steps of the control algorithm, it should be clear that:

(a) the focus and policy objects are generated, then the task executor of figure 7.3(c) is applied. No attempt is made to generate other tasks.

(b) the task executor matches the first block and applies the first domain rule (since all blocks are 'on' something and all operators have this in their precondition.

(c) on subsequent cycles the algorithm scans down levels of the control blackboard. No new tasks can be generated so the executor is applied again. The last block generated will now be matched since it is top of stack.

(d) as block configurations are generated which are identical with ancestral configurations they are 'poisoned'. The executor will match with the last non-poisoned configuration on the next cycle (ie. we have back-up).

• All configurations use the same three blocks, so the search is exhaustive. Figure 7.4 describes the resulting search tree. Up to 30 block configurations may be generated before the goal configuration is found. Alternative orderings of domain rules may generate the solution (by chance) within a smaller search tree. Thus system performance tends to vary widely with this uninformed search methods.

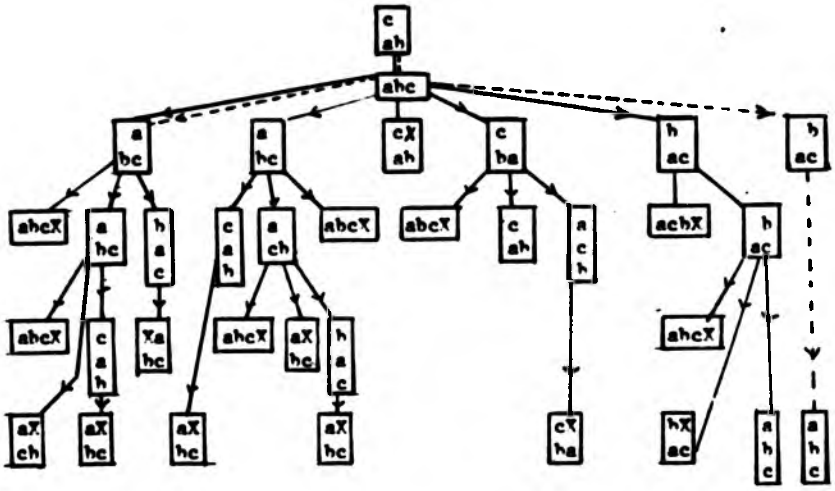


Figure 7.4. Superposition of Search Trees (X = poisoned).
(——— Depth-first Search - - - - Opportunistic Search)

Consider how a 'informed' strategy, that is a control regime that takes account of special features of the problem to narrow the search, might be represented. Note that the configuration where all blocks are on the table occurs frequently in the search. From this configuration one can attain the goal by simply putting b on c then a on b. This opportunistic strategy can be represented by the rules of figure 7.5 which are added to the knowledge base (ie. stack of rules).

```
 if focus(1) has       if policy(1) has              if policy(1) has
  &  name = block       &  on = c                      &  on = b
then  policy(1) has     & all block(s) have            & block(1) has
  &   on = c            & on = table                   &  name = b
  & policy(2) has       & a rule(1) has                &  on = c
  &   on = b            & block-actions='__on=c__'     & a rule(1) has
                      then  tryrule(1)                 & block-actions='__on=b__'
                                                     then  tryrule(1)
```

Figure 7.5. Rules Representing Opportunistic Strategy.

The control algorithm displays the following behaviour:

(1) the two new policies are generated first but none trigger a task executor since the necessary block configurations are not matched.

(2) backtracking permitted between steps (3) and (4) of the algorithm reverts attention to rules of the uninformed strategy. The executor of this strategy fires generating a new configuration of blocks.

(3) the control mechanism tries to apply executors of the opportunistic strategy again.

(4) the previous two steps are repeated until an unstacked configuration is generated, thereafter the executors of the informed strategy drive the search towards the goal configuration.

The search tree enscribed by dotted lines in figure 7.4 results. Note that rule 7.5(b) initially put a on c. The task was not fully specified, however, since the search is exhaustive, the intended configuration (ie. b on c) was soon generated.

The blocks example illustrates a number of points about ENCORES' control mechanism:

(1) opportunistic strategies may be combined with uninformed strategies,

(2) tasks need not be fully specified. The control mechanism will back

up to generate all alternatives,

(3) the control algorithm generates tasks of one plan before attending to another,

(4) the order of control rules in the stack is critical (eg. if the rules of the informed strategy were entered before other control rules, then the latter will make no contribution to a solution.

These points also apply to applications in knowledge-based programming since 'methods' are represented as groups of control rules like those of figures 7.3 and 7.5.

7.3. ENCORES' Control Algorithm.

The control algorithm, like ENCORES itself, is implemented in PROLOG. A simple loop, written in the procedural style described in chapter 5, establishes the control cycle. The user may supply a predicate, 'now_found', to detect and return the solution. The code is,

```
infer(Solution):-!,
        repeat,
        ( now_found(Solution);
        ( ( hypothesise(scan,1),  /*must change domain blackboard*/
            fail)).
```

If no clauses for predicate 'now_found' are supplied, the loop will apply all rules exhaustively.

The 'hypothesise' predicate encodes steps 2,3,5 and 6 described earlier. It succeeds by making one change only in the domain blackboard. The first parameter takes value 'scan' or 'backtrack' to indicate that it should display the scanning behaviour of a pure production system or backtrack during certain steps of the cycle. The second parameter is the number of the control blackboard level receiving attention for the generation of tasks. Consecutive numbers are associated with descending levels.

The predicate only succeeds when the RULE level of the control blackboard is receiving attention and a task executor has fired,

```
hypothesise(_,Levelnum):-
        get_level(rule,Levelnum,control),
        get_rule(control,rule,Nextrule),
        generate_objects(control,Nextrule,_,_),!.
```

where 'get_rule' identifies the level number as being that of RULE. Procedure 'get_rule' retrieves rules according to the blackboard and level where they make their contribution. As used above, 'get_rule' will return the next task executor. Procedure 'generate_objects' invokes the lhs- then rhs- evaluators in an attempt to execute the rule,

```
generate_objects(Blackboard,Rule,Focuscxt,Ruleactivation):-
        applicable(Blackboard,Rule,Focuscxt,Ruleactivation),
        try_rule(Blackboard,Ruleactivation), !.
```

Note that backtracking is permitted within and between these two clauses until a task executor is found and fired.

A second clause of 'hypothesise' attempts to generate tasks on descending levels, backtracking if need be, until a task executor is fired. The code is,

```
hypothesise(_,Levelnum):-
        get_level(Levelname,Levelnum,control),
        Levelname \= rule,
        get_rule(control,Levelname,Nextrule),
        generate_objects(control,Nextrule,_,_),
        Nextlevel is Levelnum + 1,
        hypothesise(backtrack,Nextlevel).
```

In contrast, the following clause allows attention to focus on descending levels until a control rule is found which will fire,

```
    hypothesise(scan,Levelnum):-
        get_rule(Levelname,Levelnum,control),
        Levelname \= rule, Nextlevel is Level + 1,
        hypothesise(scan,Nextlevel).
```

The control algorithm finds easy expression in PROLOG, given the language's declarative semantics and in-built backtracking. A rendering of the algorithm in an imperative language would be correspondingly obscure.

7.4. An Event List for Uninformed Search Strategies.

Since ENCORES does not record rule activations in an agenda, it is necessary to ensure that no blackboard object receives the systems attention if it cannot contribute to the generation of new objects. Objects in duplicate contexts should not receive the system's focus again so the context is poisoned, that is, removed from further consideration. With the depth-first search method described above, another such possibility is presented. All possible rules may have been previously tried with an object.

It is not desirable that such objects be poisoned; they may yet serve to trigger a rule in conjunction with another focused object. A list of such objects is therefore maintained by the system. Each object, or 'event', is labelled as 'closed' and the system will not focus on it in future. These labels are currently implemented as PROLOG assertions (not FACTS, as described in section 5.3, page 96). The usefulness of this device remains to tested with sizable programming problems and knowledge bases (where the width of search trees may render uninformed search useless in any case).

7.5 Summary.

The control cycle was described both informally and formally. A simple block stacking example was used to explain the interplay of the stack organisation and refocusing with back-up and opportunistic strategies.

Direct application of rules minimises the cost of rule evaluation at each inference step. However, back-up is costly since the evaluation must be repeated. The control mechanism therefore differs from that of OPM where agendas are used to record all rule activations with focused objects. One might also view this difference as an implementation feature where the rule activations of ENCORES are supplied on demand by a process similar to lazy evaluation of functional programs [Henderson80].

Certainly units of time, such as 'control cycles', play no role in the representation of control knowledge in tools built with ENCORES. For this reason we suggest the control mechanism may only find application in other non-time critical domains where search trees tend to be wide and where the layering of the blackboard architecture can be exploited. One such domain may be VLSI design.

CHAPTER 8.

BLACKBOARD-BASED SOFTWARE ENVIRONMENTS : FUTURE DIRECTIONS.

The final chapter sketches some future directions of this work, in particular to overcome inadequacies observed with ENCORES and extend the interpretation of the blackboard architecture to support conventional programming methods. It will be shown, in both cases, that the key to better programming support lies in integrating more information within blackboards.

The software and tools described in chapters 5 to 7 comprise ENCORES, a prototype environment for testing and demonstrating the concepts of chapters 2 to 4. Intended as a "laboratory" for investigating knowledge-based programming, the system has not, as yet, seen any hard use. However, attempts to represent the knowledge embodied in the author's previous blackboard environment within ENCORES, suggests we now have a more transparent system for acquiring and using programming knowledge. These efforts have also revealed many directions in which ENCORES can be improved, both as an interpretation and implementation of the blackboard architecture.

One promising direction is the design of explanations for users of the environment. Some insight may be gained on the application of knowledge to a particular problem by viewing the net of contexts - if one is familiar with the knowledge base. However, casual users of a tool built with ENCORES will gain little insight from this information. Rather, they will demand some explanation of how the tool solved the particular problem. Fortunately, a rich variety of explanatory forms may be composed from the task structure and rule activations maintained by ENCORES. Some of these possibilities will be sketched.

Knowledge based tools, like those described in chapter 1, may in future reduce the role of, or even eliminate, the human agent from software development. Until then, large systems will be built with not one, but many experts cooperating within some software project. If these individual efforts can be integrated in an effective manner then there is less scope for error and the software product will be more reliable. Central to this integration is the maintenance and communication of all records concerning the software system. Such a data base must be resilient in the face of changing requirements and be the hub around which project members contribute to system development.

Software environments, known as Integrated Project Support Environments (IPSEs), are under development which are intended to fulfill the tooling and information recording needs of software projects [McDermid85]. These environments are expected to evolve through three generations : each succeeding generation offering better integration of information and tooling by being based on a more flexible data model. The data model for the third generation is not defined but must serve knowledge-based as well as conventional tools [Dignan84].

To begin this chapter, we report initial experience of knowledge engineering within ENCORES. Some problems encountered have stimulated ideas for extending the interpretation of the blackboard architecture and improving its implementation within ENCORES. The potential for supplying explanations to users of ENCORES will be briefly explored. Finally, it will be shown that an object-centred blackboard might be used to flexibly serve the recording needs of an advanced IPSE and, in addition, integrate knowledge-based programming tools.

8.1 Knowledge Engineering with Encores : some initial experiences.

The work described in this thesis was motivated, in part, to support the automatic programming of algebraic specifications (see footnote, page 19). Since this work began, this investigation has taken new directions. Thomas [Thomas87] has put the selection of representing types for algebraic specifications of abstract data types on a more formal footing. The specifications are analysed in terms of graph structures which model the behaviour of their operations. Efficient representing types can then be chosen.

The representation tools of ENCORES cannot reference graph structures directly, so the system was tested with the small knowledge base developed with the author's previous environment [McArthur86]. The complete knowledge base is reproduced in APPENDIX C. This choice was appropriate given that the current implementation leaves little PROLOG workspace on the development machine (DEC PDP11/70) to search for solutions. Despite this constraint it has been demonstrated that ENCORES overcomes the shortcomings observed with the author's earlier blackboard system.

8.1.1. Preliminary Evaluation of ENCORES.

A system may be evaluated along a number of dimensions. One can ask the following of a system building tool:

(1) Is the knowledge representation scheme adequate or does it need to be extended?

(2) Is it easy for users to interact with the system?

(3) What facilities and capabilities do users need?

(4) Is the system response time adequate?

Ultimately, "..the best way to evaluate a system is to get it built, turn it over to friendly users, and solicit and respond to their feedback"[Gaschnig etal83].

The context of the Alvey project in which this research took place afforded the means to obtain feedback and so arrive at an informal evaluation of ENCORES. The author's first attempt to provide an environment in which knowledge bases for automatic programming could be accumulated and tested brought a negative response from the project team.

Specifically, my colleague, Muffy Thomas, was given details of the rule language and asked to supply some transformation rules for insertion in the knowledge base. Ms. Thomas supplied criticism but no rules. The language was described as tortuous in the extreme and the process of program transformation on the blackboard opaque.

At this point (late 1986) my supervisor, Charles Rattray suggested that I conduct a wider survey of the research literature and design another blackboard-based environment for knowledge-based programming, this time with the goal of ease-of-use to its intended users being paramount. The clarity with which blackboards and programming knowledge can be described in the ART (page 31) and CHI (page 11) systems, respectively, pointed the way forward. The Alvey team also gained a research assistant, Jean McInnes, whose duty was to mediate between the mathematically orientated Ms. Thomas and myself. Thus Jean was in the perfect position to provide feedback on ENCORES to me whilst translating the automatic programming developement model of Ms. Thomas to the prototype knowledge base.

Implementation of the APPEAL language was complete in late 1987. Jean McInnes was given the initial task of translating the earlier rule set collected for SPECTRE into APPEAL rules. Feedback was now positive : the new language was described as being much easier to use than the old language. Further, the APPEAL rule list was passed round other members of the project team. Each member reported the intent of these rules being clear from their reading.

Confident in the use of APPEAL, Ms. McInnes has gathered and is testing a rule set which can automatically create the specification rules associated an algebraic specification (such as those at the EQUATION level in the blackboard of figure 2.5, page 40). This effort has revealed problems with ENCORES which will be described shortly.

Also listings of ENCORES blackboards were passed round the project team and members invited to guess which program transformations had taken place. Again, there was little difficulty reported on interpretating these listings.

The new choice of blackboard components improve knowledge engineering support since:

(1) Blackboards are more transparent and organised more economically. Most of the blackboard of figure 2.5 (page 40) has been generated (in a linear rather than graphical form) using the APPEAL rules described in chapters 3 and 4. Tables describing contexts clearly distinguish alternative lines of reasoning. Associated PROLOG assertions, describing linkages, reveal relationships between contexts at adjacent blackboard levels. In contrast, the author's previous system presented blackboards as lists of assertions such as,

```
datatype(1, stack, typecontr(stack,of_types(typevar(alpha)),unknown).
operation(4,top, signature(in_types(typecontr(stack,typevar(alpha)),
                                    out_type(typevar(alpha))), unknown).
operation(24,top,signature(in_types(typeconstr(stack,typevar(alpha)),
                                    out_type(typevar(alpha))), reads).
```

where each assertion describes an object and inner terms are values for an ordered set of attributes. Alternative lines of reasoning were opaque in such assertions as were the programs under development (from their attributed tree representation). Consistent descriptions of programs at different blackboard levels were difficult to discern (using the AGE-like pointers in the 'and' and 'or' terms).

(2) The APPEAL representation language is more accessible. The examples of APPEAL rules given in chapter 3, taken from the knowledge base of the previous system, have been communicated readily to colleagues in the project team and entered to ENCORES via the editor described in section 6.5, page 134. Although this editor is

primitive (no deletion or viewing of rules is possible), acquisition of programming knowledge is much improved. The symmetry of APPEAL rules, led to the quick discovery of an alternative form of the fold transformation. In contrast, knowledge bases had to be prepared as files of PROLOG assertions for the previous system, with each assertion requiring painful bracketing both to differentiate clause structure and tree-matching templates contained therein.

(3) Control knowledge is explicit and more perspicuous in the new system. The four step metaplan for developing algebraic specifications (described in section 2.4.2, page 38) was readily represented with the APPEAL rules shown in chapter 4. In contrast, control knowledge had to be programmed directly as PROLOG code in the previous system; this code being one 'step' in its AGE-like control cycle.

Thus we are now in the position to answer the first two questions posed at the beginning of this section. The knowledge representation scheme is superior to that of the author's previous blackboard systems. Although too early to claim that it is 'adequate', we are assured that it provides a foundation for our subsequent investigations of knowledge-based programming. Ease-of-use has been achieved.

Certainly, further tools can be added to increase the utility of the system. Some of these will be described in the next section. Until a development machine larger than a PDP is available further tooling must wait. Similarly, the response time (of the order of 10 seconds to fire each rule) will be improved readily with a larger machine with no adjustment to the software.

8.1.2. Problems and Further Work with ENCORES.

New problems inevitably presented themselves. For each blackboard component, these are:

(1) Programs cannot be decomposed when descending blackboard levels. The blackboard of figure 2.5 (page 40), to program the stack specification clearly reveals this problem. Given a choice of representing type (the singly linked list), all operations of the stack must be related to those of the chosen type with the generation of specification rules in a single context at the equation level of the blackboard. This is not expressible in a single APPEAL rule. One would prefer to use rules to both generate and subsequently transform one specification rule at a time. Such an approach would give rise to a root context for each operation and each would be treated as alternative specialisms of the specification. Decomposition of programs must be represented explicitly within the blackboard structure. Information must be associated with abstraction links to show that the context has been decomposed (eg. using the 'AND' links of ACE). In our stack problem, this would permit each specification rule to be transformed separately into the equation of the implementation, yet all may be collected via link information to assemble the final program.

(2) The APPEAL language does not contain constructs to effect decomposition. APPEAL was designed with a minimum of constructs

for simplicity of use. In support of the amended blackboard structure suggested in (1), APPEAL might be enlarged to make explicit reference to contexts (just as explicit reference to 'viewpoints' is permitted in the rule language of ART). One might also introduce quantification over subsets of objects in a context, for example, with a 'for each' construct.

(3) It is not possible to select objects for focus on the basis of their order in the stack. Consider the representation of breadth-first search with use of control rules. One might imagine a rule of form,

```
    if an <OBJECT> has
     &      <ATTRIBUTE> = Any
     &  all rule(s) have
     &      <OBJECT>-conds = '__<ATTRIBUTE>=Any__'
     then  tryrule(s)
```

might be adequate. However, for this form of search we wish to apply rules to each new object generated in sequence. Since the first generated is not top of stack and we do not know its identity, we cannot access it. Again, explicit reference to contexts may be one approach to this problem.

(4) The construction of parsers for languages of an application is difficult. In the current system, a parser must be written in PROLOG to convert both programs and APPEAL metalanguage expressions to internal tree representations (see APPENDIX-D). Since metavariables and their matching types must be gathered, the code does not take the simple form (of a definite clause grammar) as

used in the previous system. Further, all possible configurations of program constructs and metavariables must be anticipated, thus the parsers are large PROLOG programs. Code to handle expressions with infix operators is particularly tortuous to write.

The CHI system (page 11) overcomes these problems by incorporating all descriptions of an application language within the knowledge base. Both internal (abstract syntax) and external (parse syntax) descriptions are explicit and accessible within the system. Similar descriptions might be retained in the GLOBAL area of ENCORES' blackboards. Parsing and pretty- printing of language expressions might then be accomplished with general purpose code within the system and the burden of parser writing removed.

Minor problems arise with ENCORES, the performance program, due to the small PROLOG workspace afforded within the DEC PDP11/70. ACSYS, the APPEAL editor and the consultation software must be run as separate programs. The major programming task outstanding is the enhancement of the APPEAL editor for the management of methods, that is, sets of control rules with a pre-assigned priority. In the current implementation, control rules must be entered in a strict order : lowest priority rules (representing a method) first and highest priority rules last.

In addition to these particular problems, it should be noted that the fundamental problem of transforming a program from one formalism to another remains unsolved. APPEAL rules may be used to change certain constructs of a program from one language to another but the resulting intermediate state of the program may be meaningless (see Wile's comment, page 12). We skirted this problem by working entirely in the HOPE language for the stack example. One might approach this problem by applying sets of APPEAL rules together or employ 'temporary blackboard space' (a concept we will discuss shortly) to record intermediate results. The problem awaits further research.

8.2 Explanations for Knowledge-based Programming.

The task structure recorded in the control blackboard and the record of activations of APPEAL rules might be used to compose a rich variety of explanations to users consulting a tool built with ENCORES. Yet such variety is necessary, for according to Hasling [Hasling etal84], "..explanations should not presuppose any particular user population...should be informative... should be concrete or abstract, depending on the situation...should be useful for the designer, as well as the end user...and should be possible at the lowest level of interest. Higher level explanations can later be generated by omitting details below the appropriate level".

Such a variety is possible within a blackboard environment by virtue of the rich structuring imposed on the search tree and knowledge sources. Some possibilities for explaining how a program was developed with use of blackboards are:

(1) Trace the development of objects and rules which generated them along any one level of the domain blackboard. With reference to the blackboard for developing the stack specification (figure 2.5, page 40) and the description of its transformational development (page 64), the might respond to query 'HOW ---top(nil)<=error;'. with,

```
to  '---h(top(nil))<=top(h(nil));' we applied rule,
  if  an equation(1) has  (activation of unfold rule, page 62)
   &    code = '---h(top(nil))<=top(h(nil));'
   &    top(h(nil) = '__h(nil)__'
   &  an equation(2) has
   &    code = '---h(nil)<=newstack;'
then  an equation(1) has
   &    top(newstack) = '__newstack__'
   &    code = '---h(top(nil))<=newstack;'
  to  '---h(top(nil))<=newstack;'  we applied rule, etc.,
```

with the equation in the query being concluded at the end of this chain. One might also offer an explanation in terms of major design decisions taken at higher blackboard levels (eg.'a singly linked list was chosen as the representing type').

(2) Trace the sequence of development of objects and rules which generated them at all levels of the domain blackboard. Explanations might be presented as for (1), but note that a very large chain is likely to result.

(3) Trace the sequence of tasks along any one level of the control blackboard. With reference to the control blackboard of figure 4.5 (page 87), the system might respond to query 'HOW --- top(nil)<=error;' with

```
we did the following:
applied 6 domain rules to complete tactic 'findactions',
applied 1 domain rules to complete tactic 'represent',
applied 5 domain rules to complete tactic 'implement'.
```

(4) Trace the development of tasks on all levels of the control blackboard. Explanations might be presented as for (3) so revealing the complete programming method that was used.

Other possibilities are offered with queries as to how specific tasks were achieved. The domain rules used might then be quoted. A more general approach might be to use an 'explanation schema' supplied by the user. This might take the form of a triple, '<BLACKBOARD>,<LEVEL(s)>,<RULEFORM>', where 'LEVEL(S)' name the classes of objects of interest which is to be traced with relevant rules presented with a 'RULEFORM' of 'activated' or 'non-activated' as specified.

During a particular consultation with ENCORES the user might interrupt the development to enquire 'why' a particular rule is being used. Explanations to this query may be composed by a reverse process to that described for 'how' type queries. Rule activations may be retraced or the task structure ascended to compose a rich variety of explanations.

Finally task structures, recorded within ENCORES, may themselves be fruitful objects of study. One motivation for developing the PADDLE language (see chapter 4, page 78) was to investigate whether developments, recorded within PADDLE goal structures, might be rerun on similar specifications, possibly after some modification. Since the task structures of ENCORES are factored along abstraction levels of the control blackboard, one might factor their study along these levels. Portions of the task structure on different levels may be rerun on similar problems to determine the level of detail on which it requires modification.

8.3 A Blackboard-based Integrated Project Support Environment.

First and second generation IPSE's have used increasingly elaborate data models (files and relational database, respectively) as their foundation, so we might expect a similar leap in sophistication of the data model underlying the third generation. In chapter 1 we argued that the blackboard architecture provides a flexible framework for applications in knowledge-based programming. Here we argue that the significant features of the architecture may also serve the recording needs of an advanced support environment. Thus the architecture should be considered a strong candidate as the evolutionary stage on which a third generation IPSE may be set.

In the following sections I describe the features of the blackboard architecture which make it an attractive framework for an IPSE; describe how both current and knowledge-based tools might be integrated within a blackboard-based IPSE; and finally show how a general-purpose human computer interface may be developed for such a support environment.

8.3.1. Adapting the Blackboard Architecture for an IPSE.

The 'multiple expert' metaphor commonly used to describe the Blackboard Architecture suggests how it may be adapted for use within an IPSF. Figure 1.3 (page 16), depicts the blackboard as a database on which partial solutions to some AI problem can be represented. The knowledge accumulated from human experts in the domain is modelled in a number of so called 'knowledge sources' which react to, and propose further partial solutions. The first step in adapting the model for an

IPSE is simply to take the metaphor as the reality : the blackboard will record the software system under development, and software engineers (the human experts) will make their contribution to the blackboard with use of software tools.

In chapter 2, we showed that blackboards can be used to record a program under development. Further, this information is conveniently factored along a number of dimensions since:

(1) The ABSTRACTION LEVELS of the blackboard distinguish the representations of of the software system in use.

(2) ALTERNATIVE SOLUTIONS are modelled within the network of elements at any one blackboard level. In other words, an explicit search space is managed, preferably using some CONTEXT MECHANISM (as argued on page 33). Of these factorisations, it is not difficult to show how ABSTRACTION and a CONTEXT MECHANISM may be exploited by an IPSE. The advantages of using a particular interpretation of the architecture (eg. based on objects) will be described later.

The concept of abstraction is very powerful in AI work and is particularly attractive for recording system development in an IPSE. In section 1.1 (page 2), we described some of the high level representations introduced in software engineering for describing software systems. System design typically proceeds through multiple representations, each of which should be distinguished.

Consider a possible blackboard for a software project shown in figure 8.1. Information about the development may be recorded at the 'level' of:

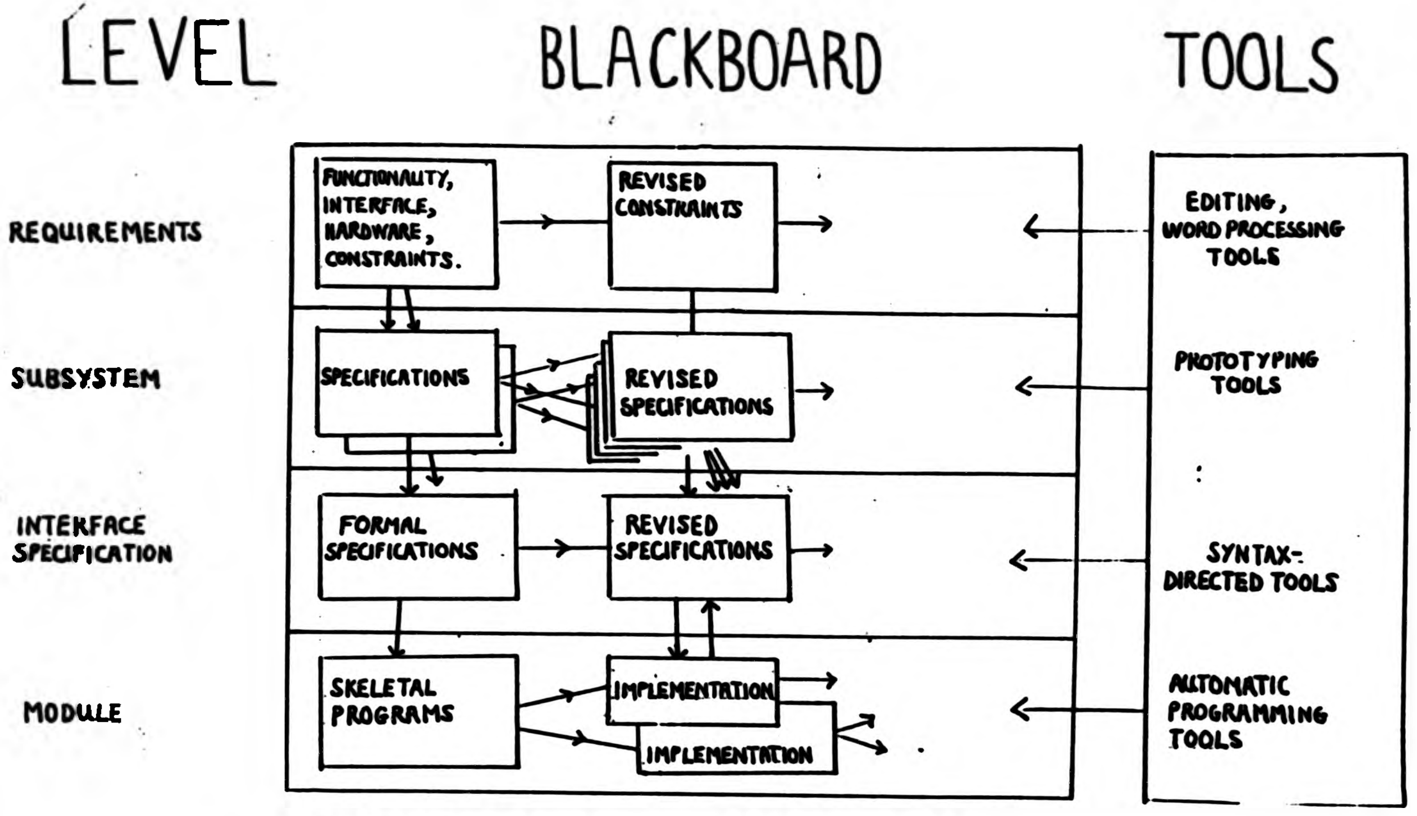


FIGURE 8.1. BLACKBOARD FOR A SOFTWARE PROJECT.

(i) REQUIREMENTS. Documents at this level would outline constraints on the system imposed by the client.

(ii) SUBSYSTEM. Major design decisions might be recorded in documents at this level.

(iii) INTERFACE SPECIFICATION. Formal specifications of lower level modules would be recorded at this level.

(iv) MODULE. Implementations of the above specifications would be recorded here.

Note that the figure uses a similar decomposition of blackboards to the object classes and linkages as the blackboard structure advocated in chapter 2. Regardless of the particular blackboard structure in use, techniques of inheritance can be utilised in the implementation of the structure to optimise data storage.

Such a blackboard for software development would be expected to 'grow downwards' as work proceeded from design to implementation. Project team members concerned with major design decisions will generate documents at upper levels of such a blackboard. Programmers or automatic transformation tools would be expected to generate documents at the lowest levels. Thus the recording of system development at different levels of abstraction provides a clear demarcation of system details to meet the needs of both individual responsibilities and the appropriate software tools which serve them.

A primary concern in the development of architectures for IKBS is the management of search amongst alternative solutions to AI problems. The software project team is also typically confronted with many alternative design and implementation choices some of which will be explored. Thus an IKBS architecture that maintains data representing alternative choices may flexibly serve the recording needs of software engineers and the management of search space for IKBS tools.

Context-sensitive data models like those of CONNIVER [McDermott&Sussman72] and ART (page 31) open new opportunities for maintaining integrity and reusability of documents pertaining to the software development process. During development more than one design or implementation may be considered and these may be recorded in separate documents but both linked to the same mother document. One of the competing documents will be chosen for the next stage of system development but the others should not be discarded. On subsequent debugging or maintenance earlier documents and the decisions they contain may be revised and the rejected documents may now seem more attractive as the basis for a revised software development. Earlier decisions or specifications should be both recorded and be reusable.

8.3.2. Integrating Conventional and IKBS Tools.

IPSEs should transcend conventional tool environments, such as that of UNIX [Kernighan&Pike84], in the management of documents pertaining to a system development to ensure integrity and ease of usage by project staff. One can view such documents as describing a 'state' of the system in the history of its development. A high level of integration of both documents and tools can be achieved in a blackboard-based IPSE in which software tool usage is seen as the examination or generation of new 'states' of the software system.

The linkages or pointers used to construct the search graph for an IKBS application can be used in a blackboard-based IPSE to both model the time development of software documents and their dependencies. We saw that ENCORES uses three forms of linkage to model:

(a) an ABSTRACTION of some previous state on a lower blackboard level,
(b) a SPECIALISM of some previous state at a higher blackboard level,
(c) a REFINEMENT of some previous state at the same level.

A blackboard-based IPSE might employ similar pointers between documents which are both accessible to, and generated from tool usage.

Finally, careful consideration must be given to the representation of documents themselves to ensure the highest level of integration of software tools. For example, source code documents composed in any formal language might be represented as attributed trees of an abstract grammer . Such a representation has been employed successfully in software development environments with an integral suite of context-sensitive tools (eg.MENTOR [Lang85]) and knowledge-based programming environments requiring efficient pattern-matching eg. CHI (page 11) and ENCORES.

A perspective of software documents as states demands that we distinguish conventional tools for the analyses of documents from those used to change them. Use of browsers and debuggers, for example, would not change the blackboard. Editors, compilers and word-processing tools will generate new or modified documents and the IPSE should extend the blackboard using the sort of pointers described above.

Documents in some formal language may be retained internally as attributed trees and tools expecting this representation, such as a browser, editor and syntactic debugger, should integrate well and present a common interface to the user. Pursuit of a common MMI leads to exciting possibilities when we raise our sights beyond a single document. Section 8.3.3 outlines one such possibility.

The choice of specific blackboard for a project will reflect the development paradigm in use by the project team. As formal methods come into wider use we might expect the requirements and specification documents on upper blackboard levels to be expressed formally and thus be amenable to manipulation by the context sensitive tools discussed above. However, the most exciting possibilities are opened by combining use of formal methods with IKBS tools.

Linkages, of the sorts described above, may be used to maintain integrity of state information during search by a knowledge-based system as well as describe document history in an IPSE. However, we must be careful to distinguish use of 'blackboard space' for documents which will be considered permanent from the search space that might be needed by an individual knowledge-based or transformation tool.

Recent research on distributed blackboard environments [Corkill&Lesser83] has introduced the concept of 'local blackboard space' for separate use by an intelligent agent. The similarities to conventional concepts of 'workspace' and virtual memory should be obvious. In general, the blackboard schema chosen to describe the arrangement of documents in an IPSE will not be suitable for representing state information used by an IKBS tool such as that employed with ENCORES. Thus a blackboard-based IPSE must maintain temporary storage and possibly specific blackboard schemas for the integration of IKBS tools. We should note however that the blackboard management software can both serve the IPSE and individual IKBS tools.

The sorts of knowledge-based software tools that might be incorporated in such an IPSE were described in section 1.1 (page 2). Tools were described which are designed to assist the programmer produce or maintain software along conventional lines. All such tools demand manipulation of a search space and could be implemented within a. blackboard architecture. Most however are research prototypes requiring considerable development before they will be generally useful. One exception is the PIE system [Golstein&Bobrow81], which does not use a knowledge base but offers techniques for maintaining alternative versions of software systems using context sensitive databases.

The adoption of formal methods for software development offers, not only precision of system specification, but also the possibility of automatic transformation to executable code. Some such transformation systems were described in section 1.1.1 (page 3). None of these systems are yet generally useful. Rather, research must continue with the transformational approach for some years, probably supported with environments such as ENCORES. Yet, in anticipation of success in this endeavour, one might sensibly adopt an architecture for an IPSE in which transformation tools may be readily incorporated with use of the sort of knowledge representation tools developed for ENCORES.

8.3.3. A General-purpose MMI for a Blackboard-based IPSE.

In the previous sections we have seen that by recording the development of an information system on a blackboard we impose a rich structuring on the documentation which is in many ways novel. Rather than develop a plethora of new software tools to take advantage of this structure an attempt should be made to simplify and combine the many functions that might be envisaged into a general purpose man-machine interface (MMI).

We propose that documents be recorded as objects within a blackboard as proposed in chapter 2 and that this be open to view by IPSE users (not subject to access restrictions) through window management software. Further, each software document may be viewed in more or less detail. Such an MMI combines the object-centred representation within ENCORES, the 'viewpoints' concept of the ART system and the 'detail hiding' concept of syntax-directed software tools.

The advantages of ART's viewpoint mechanism were described in chapter 2 (page 32) and are adopted in ENCORES' object-centred blackboard structure. An interface built round these objects might be perused to reveal erroneous use of a tool or knowledge source, reveal alternative versions of a program that might be developed, reveal more abstract or specialised versions of a program, or trace the complete history of the software development.

Syntax-directed tools make use of the attributed tree representation of software to assist the programmer reveal the structure of her code. Not only might the outer structure of a software document be revealed whilst inner details are hidden, but the inner structure may also be incomplete, thus encouraging good structuring during its composition. The essential point is that tree traversal and composition are the underlying operations performed by such tools.

Since both the network of documents in a blackboard-based IPSE and individual software documents are graph structures, no great difficulty is foreseen in the development of software to present any portion of either through a window. A single command set might combine many possible functions and access alone might be provided by a few operations relegated to a mechanical aid such as a 'mouse'.

The blackboard architecture therefore offers attractive features for an advanced support environment. Software engineering has introduced methods and tools to aid programmers produce explicit, formal descriptions of systems at each phase of their activity. Since these descriptions are hierarchical, reflecting their origins in the programming craft, they should be recorded for project use in a support environment with a hierarchical architecture. The blackboard architecture is both hierarchical and highly flexible so commending itself as the data model on which to base an advanced IPSE.

Although our emphasis here is use of a domain blackboard to record software development, use of a control blackboard in an IPSE is also worthy of investigation. Such a blackboard might be used for posting management decisions or recording progress regarding the process of the software development. Of particular interest is process information regarding security and access, tool usage and schedules of activity on the domain blackboard.

Finally, it should be observed that this IPSE proposal is largely concerned with issues of architecture and man-machine interface. It is not orientated to the use of particular development methods or tools. In contrast, the work of Cottam and Jones [Cottam etal.85] seeks to develop an advanced support environment centred on the use of the Vienna Development Method and associated tools. The proposed work is seen as being closest to the IPSE 2.5 project [Snowdon etal.85] which also aims for a product which is generic and will serve a wide range of project management and engineering needs.

8.4 Summary

The final chapter revealed some oversights in the interpretation of the blackboard architecture and suggested further applications of blackboards for the programming domain. The concepts embodied in ENCORES were shown not to be perfect. Yet this was inevitable : not all uses of a knowledge-based programming environment can be foreseen. Instrumentation may yet be added to a laboratory for investigating knowledge-based programming as needs arise. Some new needs are clear : the blackboards and knowledge representation language must support explicit decomposition of contexts and blackboards must incorporate descriptions of the application languages in use.

The prospects for delivering explanations in support of knowledge-based programming are exciting. A rich variety of explanatory forms remain to be investigated. The task-orientated control scheme, devised for NEOMYCIN for the purposes of tutoring and explanation, is captured in another form in ENCORES. Thus the system is an excellent test-bed, not only for knowledge-based programming, but also studies of tutoring and explanation in the programming domain.

It has been reported [Polak86] that Kestrel Institute now view their CHI system as a springboard to an advanced support environment. So too has the author's experience with ENCORES been inspiring. However, the layered architecture of ENCORES holds greater promise for encompassing more of the software development process.

APPENDIX-A. MODULE : A tool for developing large PROLOG Systems as sets of small modules.

The MODULE tool permits large PROLOG systems to be developed as sets of small software components, preferably written as abstract data types and algorithms using these. The components, or modules, may be overlayed thus permitting large systems to be run on machines which would normally offer insufficient workspace (eg. micros, DEC PDP). Modules which are altered may also be stored so that PROLOG databases may be written back to store after updating.

The tool requires that the PROLOG code for a system be written in a special style. The code must conform to the following:

(1) All modules must have an assertion as to the list of predicates it contains : this is the 'export list'. The assertion should have the form,

```
export(<FILENAME>,[FUNCTOR1(<ARITY1>),.....FUNCTORn(<ARITYn>)]).
```

Note that the file name of the code is treated as the module name. The FUNCTOR's are the PROLOG procedure names and the ARITY's are integers for the number of parameters of each procedure.

(2) Each module must have assertions as to the modules which it uses, if calls are made to procedures in these modules. The assertions have the form,

```
uses(<FILENAME>,<USED-MODULE1>).
uses(<FILENAME>,<USED-MODULE2>).
...etc....
```

The tool contains three procedures which comprise an abstract data type on these modules. The 'load' procedure consults module(s) from file (if not already consulted) and consults (recursively) all modules used by it (them). The 'unload' procedure retracts all predicates in the named module(s) and all predicates from modules it uses (if no other module uses them). The 'store' procedure writes the code of a single module back to the file name including the export and uses assertions.

The 'load' procedure can be used to consult a complete PROLOG system to the PROLOG workspace. The user first consults the tool itself to the workspace then issues the query 'load([<MODULE-NAME>])'. If the code of MODULE-NAME is the calling code of the system then the complete system will be loaded.

Modules are overlayed by using the 'load' and 'unload' procedures from within PROLOG procedures. However, modules are not overlayed by simply issuing a 'load' call, then at a later point, issuing a 'unload' call. The workspace is not freed until backtracking progresses back beyond the original 'load' call. This backtracking can be forced by enclosing the calling code inside a 'repeat.....fail.' loop. As all variables used as parameters to the overlayed code become unbound due to backtracking, results must be returned via assertions from the overlayed module(s). If the overlayed code is used for its side effects only then some assertion must still be made to signify that it was successful. In some PROLOG systems, a call to the garbage collector will also be necessary after the 'repeat' statement (for example, this is 'trimcore' in the NU-7 interpreter). The body of procedures which overlay a module will have the form,

```
repeat,
        (retract(<ASSERTION OF RESULT>);
         (load([<MODULE-NAME>],
          <CALLS TO PROCEDURES IN MODULE>,
          unload([<MODULE-NAME>]),
          fail)
        ).
```

The 'store' procedure, if used in the same manner as 'unload', will write the code back to the file corresponding to the file name. (Note that the code for 'store' has not been tested extensively).

The code for the three procedures is given with suitable commentary below.

```
/* 'load' overlays modules, and all modules which are used by it, */
/* and all modules used in turn etc. A check is made to see if any*/
/* module is already loaded. eg load([mod1,mod2]), overlays these*/
/* 2 modules, and all modules which they use.          */

/* case, empty list */

load([]):-!.

/*case, module already loaded (since its predicate list found)*/

load([Module|Restm]):-
        export(Module,_),
        !,
        load(Restm).

/*case, module not already loaded */

load([Module|Restm]):-
        consult(Module),
        !,
        load_child_mods(Module),
        load(Restm).
```

```
/* case, fatal error if cannot load module */

load([Module|_]):-nl, write(Module), write(missing), abort.

/* 'load_child_mods' takes each 'uses(<mod>,<used-mod>)' assertion*/

/* for the given <mod> module and calls load for each <used-mod>*/


load_child_mods(Module):-
        uses(Module,Child),
        load([Child]), /*only succeeds once*/
        fail.

/*case, end fail loop*/

load_child_mods(_).


/* 'unload' removes all predicates associated with a module*/

/* (as given in 'export' predicate for the module) and removes*/

/* modules used in turn by this one provided no other module*/

/* uses them. */

/* eg. unload([mod1,mod2]) removes all predicates in these two*/

/* and all predicates in modules used by them*/

/*case, empty list*/

unload([]):-!.

/* case, module to unload*/

unload([Module|Rest]):-
        retract(export(Module,Predlist)),
        !,
        retract_preds(Predlist),
        retract_used(Module),
        !,
        unload(Restm).
```

```
/*case, error if unload fails*/


unload([Mod|_]):-
        nl, write(unloaderr),
        write(Mod),
        abort.


/* 'retract_used' unloads modules that are used by module*/

/* being unloaded (provided no other module is using them) */


retract_used(Module):-
        retract(uses(Module,Usedmod)),
        not(uses(Othermod,Usedmod)),
        unload([Usedmod]), /*succeeds only one way*/
        fail.

/* case, end fail loop*/


retract_used(_).


/* 'retract_preds' retracts module predicates and those*/

/* possibly generated by it (local variables) */

/* eg. retract_preds([f(2),g(1),h(3)])  retracts 3 predicates*/

/* with arity 2,1,3 respvly*/



/*case, all retracted */

retract_preds([]):-!.


/* case, retract next predicate*/



retract_preds([Term|Rest]):-
        Term=..[Func,Ar],
        functor(Predhead,Func,Ar),
        retractall(Predhead),
        !,
        retract_preds(Rest).


/*'store' takes export(<mod>,<predlist>) assertion and writes*/
```

```
/* predicates back to store, incl. associated 'export' and*/

/* 'uses' assertions. Nothing is retracted. Treatment of new*/

/* module 'versions' not yet implemented, thus overwrites file*/

/* eg. store(mymod), puts predicates of 'mymod' back in file mymod*/


/* case, module storage as normal */


store(Module):-
        telling(Output), tell(Module),
        export(Module,Predlist),
        write(export(Module,Predlist)),
        write('.'), nl,
        store_uses(Module),
        store_preds(Predlist),
        told, tell(Output).

/* case, module not found*/


store(Module):-
        told, tell(user), nl,
        write(erronstore), write(Module), abort.


/* 'store_preds' writes each predicate of Predlist back*/

/* back to file Module*/


/* case, all written*/


store_preds([]):-!.


/* case, store a predicate */


store_preds([Term|Rest]):-
        Term=..[Func,Ar],
        functor(Predhead,Func,Ar),
        write_pred(Predhead),
        !,
        store_preds(Rest).


/*'store_uses' writes 'uses' assertions for Module back */

/* to the file with name Module */
```

```
store_uses(Module):-
        uses(Module,Usedmod),
        write(uses(Module,Usedmod)),
        write('.'), nl,
        fail.


/* case end fail loop */


store_uses(_).



/*'write_pred' more selective than 'listing'.*/

/* eg. write_pred(f(X)) writes clauses with head 'f(X)'*/



write_pred(Predhead):-
        clause(Predhead,Predbody),
        write(:-(Predhead,Predbody)),
        write('.'), nl, fail.


/*case, end fail loop */


write_pred(_).
```

APPENDIX-B. MULTIFACTS : A Tool for Managing Associative Triples in Multiple Databases.

The data model used within MULTIFACTS is an extension of that developed for the FACT machine (see section 5.3, page 95). The basic unit of data is a labelled triple (known as a "fact") designated as belonging to a particular database. Retrieval of data (deductive or associative) may be specified for a particular database or a 'global' data area from which set properties may be propagated.

The tool is implemented in PROLOG as a module (of the sort described in APPENDIX-A). Each fact is represented as two assertions of form:

```
     fact(<Factid>,<Subject>,<Relation>,<Object>)
and, fact(<Otherid>,<Factid>,in_dbase,<Dbase-id>).
```

The following operations (with signatures) may be used to manage the FACT data type:

```
next_fact(<Factid>)
new_facts
checking(<Status>)
check_fact(<Dbaseid>,<Subject>,<Relation>,<Object>)
write_fact(<Tabval>,<Factid>,<Subject>,<Relation>,<Object>)
fetch_fact(<Code>,<Dbaseid>,<Protofact>)
insert_fact(<Code>,<Dbaseid>,<Fact>)
delete_fact(<Code>,<Dbaseid>,<Protofact>)
read_facts(<File>)
```

where 'Protofact' is a partially instantiated fact. The PROLOG code for these procedures follows (with suitable commentary).

```
export('tools/multifactdb',[next_fact(1),new_facts(0),check_fact(4),
   write_fact(5),checking(1),fetch_fact(3),insert_fact(3),delete_fact(3)]).

/* 'next_fact(<Factid>)' returns a identifier for the next fact.        */
/* The procedure maintains a counter (of form 'last_facts(<INTEGER>)')  */
/* which it updates so forming an identifier of form 'f(<INTEGER>)' */

next_fact(f(Factid)):- retract( last_fact(N) ),
                      Factid is N+1,
                      asserta( last_fact(Factid) ).

/* 'new_facts' removes all previous facts and restarts the fact counter */

new_facts:-
        retract( fact(_,_,_,_) ),
        retract( last_fact(M)),
        fail.

new_facts:- asserta(last_fact(0)).

/* 'check_fact(<Dbaseid>,<Subject>,<Relation>,<Object>)'        */
/* checks that each fact triple is consistent with the semantic rule    */
/* in which the relation is defined (with the exception that set        */
/* membership relations and database labels are assumed to be valid).   */


/* case, checking has been switched off                                  */
check_fact(_,_,_,_):- nocheck, !.
/* cases where checking excluded                                         */
check_fact(_,Member,is_a,Set):-!.
check_fact(_,Factid,in_dbase,Dbase):-!.

/* normal case, 'Subject' and 'Object' must be elements of sets         */
/* used in a semantic rule. The object of a unary relation is not       */
/* checked (eg. 'fact(N,rain,wet,true').                                 */

check_fact(Dbaseid,Subject,Relation,Object):-
        fetch_fact(rule,_,[_,Sset,Relation,Oset]),
        fetch_fact(_,Dbaseid,[_,Subject,is_a,Sset]),
        (Oset=true; fetch_fact(_,global,[_,Object,is_a,Oset])).

/* 'checking(<Status>)' uses flag 'nocheck' to switch checking on/off  */

checking(on):-retract( nocheck ).
checking(on).
checking(off):-nocheck.
checking(off):-asserta( nocheck ).

/* 'fetch_fact(<Code>,<Dbaseid>,[<Factid>,<Subject>,<Relation>,<Object>])' */
/* retrieves a fact from the MULTIFACTS database. Facts may be  */
/* represented explicitly (in which case retrieval is associative)) or  */
/* implicit (in which case it is deductive) using set properties        */
/* case, fetch a semantic rule                                      */
fetch_fact(Code,_,[Id,Subject,Relation,Object]):-
        Code == rule,
        fact(Id,Subject,Relation,Object),
        fact(_,Id,is_a,semantic_rule).

/* case, efficient retrieval of explicit facts where there          */
/* there is a partial specification of fact details                 */

fetch_fact(explicit,Dbaseid,[Factid,Subject,Relation,Object]):-
        nonvar(Relation),
        check_fact(Dbaseid,Subject,Relation,Object),
        fact(Factid,Subject,Relation,Object),
        fact(_,Factid,in_dbase,Dbaseid).

/* case, efficient retrieval of explicit facts where there          */
/* is no partial specification of fact details                      */

fetch_fact(explicit,Dbaseid,[Factid,Subject,Relation,Object]):-
        var(Relation),
        /* pointless call 'check_fact' */
        fact(_,Factid,in_dbase,Dbaseid),
        fact(Factid,Subject,Relation,Object).

/* case, set member relation implicitly defined by a built-in       */
/* predicate. For example, 'fact(..4,is_a,integer)' becomes 'integer(4)' */

fetch_fact(implicit,_,[_,Elem,is_a,Settest]):-
        atom(Settest), Pred =.. [Settest,Elem], call(Pred).

/* case, deductive retrieval of set/superset relations              */

fetch_fact(implicit,Dbaseid,[_,Set,is_a,Superset]):-
        fetch_fact(explicit,Dbaseid,[_,Set,is_a,Parentset]),
        fetch_fact(_,global,[_,Parentset,is_a,Superset]).

/* cases, deductive retrieval by propagation of set properties      */
/* case, subject is related to object which is a set in some        */
/* deductive rule                                                   */

fetch_fact(implicit,Dbaseid,[_,Subject,Relation,Object]):-
        fetch_fact(rule,_,[S,_,Relation,_]),
        fetch_fact(explicit,global,[D,Set,Relation,Object]),
        /* S is semantic & D is deductive rule identifiers */
        fetch_fact(_,Dbaseid,[_,Subject,is_a,Set]).

/* case, subject set is related to elements of object set in        */
/* a deductive rule                                                 */

fetch_fact(implicit,global,[_,Subject,Relation,Object]):-
        fetch_fact(rule,_,[S,_,Relation,_]),
        fetch_fact(explicit,global,[D,Subject,Relation,Set]),
        fetch_fact(_,global,[_,Object,is_a,Set]).
```

```
/* case, element of subject set is related to element of object */
/* set by a deductive rule                                      */

fetch_fact(implicit,Dbaseid,[_,Subject,Relation,Object]):-
        fetch_fact(rule,_,[R,_,Relation,_]),
        fetch_fact(explicit,global,[D,Set1,Relation,Set2]),
        fetch_fact(_,Dbaseid,[_,Subject,is_a,Set1]),
        fetch_fact(_,global,[_,Object,is_a,Set2]).

/* 'insert_fact(<Code>,<Dbaseid>,[<Factid>,<Subject>,<Relation>,<Object>])'*/
/* adds facts to the database if they are not already there.        */

/* case, fact is a semantic rule indicated by Code 'rule'      */

insert_fact(Code,_,[N,Subject,Relation,Object]):-
        Code == rule, !,
        next_fact(N), asserta( fact(N,Subject,Relation,Object) ),
        next_fact(M), asserta( fact(M,N,is_a,semantic_rule) ).

/* case, fact already explicit or deductible in some database          */

insert_fact(Code,Dbaseid,Fact):-
        fetch_fact(Code,Dbaseid,Fact),
        !.

/* case, fact explicit in another database so label also in this database*/

insert_fact(explicit,Dbaseid,[Factid,Subject,Relation,Object]):-
        fetch_fact(explicit,Otherdb,[Factid,Subject,Relation,Object]),
        Dbaseid = Otherdb, !, next_fact(F),
        asserta( fact(F,Factid,in_dbase,Dbaseid) ).

/* case, check for validity then insert a new fact and its database label */

insert_fact(explicit,Dbaseid,[Id,Subject,Relation,Object]):-
        check_fact(Dbaseid,Subject,Relation,Object),
        next_fact(Id), asserta( fact(Id,Subject,Relation,Object) ),
        next_fact(Id2), asserta( fact(Id2,Id,in_dbase,Dbaseid) ).

/* 'delete_fact(<Code>,<Dbaseid>,[<Factid>,<Subject>,<Relation>,<Object>])'*/
/* removes facts from the MULTIFACTS database                      */

/* case, remove a semantic rule (signified by value 'rule' in <Code>'*/

delete_fact(Code,_,[Factid,Subject,Relation,Object]):-
        Code == rule, !,
        retract( fact(Factid,Subject,Relation,Object) ),
        retract( fact(_,Factid,is_a,semantic_rule) ).

/* case, explicit fact belongs to another context so just remove        */
/* database label                                                       */
```

```
delete_fact(explicit,Dbaseid,[Id,Subject,Relation,Object]):-
        fetch_fact(explicit,Dbaseid,[Id,Subject,Relation,Object]),
        fact(_,Id,in_dbase,Otherdb),
        Dbaseid = Otherdb, !,
        retract( fact(_,Id,in_dbase,Dbaseid) ).

/* case, delete explicit fact and associated database label     */

delete_fact(explicit,Dbaseid,[Id,Subject,Relation,Object]):-
        fetch_fact(explicit,Dbaseid,[Id,Subject,Relation,Object]),
        retract( fact(Id,Subject,Relation,Object) ),
        retract( fact(_,Id,in_dbase,Dbaseid) ).


/* 'write_fact(Tabval,Factid,Subject,Relation,Object)'  */
/* simply writes out the contents of a fact. Note that  */
/* if the 'Object' is an abstract syntax tree then the  */
/* appropriate pretty-printer is called (with goal 'pp')        */

/* case, check value is an program                      */

write_fact(Tabval,Id,Sub,Rel,Obj):-
        fetch_fact(rule,_,[_,_,Rel,is_ast]), !, tab(Tabval),
        write('fact('),write(Id),write(','),write(Sub),write(','),
        write(Rel),write(','), pp(_,Obj,Tabval),write('   )'),nl,!.

/* case, value is unstructured object          */

write_fact(Tabval,Id,Sub,Rel,Obj):-
        tab(Tabval), write(fact(Id,Sub,Rel,Obj)), nl,!.
```

APPENDIX-C. HOPE ABSTRACT SYNTAX DEFINITION.

An abstract syntax is defined for the HOPE language below in a formalism very similar to IDL. The syntax can be understood by studying the AST given in APPENDIX-F for the stack module of figure 2.4 (page 37). Note that I use terser identifiers for sorts than those described in the figures of chapter 3 (pages 57 and [illegible]). Here I distinguish sorts of expressions which will be found on the left- and right-hand-sides of HOPE equations with identifiers 'cl_fnexpr' and 'cl_rhsexpr', respectively (not simply 'sequence_of cl_expression'). Recall also that I use the '^' symbol for disjunction in grammar rules defining classes.

```
module => as_modhead : mod_id,
     as_decls : declarations.

declarations => as_list : seq_of cl_decls.

cl_decls ::= datatypes ^ imports ^ typevars ^ export_ops ^
          signature ^ syntax ^ constructors ^ operations ^ eqn.

datatypes => as_list : seq_of adt_id.

imports => as_list : seq_of adt_id.

typevars => as_list : seq_of typevar.

export_ops => as_list : seq_of oper_id.

syntax => as_fix : cl_fix,
      as_list : seq_of oper_id,
      prec : integer.
```

```
cl_fix ::= infix ^ postfix ^ prefix.

constructors => as_list : seq_of signature.

/* following is a dummy 'sort' to help prettyprint of 'data' statement*/

in_types => as_list : seq_of cl_type.

operations => as_list : seq_of signature.

signature => as_op : oper_id,
          as_typexpr : typexp.

typexp => as_list : seq_of cl_type,
       typout : cl_type.

cl_type ::= typeconstr ^ cl_primitive ^ typevar ^ nil.

typeconstr => as_adt : adt_id,
           as_list : seq_of cl_type.

cl_primitive ::= num ^ char ^ truval.

eqn => as_lhs : cl_fnexpr,
     as_rhs : cl_rhsexpr.

cl_rhsexpr ::= cl_fnexpr ^ cond ^ abstract ^ lambda.

cl_fnexpr ::= var_id ^ cl_literal ^ func.

func => as_id : oper_id,
     as_params : params.

params => as_list : seq_of cl_fnexpr.
```

```
cl_literal ::= num_lit ^ char_lit ^ bool_lit.

cond => as_pair : pair,
     as_else : cl_rhsexpr.

pair => as_if : cl_fnexpr,
     as_then : cl_rhsexpr.

abstract => as_in : cl_fnexpr,
         as_abstact : cl_rhsexpr,
         as_is : cl_rhsexpr.

num => void.
char => void.
truval => void.

mod_id => lx_symrep : symbol_rep.
var_id => lx_symrep : symbol_rep.

oper_id => lx_symrep : symbol_rep.

adt_id => lx_symrep : symbol_rep.

typevar => lx_symrep : symbol_rep.

char_lit => lx_symrep : symbol_rep.

bool_lit => lx_symrep : symbol_rep.

num_lit => lx_numrep : number_rep.
```

APPENDIX-D. Construction of Language Parsers for use with ENCORES.

The construction of language parsers in PROLOG is briefly described in this appendix, particularly the lexical analysis phase which is not described adequately in the PROLOG literature. Some examples of code from the HOPE parser of of a tool built with ENCORES is given.

Two papers describe the construction of compilers of programming languages implemented in PROLOG. The paper by Warren [Warren80] describes how the syntactic-analysis and code-generation phases of compilation can be coded. The paper by Colmerauer [Colmerauer78] gives a complete but very terse treatment of compiler writing using a unfamiliar dialect of PROLOG. Indeed, the treatment is so terse that code for a compiler of a PASCAL type language is presented in two A4 pages !! The lexical- and syntactic-analysis phases of Colmerauer's compiler were translated and adapted to produce parse trees of HOPE using the abstract syntax of APPENDIX-C.

The language parser must serve two purposes. It must parse specifications/programs into the internal tree representation and must also parse metalanguage expressions to tree templates. This latter function complicates the syntactic analysis phase as we shall see.

Users normally prepare (syntactically correct) specifications on file as the parser does not perform type checking nor does it interface to an editor. The first section of code sets the standard input of PROLOG to this file,

```
        get_spec:- ask_infile(F),
                seeing(Me), see(F), inputspec,
                asserta(inputok), seen, see(Me), !.
        ask_infile(F):-nl, write('inputfile:'), read(F).
```

PROLOG parsers are large programs and are overlayed, so we must assert the atom 'inputok' if the parse is successful.

The parser builds an abstract syntax tree (AST) of the specification in three phases: first all the source characters are assembled in a list; this is then transformed into a list of tokens by the lexical analyser ; finally, an AST is constructed from these tokens. Procedure 'inputspec' calls procedures for these three phases,

```
        inputspec:- nl, put("$"),
                readin(Module),
                lexanal(Module,Tokenlist),
                parse(Tree,Tokenlist,_,Varinfo).
```

Procedure 'readin' gathers characters of the source program but discards extra spaces and carriage returns. Punctuation symbols are given token names. Integer tokens and identifiers are enclosed by functors 'int' and 'id' respectively to increase the efficiency of syntactic analysis. The code is,

```
        readin([L|U]):-!,
                getnonsp(K), puncn([K],L), readrest(L,U).
        getnonsp(X):-get0(X), X=32, X=10. /*space, line feed*/
        getnonsp(X):-getnonsp(X).
        puncn("-",minu):-!.
        puncn(":",col):-!.
        puncn(".",dot):-!.
        puncn(";",semi):-
        puncn(.....other punctuation...
        readrest(eof,[]):-!. /* cntl-z is end of input */
        readrest(blank,U):-!, readin(U).
        readrest(F,[M|U]):-get0(L), puncn([L],M), readrest(M,U).
```

The lexical analyser gathers characters into tokens of the language which are then represented as PROLOG atoms. Remaining spaces are discarded. The code is,

```
    lexanal(Module,Toklist):-tokens(Toklist,_,Module).
    tokens([U|X],V3,V0):-token(U,V1,V0),!,space(V2,V1),tokens(X,V3,V2).
    tokens([],V0,V0).
    space(V0,[blank|V0]).
    space(V0,V0).
```

A token may be a string of digits which are converted into an integer. Alternatively, a token may an alphanumeric which is either an identifier or a reserved word of the language. (The following code uses reserved words of HOPE). A PROLOG variable in the input stream (from a metalanguage expression) is returned as term 'var(<VARIABLE>)' where VARIABLE is the identifier with initial letter lower case (eg. 'X' becomes 'var(x)').

```
    token(int(X),V1,[K|Rest]):- digit(K), !,
         digits(U,V1,Rest), todecimal([K|U],X).
    token(var(Y),V0,[K|Rest]):-letter(_,K,Kok),!,
         alphanums(U,V0,Rest),name(Y,[Kok|U]).
    token(Y,V0,[K|Rest]):-letter(_,K,Kok),!,alphanums(U,V0,Rest),
         X=[Kok|U], checktok(X,Y).
    token(K,V0,[K|V0]).
    digits([K|U],V0,[K|Rest]):-digit(K),digits(U,V0,Rest).
    digits([],V0,V0).
    digits(K):-integer(K), K>47, K<58.
    /*'todecimal' returns an integer given a list of digits*/
    todecimal([Ones],X):-X is Ones -48, !.
```

```
todecimal([H|T],X):-N is H - 48, todecimal(T,W),Tens is N * 10,
     X is Tens + W.
alphanums([Kok|U],VO,[K|Rest]):-alphanum(K,Kok),alphanums(U,VO,Rest).
alphanums([],VO,VO).
alphanum(K,Kok):-letter(_,K,Kok), !,
alphanum(K,K):-digit(K).
checktok(String,Term):-reserved(String,Term), !.
checktok(String,id(Atom)):-name(Atom,String).
reserved("module",mod):-!.
reserved("data",data):-!.
reserved("dec",dec):-!.
..etc..
/*all letters are converted to lower case*/
letter(low,K,K):-integer(K),K>96,K<123,!./*a,b,c..z*/
letter(cap,K,K1):-integer(K),K1 is K+32. /*A,B,C,..Z*/
```

Syntactic analysis converts the token list into an AST. Further, metavariables (recorded within 'var' terms) give rise to a list of triples : each triple recording the user supplied identifier, the PROLOG identifier substituted and the matching sort of the variable (determined by the parse). ASTs are represented as nested PROLOG terms, with inner terms representing attributes of sorts. Metavariables in the token list give rise to PROLOG variables in the parse tree for matching the appropriate components of programs.

A procedure should exist for each sort of the abstract syntax in the syntactic analyser. Clauses of the procedure must cover all possible values of the sort, including a user supplied metavariable.

Each procedure builds an AST in the first parameter given the tokenlist in the second parameter. Remaining tokens are returned through the third parameter and metavariable information through the fourth. For example, given the HOPE type expression

'stack(alpha) -> stack(alpha)'

it will be parsed to AST (a component of the tree in figure 3.5,page 52),

```
typexp([typeconstr(adt_id(stack),[typevar(alpha)])],
        typeconstr(adt_id(stack),[typevar(alpha)]) )
```

given the following code,

```
/* typexp => as_list : seq_of cl_type,            */
/*        typout  : cl_type.                      */

parse(typexp(I,S),Tokens,V0,Vartypes):-
    seq_of(cl_type,I,Tokens,[minu,rarr|V1],Vartypes1),
    class(cl_type,S,V1,V0,Vartypes2),
    append(Vartypes1,Vartypes2,Vartypes).

/* cl_type ::= typeconstr ^ cl_primitive ^ typevar ^ nil */

class(cl_type,X,[var(V)|V0],V0,[[X,cl_type,var(V)]]).

class(cl_type,Type,Toks,V0,Vt):-
    (Type=typevar(I); Type=typeconstr(I,0)),
    parse(Type,Toks,V0,Vt).

class(cl_type,I,Toks,V0,Vt):-
    class(cl_primitive,I,Toks,V0,Vt).

parse(typevar(T),[id(T)|V0],V0,[]):-
```

```
        typevars(L),
        member(typevar(T),L).

/* cl_primitive ::= num ^ char ^ truval.          */

class(cl_primitive,num,[num|V0],V0,[]).
class(cl_primitive,char,[char|V0],V0,[]).
class(cl_primitive,truval,[truval|V0],V0,[]).

/* typeconstr => as_constr : adt_id , as_list : seq_of cl_type   */

parse(typeconstr(adt_id(list),P),[list,lbrac|Rest],V0,Vt):-
        class(cl_type,P,Rest,V1,Vt),
        V1 = [rbrac|V0].

parse(typeconstr(adt_id(I),P),Tokens,V0,Vt):-
        parse(adt_id(I),Tokens,Rest,Vt1),
        seq_of(cl_type,P,Rest,V0,Vt2),
        append(Vt1,Vt2,Vt).

parse(typevar(I),[id(I)|V0],V0,[]).
parse(typevar(I),[var(V)|V0],V0,[[I,typevar,var(V)]]).
parse(adt_id(I),[id(I)|V0],V0,[]).
parse(adt_id(I),[var(V)|V0],V0,[[I,typevar,var(V)]]).
```

As such parsers are large PPOLOG programs they may be best overlayed. The tool built for our investigations of automatic programming has a parser overlayed in three sections : the code of the lexical analyser, syntactic analysis of signatures and analysis of axioms are held in three separate modules. Structured languages, such as

PASCAL, will require larger parsers with code possibly divided into a tree of modules.

APPENDIX-E. PRETTYC : A Tool to Compile Pretty-printers.

The PRETTYC tool compiles a pretty-printer in PROLOG code given a definition of the abstract syntax of the language in one file and a set of pretty=print rules in another file. The abstract syntax should be defined by statements like those of APPENDIX-C, which is read by the tool as a set of assertions. A notation has been developed for the specification of pretty-print rules. The following are a matching set of print rules in this notation for the abstract syntax defined APPENDIX-C. Again, these statements are treated as PROLOG assertions.

```
module --> >>"module " , as_modhead, ";",
          nl, as_decls,";",nl, >>"end;",nl.
declarations --> as_list : delimit([59,10]).
datatypes --> >>"pubtype " , as_list : delimit(",").
imports --> >>"uses", as_list : delimit(",").
typevars --> >>"typevar ", as_list : delimit(",").
export_ops --> >>"pubconst " , as_list : delimit(",").
syntax --> write(as_fix), as_list : delimit(","),":", write(prec).
constructors --> >>"data ", as_list=[[signature(O1,typexp(Tin,To)),
                      signature(O2,typexp(Tin2,To2))]],
          cl_type(To), " == ", oper_id(O1), in_types(in_types(Tin)),
          " ++ ", oper_id(O2), in_types(in_types(Tin2)).
/* print rules for dummy sort 'in_types' - helps print of 'data' statement*/
in_types --> as_list = [nil].
in_types --> "(", as_list : delimit("#"), ")".
operations --> as_list : delimit([59,10]). /* ';<CR>' */
signature --> >>"dec ", as_op, " : ", as_typexpr.
```

```
typexp --> as_list=[nil], "  -> " , typout.

typexp -->  as_list : delimit("#"),  " -> ", typout.

typeconstr --> as_list=[nil], as_adt.

typeconstr --> as_adt, "(", as_list : delimit(","), ")".

num --> "num".

char -->"char".

truval -->"truval".

eqn --> >>"---- ", as_lhs, " <= ", as_rhs, ";".

params --> as_list=[nil].

params --> "(", as_list : delimit(","), ")".

func --> as_id=[oper_id(plus)], as_params=[params([A,B])],
      "(", cl_fnexpr(A), "+", cl_fnexpr(B), ")".

func --> as_id, as_params.

num_lit --> write(lx_numrep).

char_lit --> write(lx_symrep).

bool_lit --> write(lx_symrep).

cond --> as_else=[void], as_pair.

cond -->  as_pair, " else ", as_else.

pair --> "if ", as_if, " then " , as_then.

oper_id --> write(lx_symrep).

mod_id --> write(lx_symrep).

adt_id --> write(lx_symrep).

typevar --> write(lx_symrep).

var_id --> write(lx_symrep).
```

The notation can be understood as follows:

1. Each rule is a statement of how the tree with node matching the

sort on the left-hand-side of the '—>' symbol should be printed.

2. The node is printed in accordance with instructions for printing relevant attributes found on the right-hand-side of the rule. If one or more attributes are not mentioned on the right-hand-side do not contribute to regeneration of source code.
3. Strings between double-quotes are simply printed (eg. "if").
4. An 'indentation index' is maintained by the pretty-printer when printing.
5. The ">>" symbol before a string or attribute results in a number of spaces (that of the index value) being output before the string or value.
6. ">>N", where 'N' is some integer results in the indentation index being increased by 'N'. Conversely, "<<N" results in the index being decreased by 'N'.
7. An attribute name on the RHS means the value of the attribute should be pretty-printed.
8. Instructions of the form 'write(<ATTRIBUTE NAME>)' result in the value of the attribute being printed. That is, the attribute is some terminal value of the tree.
9. Some delimiter may be specified to seperate the output of attributes which consist of a sequence of values. For example, 'as_list' : delimit(",")'.
10. Conditions may be placed on the value of an attribute before the print rule will be obeyed (eg. 'as_list=[nil]'). Conseouently, there may be more than one

instructions to direct them to the relevant print rule.

Instructions have the form <SORT>(<COMPONENT>).

For example, "as_list=[[signature(Ol,typexp(Tin,To))]],

out_type(To), "==", oper(Ol), in_types(Tin)".

The statements describing the abstract grammar and the pretty-print rules are supplied to the PRETTYC tool for the compilation of the pretty-printer. Suppose the syntax statements of Appendix-C are held in file named 'syntax, and the print rules above are held in file 'prules'. Then the pretty-printer is simply compiled by 'consulting' the PRETTYC tool into the PROLOG system then entering the query 'prettyc.'. The dialogue proceeds as follows (user replies are underlined):

```
Welcome to the pretty-printer compiler
Name of Language : hope.
Files maintained in directory  hope/
Enter pretty-print rules. Name of File ? prules.
Enter syntax rules. Name of File ? syntax.
Compiling pretty-printer....
New pretty-printer for hope language written on file hope/pretty
Bye.
```

APPENDIX-F. COMPLETE ABSTRACT SYNTAX TREE OF STACK MODULE.

A complete AST for the stack module is given below. Node names are based on the abstract syntax given in APPENDIX-A and the tree was constructed using the HOPE parser described in APPENDIX-D. This AST is in the internal representation used by the system but has been pretty-printed for clarity.

```
module(
    mod_id(anystack),
    declarations([
            datatypes([
                    adt_id(stack)]
                     ),
            export_ops([
                    oper_id(pop),
                    oper_id(top),
                    oper_id(empty),
                    oper_id(push),
                    oper_id(newstack)]
                      ),
            typevars([
                    typevar(alpha)]
                    ),
            constructors([
                    signature(
                        oper_id(newstack),
                        typexp(
                                [nil],
                                [typeconstr(
                                    adt_id(stack),
                                            [typevar(
                                 alpha)
                                                   ]
                                          )
                                    ]
                             )
                           ),
                    signature(
                        oper_id(push),
                        typexp(
                              [typeconstr(
                                        adt_id(stack),
                                        [typevar(
                                                   alpha)
                                                      ]
                                               ),
                          typevar(
                                  alpha
                                     )],
                          typeconstr(
                               adt_id(stack),
                               [
                                      typevar(
                                          alpha)
                                             ]
                                      )
                               )
                         )
                        ]
                    ),
    operations([
           signature(
               oper_id(pop),
               typexp(
                   [
                       typeconstr(
                               adt_id(stack),
                               [
                                      typevar(
                                          alpha)
                                             ]
                                      )
                                    )],

                       typeconstr(
                           adt_id(stack),

                               nil
                                   )
                                 )
                          )
                    )
                  )]
             ),
  operations([
         signature(
             oper_id(top),
             typexp(
                 [
                     typeconstr(
                             adt_id(stack),
                             [
                                    typevar(
                                        alpha)
                                           ]
                                    )
                                  )],

                     typevar(
                         alpha)
```

```
                              )
                           )
                     )
                    )]
             ),
  operations([
          signature(
              oper_id(empty),
              typexp(
                  [
                     typeconstr(
                              adt_id(stack),
                              [
                                      typevar(
                                          alpha)
                                             )],
                                      )
                                    )],

                      truval
                          )
                    )
                   )]
            ),
 eqn(
     func(
         oper_id(pop),
         params([
                 func(
                    oper_id(newstack),
                     params(
                         nil
                           )
                     )]
               )
        ),
         var_id(undef)
    ),
 eqn(
     func(
         oper_id(pop),
         params([
                 func(
                    oper_id(push),
                     params([
                             var_id(s),
                             var_id(item)]
                           )
                    )]
               )
        ),
         var_id(s)
    ),
 eqn(
          func(
              oper_id(top),
              params([
                          func(
                              oper_id(newstack),
                               params(
                                   nil
                                     )
                               )]
                        )
               ),
                var_id(undef)
        ),
    eqn(
          func(
              oper_id(top),
              params([
                          func(
                              oper_id(push),
                               params([
                                           var_id(s),
                                           var_id(item)]
                                     )
                               )]
                        )
               ),
                var_id(item)
        ),
    eqn(
          func(
              oper_id(empty),
              params([
                          func(
                              oper_id(newstack),
                               params(
                                   nil
                                     )
                               )]
                        )
               ),
                bool_lit(true)
        ),
    eqn(
          func(
              oper_id(empty),
              params([
                          func(
                              oper_id(push),
                               params([
                                           var_id(s),
                                           var_id(item)]
                                     )
                               )]
                        )
        ),
         bool_lit(false)
    )]
     )
```

APPENDIX-G. APPEAL Description of Prototype Knowledge Base.

This appendix collects the rules accumulated during our earlier work with SPECTRE [McArthur85] but described in the APPEAL language. The reader is reminded that our approach to the Alvey research project (see footnote, page 19) is being reappraised. Thus the knowledge base described here is no more than a 'frozen' prototype, is incomplete and requires further test on specifications of 'linear' objects (such as stacks, queues etc.) for which it was designed.

The rules are designed to analyse operations then choose some representing type. Specification rules (like those of figure 2.5, page 40) will be added manually to the blackboard before being transformed to the equations of the implementation (by the process described in page 64).

Many of the following APPEAL rules have been described in earlier chapters. Some rules, such as the find-deletes-operation rule, will be found in [Bartel etal81]. All the control rules are to found in sections 4.4.2 (page 83) of the thesis so are not duplicated here. Further acquisition and evaluation of control knowledge will not take place until a larger domain knowledge base is in place. Where commentary is necessary to explain the behaviour of rules, we enclose this within '/*.........*/'.

```
/*  Rule name is 'find-reads-operation'.                         */
/* This rule differs a little to that shown in figure 3.15a, page 65.*/
/* Extra preconditions are added to check that the action of the    */
/* operation has not been established (to increase efficiency) and  */
/* check that the operation is associated with the data type to be  */
/* implemented (for cases where information on more than one data   */
/* type is recorded on the blackboard).                              */

        if  a datatype(1) has
         &    type = 'D(P)'
         &  an operation(1) has
         &    action = unknown
         &    signature = 'dec Op : Insorts -> P;'
         &    Insorts = '__D(P)__'
      then     operation(1) has
         &    action = reads


/* Rule is find-nullary-constr                              */

        if  a datatype(1) has
         &    type = 'D(P)'
         &  an operation(1) has
         &    signature = 'dec Op : nil -> D(P);'
      then     operation(1) has
         &    action = creates



/* Rule name is find-constructor-operation                          */

        if  a datatype(1) has
         &    type = 'D(P)'
         &  an operation(1) has
         &    action = unknown
         &    signature = 'dec Op2 : Insorts1 -> D(P);'
         &  an operation(2) has
         &    action = reads
         &    signature = 'dec Op : Insorts2 -> Out2;'
         &  an equation(1) has
         &    axiom = '---Op(Patt) <= R;'
         &    Patt = '__Op2(Params)__'
      then     operation(1) has
         &    action = constructs
         &    actposn = front
```

```
/* Rule name is find-writes-operations.                   */
/* Used to distinguish primitive constructor operations */
/* from other 'writes' operations by presence of axioms */
/* defining these latter operations.                      */

         if  an operation(1) has
          &   signature = 'dec Op : Insorts -> D(P);'
          &   action = constructs
          &  an equation(1) has
          &   axiom = '---Op(Lhs) <= Rhs;'
       then     operation(1) has
          &     action = writes


/* Rule name is find-deletes-operation                    */

         if  a datatype(1) has
          &   type = 'D(P)'
          &  an operation(1) has
          &   action = constructs
          &   signature = 'dec Op1 : In -> D(P);'
          &  an operation(2) has
          &   action = unknown
          &   signature = 'dec Op2 : In2 -> D(P);'
          &  an equation(1) has
          &   axiom = '---Op2(Op1(Parms)) <= Rhs;'
          &   Parms = '__Rhs__'
       then     operation(2) has
          &   action = deletes



/* Rule name is find-tests-operation                      */

         if an operation(1) has
          &  action = unknown
          &  signature = 'dec Op : Insorts -> truval;'
       then    operation(1) has
          &    action = tests
```

```
/* Rule name is act-front-operations                    */
/* Absence of recursion indicates that the operation    */
/* takes its effect at the front of the abstract object */

        if an operation(1) has
         &  actposn = unknown
         &  signature = 'dec Op : In -> Out;'
         & an equation(1) has
         &  axiom = '___ Op(Patt) <= Rhs;'
         &  not( Rhs = '__Op(Parms)__')
      then    operation(1) has
         &  actposn = front


/* Rule name is choose-sl-list                          */
/* If all operations of object act at the front of the  */
/* object then represent with singly linked list.       */
/* Note obvious precondition to increase efficiency     */
/* over the simple form of figure 3.15b, page 65.       */

        if  a datatype(1) has
         &   type = 'D(P)'
         &   represent = 'unknown(P)'
         &  all operation(s) have
         &   signature = '__D(P)__'
         &   actposn = front
      then      datatype(1) has
         &      represent = 'sl_list(P)'


/* Rule name is choose-representation                   */
/* More general form of above rule.                     */
/* Choose representation based on similarity of behaviour */
/* of its operations to those of specified object.      */

        if a datatype(1) has
         &   type = 'D(P)'
         &   represent = 'unknown(P)'
         & all operation(s) have
         &   actposn = End
         &   signature = '__D(P)__'
         & a datatype(2) has
         &   type = 'C(Pc)'
         & all operation(s) have
         &   actposn = End
         &   signature = '__C(Pc)__'
      then    datatype(1) has
         &   represent = 'C(P)'
```

```
/* Some rules for permuting expressions  */

/* Commuting terms being added or multiplied    */

        if  an equation(1) has
         &   code = '__A+B__'
      then  an equation(1) has
         &   code = '__B+A__'

        if  an equation(1) has
         &   code = '__A*B__'
      then  an equation(1) has
         &   code = '__B*A__'

/* Associating terms differently                */

        if  an equation(1) has
         &   code = '__A+(B+C)__'
      then  an equation(1) has
         &   code = '__(A+B)+C__'

        if  an equation(1) has
         &   code = '__A*(B*C)__'
      then  an equation(1) has
         &   code = '__(A*B)*C__'

/* Distributing multiplication over addition */

        if  an equation(1) has
         &   code = '__X*(Y+Z)__'
      then  an equation(1) has
         &   code = '__X*Y + X*Z__'
```

```
/* Rule name is fold                                  */

        if   an equation(1) has
         &    code = '---Lhs1 <= Expr;'
         &   an equation(2) has
         &    code = '---Lhs2 <= Rhs;'
         &    Rhs = '__Expr__'
       then     equation(2) has
         &    Rhs = '__Lhs1__'
         &    code = '---Lhs2 <= Rhs;'


/* Rule name is unfold                                */

        if   an equation(1) has
         &    code = '---Lhs1 <= Rhs1;'
         &    Rhs1 = '__Op(X)__'
         &   an equation(2) has
         &    code = '---Op(X) <= Rhs2;'
       then     equation(1) has
         &    Rhs1 = '__Rhs2__'
         &    code = '---Lhs1<= Rhs1;'


/* Rule name is instantiate1 (for single parameters)  */

        if   an equation(1) has
         &    code = '---Op1(X)<= Rhs1;'
         &    all( Rhs1 = '__X__')
         &   an equation(2) has
         &    code = '---Op2(Y)<= Rhs2;'
         &    not ( Y = X)
       then     equation(1) has
         &    all( Rhs1 = '__Y__')
         &    code = '---Op1(Y)<= Rhs1;'
```

REFERENCES

Adelson,B.,Soloway,E. 1985.
The Role of Domain Experience in Software Design,
IEEE Trans. on S.E., Vol.SE-11, Nov.1985, pp.1351-1360.

Albrecht,P.F.,Garrison,P.E.,Graham,S.L.,Hyerle,R.H.,Ip,P.,
Krieg-Bruckner,B. 1980.
Source-to-source translation : ADA to PASCAL and PASCAL to ADA.
ACM Sigplan Symp. on ADA programming language.

Balzer, R. 1973
A global view of automatic programming
Proc. of 3rd IJCAI, pp494-499.

Balzer,B.,Golman,N.,Wile,D. 1975.
Informality in program specifications.
IJCAI-4, pp389-397.

Balzer,R.,Goldman,N. 1979.
Principles of good software specification and their implication
for specification languages.
Proc.Conf.Specifications Reliable Software, Boston,MA., pp58.

Balzer,R. 1981a
Transformational Implementation : an example.
IEEE Transactions on Software Engineering, Vol.SE-7, No.1, pp3-14.

Balzer, R. 1981b
Final report on GIST
ISI/Univ. Southern Calif.

Barstow,D.R.,Aiello,N.,Duda,R.O.,Erman,L.D. 1983.
Languages and Tools for Knowledge Engineering,
Chapter 9 in [Hayes-Roth et al. 1983].

Barstow, D.R. 1985
Domain-specific Automatic Programming,
IEEE Trans. Soft.Eng., Vol.SE-11, No.11.

Bartel,U.,Olthoff,W.,Raulefs,P. 1981.
APE : an expert system for automatic programming from abstract
specifications of data types and algorithms.
German Workshop on AI proceedings, IFB 47, Springer-Verlag.

Begg, V. 1984
"Developing Expert CAD Systems"
New Technology Modular Series
Kogan Page : London.

Boyle, C.D.B. 1985
Acquisition of Control and Domain Knowledge
by Watching in a Blackboard Environment,
Proc. of Expert Systems '85,
Univ. of Warwick, pp.272-86, Dec. 1985.

Bratko, I. 1986
"PROLOG : Programming for Artificial Intelligence",
International Computer Science Series,
Addison-Wesley.

Budgeon, D. 1985
Some thoughts on a MASCOT design toolset,
Working Paper,
Dept. Computer Science,
Univ. of Stirling.

Bundy,A.,Silver,B.,Plummer,D. 1985
An Analytical Comparison of Some Rule-learning Systems,
AI Journal, Vol.27, pp.137-181.

Burstall, R.M., Darlington, J. 1977
A transformation system for developing recursive programs,
Journ. ACM 24,1 (Jan.),pp44-67

Burstall,R.M.,MacQueen,D.B.,Sanella,D.T. 1980.
HOPE : an experimental applicative language. Report CRS-62-80,
Dept. of Computer Science, University of Edinburgh.

Buxton, J.N., Randell, B. (Eds.) 1970
Proc. of Software Engineering Techniques Conf.
Rome, Italy, Oct. 1969
published by Nato Science Committee,
Brussels 39, April 1970.

Clancey,W.J.,Letsinger,R. 1981
NEOMYCIN : Reconfiguring a Rule-based Expert
System for Application to Teaching,
Proc. of IJCAI-7, pp.829-836.

Clancey,W.J. 1983
The Epistemology of a Rule-based Expert System :
A Framework for Explanation,
AI Journal, Vol.20, pp.215-251.

Clark,K.L.,McCabe,F.G. 1982.
PROLOG : a language for implementing expert systems.
In "Machine Intelligence", Vol.10,(J.E.Hayes, D.Michie, and Y-H.Pao eds.),
pp455-470. Chichester : Ellis Horwood.

Clocksin,W.F.,Mellish,C.S. 1981.
"Programming in PROLOG". Berlin : Springer-Verlag.

Colmerauer.A. 1978.
Metamorphosis Grammars.
In "Natural Language Communication with Computers",
(L.Bolc,ed.), Lecture Notes in Computer Science,
No. 63, Springer-Verlag, pp133-189.

Corkill,D.D.,Lesser,V.R. 1983
The Use of Metalevel Control for Coordination
in a Distributed Problem-Solving Network,
Proc. of IJCAI-8, pp.748-56.

Cottam,I.D.,Jones,C.B.,Nipkow,T.,Wills,A.C., 1985
Wolczko,M.I.,Yaghi,A.
Project Support Environments for Formal Methods.
Chapter 3 in [McDermid85].

Craig,I. 1986
The Ariadne-1 Blackboard System,
The Computer Journal, Vol.29, No.3.

Davis,R.,King,J. 1976
An Overview of Production Systems,
in "Machine Intelligence", Vol.8,
(Eds.Elcock,E.W. and Michie,D.),
New York : John Wiley & Sons, pp.300-32.

Davis,R. 1980a
Meta-rules: Reasoning about Control.
AI Journal, Vol.15, pp.179-222.

Davis,R. 1980b
Content-reference : reasoning about rules,
AI Journal, Vol.15, pp.223-39.

Deshowitz,N.,Lee,Y. 1987.
Deductive debugging
Report UILU-ENG-87-1715,
Knowledge-based Programming Assistant Project,
Univ. of Illinois at Urbana-Champaign.

Dignan,T. 1984
Strategy for Knowledge-based IPSE Development.
Alvey Directorate, Dept. of Trade and Industry.

Duda, R.O., Hart, P.E., Nilsson, N.J. 1976.
Subjective Bayesian methods for rule-based inference systems,
Proc. of AFIPS 1976 Nat. Computer Conf.,
vol. 45, pp.1075-82.

Duda,R.O.,Gaschnig,J.G.,Hart,P.E. 1979.
Model Design in the PROSPECTOR consultation system for mineral
exploration. In D.Michie,ed.,"Expert systems in the micro-electronic
age". Edinburgh : Edinburgh University Press,pp153-167.

Engelmore,R.S.,Nii,H.P. 1977.
A knowledge-based system for the interpretation of protein x-ray
crystallographic data. Heuristic Programming Project Memo HPP-77-2.
Dept. of Computer Science,University of Stanford.

Erman.L.D.,Lesser,V.R. 1975.
A multi-level organisation for problem solving using many, diverse,
cooperating sources of knowledge, Proc. 4th IJCAI, pp483-490.

Erman,L.D.,Hayes-Roth,F.,Lesser,V.,Reddy,D. 1980.
The HEARSAY-II speech-understanding system: Integrating knowledge to
resolve uncertainty. Computing Surveys 12, No.2, pp213-253.

Erman,L.D.,London,P.E.,Fickas,S.F. 1982
The design and an example use of Hearsay-III,
Proc. 7th IJCAI, pp409-415.

Fickas,S. 1980.
Automatic goal-directed program transformation.
AAAI-1980, pp68-70

Forgy, C.L. 1981.
The OPS5 user's manual,
Tech.Rept. CMU-CS-81-135,
Computer Science Dept.,
Carnegie-Mellon University, Pittsburgh.

Gaschnig, J., Klahr, P., Pople, H., Shortliffe, E., Terry, A. 1983.
Evaluation of Expert Systems : Issues and Case Studies.
Chapter 8 of [Hayes-Roth etal83].

Gerhart,S.L.,Yelowitz,L. 1976
Observations of Fallibility in Applications
of Modern Programming Methodologies,
IEEE Trans. on S.E., Vol.SE-2, No.3, pp.195-207.

Goguen,J.A.,Meseguer,J. 1986
Extensions and Foundations of Object-orientated Programming
ACM Sigplan notices, V21, No.10.

Goldberg,A.,Robson 1981
BYTE Magazine,
Special issue on SMALLTALK,
August, 1981.

Goldman,N. 1978.
AP3 user's guide.
ISI, Univ. of Southern California.

Goldstein,I.P.,Bobrow,D.G. 1981.
Layered Networks as a tool for software development.
IJCAI- 7, pp913-919.

Goos,G.,Wulf,W.A. 1983.
"DIANA reference manual", LNCS No.161, Springer-Verlag, 1983.

Green, C. 1969.
Application of theorem proving to problem solving
IJCAI-1, pp.219-239

Greenspan, S.J. 1984
Requirements Modeling : a knowledge representation approach to software requirements definition,
Ph.D thesis,
Dept. of Computer Science,
University of Toronto.

Greenspan,S.,Mylopoulos,J.,Bordiga,A. 1985
Workshop on Specification,
Dept. of Computer Science,
Univ. of Toronto, Canada, M58 1A4.

Hamilton, G. 1985.
Program Construction in Martin-Lof Type Theory
Technical Report 24,
Dept. of Computer Science,
University of Stirling.

Hammond,P. 1980.
Logic programming for expert systems. M.Sc. Thesis, Dept. of Computing. Imperial College, Univ. of London.

Hammond,P. 1983.
APES : a user manual. Report 82/9, Dept. of Computing, Imperial College, Univ. of London.

Hammond,P.,Sergot,M.J. 1983.
A PROLOG shell for logic based programs.
Proc. of Conf. on Expert Systems, British Computer Society,
Churchill College, Univ. of Cambridge.

Hanson, A., Riseman, E. 1987
VISIONS : a computer system for interpreting scenes
In Hanson,A. and Riseman.E.(editor), "Computer Vision Systems",
Academic Press, 1978.

Hasling,D.W.,Clancey,W,J.,Rennels,G. 1984.
Strategic explanations for a diagnostic consultation system.
Int.J.Man-Machine Studies 20, pp3-19.

Hayes-Roth, B. and F. 1979
A Cognitive Model of Planning,
Cognitive Science, 3, pp.275-310.

Hayes-Roth,B.,Hayes-Roth,F.,Rosenschein,S.,Cammarata,S. 1979
Modelling planning as an incremental, opportunistic process.
Proc. of 6th IJCAI, pp375-383.

Hayes-Roth,B. 1983.
The blackboard architecture : a general framework for problem solving?
HPP-83-30, Dept.of Computer Science, University of Stanford.

Hayes-Roth,F.,Waterman,D.A.,Lenat,D.B. 1983.
"Building Expert Systems". Edited by three authors above.
Massachusetts : Addison-Wesley.

Hayes-Roth,B. 1985.
A Blackboard Model of Control.
Artificial Intelligence 26, pp251-321.

Henderson,P. 1980.
"Functional programming : application and implementation".
Prentice-Hall.

Henderson,P. 1984.
ME TOO - a language for software specification and model building - a preliminary report. Internal report FPN-9, Dept. of Computer Science, University of Stirling.

Hendrix, G.G. 1975.
Expanding the utility of semantic networks through partitioning.
IJCAI 4, pp.115-121.

Johnson, C.K., Jordan, S.R. 1983.
Emergency Management of Inland Oil and Hazardous
Chemical Spills : A Case Study in Knowledge Engineering,
Chapter 10 in [Hayes-Roth etal83].

Johnson,W.L.,Soloway,E. 1985
PROUST : Knowledge-Based Program Understanding.
IEEE Trans. on S.E., Vol.SE-11,No.3,pp267-275.

Johnson,W.L. 1986
"Intention-Based Diagnosis of Novice Programming Errors",
Research Notes in Artificial Intelligence,
Pitman : London (and Morgan Kaufmann : California)

Jones,J.,Millington,M.,Ross,P. 1986
A Blackboard Shell in PROLOG,
Proc. of 7th European Conf. on AI,
Brighton, U.K., pp.428-36.

Kapur,D.,Srivas,M. 1985
A Rewrite Rule Based Approach for Synthesising Abstract
Data Types,
Proc. of TAPSOFT '85, Vol.1., Berlin,
Lecture Notes in Computer Science 185, Springer-Verlag.

Kernighan,B.W.,Pike,R. 1984.
"The Unix Programming Environment", New Jersey : Prentice-Hall.

King, R.M.,Brown, T. 1982
Research on synthesis of concurrent computing systems,
Tech. Rep. KES.U.82.10, Kestrel Inst., Palo Alto, CA.

Kowalski,R. 1979.
"Logic for problem solving". New York : North-Holland/Elsevier.

Lang,B. 1985
MENTOR - Design and Implementation of the Kernal
of a Program Manipulation System,
Chapter 14 in [McDermid85].

Leblang,D.B. 1982.
Abstract syntax based programming environments.
Proc. of ADETEC conference on ADA, ACM. pp187-200.

Lenat,D.B. 1982.
The nature of heuristics.
Artificial Intelligence 19, pp189-249.

Lesser,V.R.,Fennell,R.D.,Erman,L.D.,Reddy,D.R. 1975.
Organisation of the HEARSAY-II speech-understanding system.
IEEE transactions on acoustic, speech and signal processing.
vol. ASSP-23,No.1, pp11-24.

Lindsey,C.H. 1986
Report on meeting of IFIP Working Group 2.1
on Transformation Systems,
Sausilito, California,
(compiled Univ.of Manchester).

Liskov, B.H., Zilles, S.N. 1975
Specification Techniques for Data Abstractions,
IEEE Trans. Soft.Eng., Vol.SE-1, No.1.

McArthur,D. 1985.
SPECTRE - a specification translation and retrieval expert.
Internal report TR.NO.25, Dept. of Computer Science,
University of Stirling.

McArthur,D. 1986.
A system to aid the construction of intelligent software tools.
Technical Report 29, Dept. of Computer Science,
University of Stirling.

McDermid,J. (Ed.) 1985
"Integrated Project Support Environments",
IEE Software Engineering Series 1,
Peter Pengrinus : London.

McDermott,D.V.,Sussman,G.J. 1972
The CONNIVER Reference Manual,
AI Memo No.259, AI Laboratory,
MIT.

McDermot,J. 1982.
R1 : A Rule-based Configurer of Computer Systems.
Artificial Intelligence 19, pp39-88.

McGregor,D.R.,Malone,J.R. 1983
The FACT Database : an Entity-based System using Inference,
in "Entity-Relationship Modelling and Analysis", P.P.Chen (ed.),
Elsevier Science Publishers B.V. (North Holland).

Manna, Z., Waldinger, R. 1980.
A deductive approach to program synthesis
ACM. Trans.Prog.Lang.Syst. 2,1(Jan.),pp90-121.

Minsky,M. 1975.
A framework for representing knowledge. In P.Winston,ed.,
"The Psychology of computer vision". New York : McGraw-Hill.

Naur, P., Randell, B. (Eds.) 1969
Proc. of Software Engineering Conf.
Garmisch, Germany, Oct., 1968,
published by Nato Science Committee,
Brussels 39, Jan. 1969.

Nestor,J.R.,Wulf,W.,Lamb,D.A. 1981.
IDL - Interface Description Language.
Dept. of Computer Science, Carnegie-Mellon Univ.

Newall,A.1969.
Heuristic programming : Ill-structured problems. In A.Aronofsky,ed.,
Progress in operations research,Vol.3.,New York,
John Wiley and Sons,pp360-414.

Nii,H.P.,Aiello,N. 1979.
AGE (Attempt to Generalise) : a knowledge-based program for building
knowledge-based programs. IJCAI 6, pp645-655.

Nii,H.,Feigenbaum,E.,Anton,J.,Rockmore,A. 1982.
Signal-to-symbol transformation : HASP/SIAP case study.
The AI magazine 3(2), pp23-35.

Nilsson,N.J. 1980.
"Principles of Artificial Intelligence". Palo Alto, Calif.:Tioga Press.

Noordzij, J. 1987
Blackboard Translation,
SYSTEMS INTERNATIONAL Magazine, March, 1987.

Parker,J., Hendley, R. 1987
The Re-use of low level programming knowledge in the
UNIVERSE Programming environment,
Proc. Conf. Software Engineering Environments,
Univ. of Keele, April, 1987.

Parnas,D.L. 1985.
Software Aspects of Strategic Defence Systems,
Report DCS-47-IR, Dept. of Computer Science,
University of Victoria, B.C., July.

Parnas, D.L. 1985a.
Why Conventional Software Development does not produce Reliable Programs.
In [Parnas85].

Parnas, D.L. 1985b
Limits of Formal Methods
In [Parnas85]

Parnas, D.L. 1985c.
The Limits of Software Engineering Methods
In [Parnas85].

Partsch, H., Steinbruggen, R. 1983
Program Transformation Systems,
Computing Surveys, Vol.15, No.3.

Polak,W. 1986
AI Approaches to Software Engineering,
Proc. of BCS Conference on Complementary Approaches to the Software Engineering Process,
Imperial College, Univ. of London, March, 1986.

Rich, C., Shrobe, H.E., Waters, R.C. 1979.
An overview of the Programmer's Apprentice
Proc. 6th IJCAI, Tokyo.

Rich,C. 1981.
A formal representation for plans in the Programmer's Apprentice
IJCAI-81, pp1044-1052.

Rich, C. 1985
The layered architecture of a system for reasoning about programs,
Proc. IJCAI-85, pp540-546.

Robinson,J.A. 1965
A Machine-orientated Logic Based on the Resolution Principle,
Journal of ACM, No.12.

Sacerdoti,E.D. 1974.
Planning in a hierarchy of abstraction spaces.
Artificial Intelligence 5, No.2., pp115-135.

Scherlis,W.L.,Scott,D.A. 1983.
First steps towards inferential programming.
Report CMU-CS-83-142, Dept. of Computer Science, Carnegie-Mellon University.

Sergot,M.J. 1983.
A Query-the-User facility for logic programming.
In "Integrated Interactive Computer Systems" (P.Degano and E.Sandewall,eds.).
Amsterdam : North-Holland.

Shortliffe, E.H., Buchanan, B.G. 1975.
A model of inexact reasoning in medicine,
Mathematical Biosciences 23 : pp351-79.

Shortliffe,E.H. 1976.
"Computer-based medical consultation : MYCIN",
New York : American Elsevier.

Siddiqi,J.I.A.,Sumiga,J.H. 1986
Empirical Evaluation of a Proposed Model
of the Program Design Process,
Proc. of Empirical Foundation for Information System Sciences,
Georgia Institute of Technology, Atlanta, Oct. 1986.
(Also from the School of Computing and Information Technology,
The Polytechnic, Wolverhampton, U.K.).

Simon,H.A. 1971.
The theory of problem solving. Information Processing 71, North-Holland,
pp261-277.

Slape,J.K.,Wallis,P.J.L. 1983.
Conversion of FORTRAN to ADA using an intermediate tree representation.
Computer Journal, Vol.26, No.4.

Sleeman,D.H. 1981
Assessing Aspects of Competence in Basic Algebra,
in "Intelligent Tutoring Systems", (Sleeman,D.H. ed.),
Academic Press.

Smith, D.R. 1985
Reasoning by cases and the formation of conditional programs.
Proc. IJCAI, Los Angeles.

Smith,D.R.,Kotik,G.B.,Westfold,S.J. 1985
Research on knowledge-based software environments at Kestrel Institute,
IEEE Trans. Soft. Eng., Vol. SE-11, No.11.

Snowdon,R.A.,Munro,N.C.,Davis,N.W.,Jackson,M.I. 1985
Advanced Support Environments; an industrial viewpoint.
Chapter 15 in [McDermid85].

Sommerville,I. 1982.
"Software Engineering". Massachusetts : Addison-Wesley.

Spohrer, J.C. 1987
Personal communication,
Dept. of Computer Science,
University of Yale.

Stefik,M. 1978.
An examination of a frame-structured representation system.
Report HPP-78-13, Heuristic Programming Project, Dept. of Computer Science,
Stanford University.

Sterling,L. 1984
Logical Levels of Problem Solving,
Logic Programming Journal, Vol.1, No.2, Aug., pp.151-64.

Sterling,L.,Shapiro,E. 1986
"The Art of PROLOG",
MIT Press Series in Logic Programming.
Cambridge : Massachusetts.

Tate, A. 1985
A Review of Knowledge-based Planning Techniques,
The Knowledge Engineering Review,
Vol.1, No.2, June, 1985.

Thomas,M. 1987
Implementing Algebraically Specified Abstract
Data Types in an Imperative Programming Language.
Proc. of TAPSOFT'87, Pisa, March,
LNCS. 250, Springer-Verlag.

van Melle,W.,Shortliffe,E.H.,Buchanan,B.G. 1979.
EMYCIN : A domain-independent system that aids in constructing knowledge-based consultation programs. Machine Intelligence, Infotech State of the Art Report 9, no.3.

Warren,D.H.D. 1974
WARPLAN : A System for Generating Plans,
Research Memo 76, Dept. of Artificial Intelligence,
Edinburgh University.

Warren,D.H.D. 1980.
Logic programming and compiler writing.
Software Practice and Experience, Vol.10, pp97-125.

Waterman, D.A., Hayes-Roth, F. 1983.
An Investigation of Tools for Building Expert Systems.
Chapter 6 of [Hayes-Roth etal83].

Waters,R.C. 1985.
The programmer's apprentice : a session with KBEmacs
IEEE Trans. SE., Vol. SE-11, No.11.

Weiss,S.M.,Kulikowski,C.A. 1979.
EXPERT : a system for developing consultation models.
IJCAI-6, pp942-947.

Westfold,S.J. 1984
Very-high-level Programming of Knowledge Representation Schemes,
Proc. of American Association on AI, pp.344-49.

Wile, D.S. 1982a
POPART : producer of parsers and related tools,
System Builder's Manual, TM-82-21,
USC/Information Sciences Institute.

Wile, D.S. 1982b
Program developments : formal explanations of implementations,
Report RR-82-99,
USC/Information Sciences Institute.

Williams, C. 1984
ART the advanced reasoning tool - conceptual overview.
Inference Corporation.

Wirth, N. 1971
Program Refinement by Stepwise Refinement,
Comm. of ACM, Vol.14, No.4, pp.221-27.

Zahdeh, L.A. 1965.
Fuzzy sets,
Information and Control 8 : pp.338-53.

Zedzierski, B. 1983
Knowledge-based project management and communication
support in a system development environment,
Tech. Rep. KES.U.83.3,
Kestrel Institute, Palo Alto, CA.