Inductive Acquisition of Expert Knowledge

Stephen H. Muggleton

Ph.D.

University of Edinburgh

1986

Dedication

This thesis is dedicated to Thirza, for all the happiness and fun between the lines.

Acknowledgements

The work described in this thesis was carried out while the author was in receipt of an SERC studentship. I am thankful for facilities provided by Edinburgh University Machine Intelligence Research Unit, Edinburgh University Department of Artificial Intelligence, Intelligent Terminals Ltd, Radian Corporation (Texas) and the Turing Institute (Glasgow). Especial thanks are due to my primary supervisor Professor Donald Michie who has been a source of constant inspiration and encouragement. I am most grateful to Dr Denis Rutowitz and Dr Timothy Niblett for their additional help and instructive supervision, to Charlie Riese and Steve Zubrick for allowing my participation in the structuring of the two major Mugol expert system applications EARL and WILLARD, to John Roycroft for his invaluable chess expertise used in the construction of the KBBKN strategic expert system and to Barry Shepherd who built the GENARCH robotic system using the Mugol environment. I must also thank Dr Peter Mowforth for showing interest in my work, Alen Shapiro for invaluable practical help and suggestions and Andy Paterson and Andrew Blake for many interesting conversations about inductive inference. My thanks to all the secretarial and administrative staff at Edinburgh University, especially Penni Montgomery, who helped with so many essential details. Thanks also to Ned, our servitor, who kept me constantly amused with his anecdotes, songs and cheery personality. I must not forget my good friend and companion Mike Bain who has always listened sympathetically to my ideas and broadened my view in many areas. Last and not least I would like to thank my parents, Louis and Sylvia, for their kind support and encouragement throughout.

Declaration

I declare that this thesis has been composed by myself and the work is my own.

Stephen H. Muggleton

Publications

Part of Chapter 3 has appeared as *RuleMaster: a second-generation knowledge-engineering facility* in the Proceedings of the First Conference on Artificial Intelligence Applications.

A slightly extended version of chapter 8 is to appear as *An efficient method of inductively acquiring chess strategies* in Machine Intelligence 11, Oxford University Press.

Abstract

Expert systems divide neatly into two categories: those in which (1) the expert decisions result in changes to some external environment (control systems), and (2) the expert decisions merely seek to describe the environment (classification systems). Both the explanation of computer-based reasoning and the "bottleneck" (Feigenbaum, 1979) of knowledge acquisition are major issues in expert systems research. We have contributed to these areas of research in two ways. Firstly, we have implemented an expert system shell, the Mugol environment, which facilitates knowledge acquisition by inductive inference and provides automatic explanation of run-time reasoning on demand. RuleMaster, a commercial version of this environment, has been used to advantage industrially in the construction and testing of two large classification systems. Secondly, we have investigated a new technique called *sequence induction* which can be used in the construction of control systems. Sequence induction is based on theoretical work in grammatical learning. We have improved existing grammatical learning algorithms as well as suggesting and theoretically characterising new ones. These algorithms have been successfully applied to the acquisition of knowledge for a diverse set of control systems, including inductive construction of robot plans and chess end-game strategies.

Table of contents

Appendix G - KBBKN Mugol code generated from Appendix F (see ch.8)

Appendix H - GENARCH: a practical solution to general arch building

Table of figures

1

Overview

Abstract. With reference to Michie's definition of "expert systems" (Michie, 1985) we discuss and emphasise the necessity for machine-executable knowledge to be comprehensible to human experts. Expert systems are shown to divide neatly into two categories: those in which (1) the expert decisions result in changes to some external environment (control systems), and (2) the expert decisions merely seek to describe the environment (classification systems). Both the explanation of computer-based reasoning and the "bottleneck" (Feigenbaum, 1979) of knowledge acquisition are major issues in expert systems research. We have contributed to these areas of research in two ways. Firstly, we have implemented an expert system shell, the Mugol environment, which facilitates knowledge acquisition by inductive inference and provides automatic explanation of run-time reasoning on demand. RuleMaster, a commercial version of this environment, has been used to advantage industrially in the construction and testing of several large classification systems, two of which we describe in appendices. Secondly, we have augmented grammatical learning techniques and successfully applied these to the acquisition of knowledge for a diverse set of control systems, including inductive construction of robot plans and chess end-game strategies.

1.1. What are expert systems?

In recent years *expert systems* have been exciting a great deal of interest in the computational and cognitive sciences. What, however, do we mean by an "expert system"? An often-repeated but naive definition says that an expert system is a program that solves problems which would otherwise require a highly skilled human for their solution. Powerful, but essentially "black box" problem solvers, such as autopilots in the aircraft industry or the MACSYMA system (Moses, 1975) for symbolic manipulations, would qualify under such a definition. The definition is generally extended to include the property of self-explanation. Thus Feigenbaum (1979) required that the

system "be able to explain its activity; else the question arises of who is in control ..." and sees the issue as crucial to user acceptance. However it would be naive to go to the other extreme and say that an expert system is a computer program which has all the behavioural characteristics of a human expert. Clearly the latter definition is not restrictive enough as we would not expect a computer program to go out for lunch at midday. For the purposes of this thesis, we will adopt Michie's (1985) definition of an expert system.

> An expert system embodies in a computer the knowledge-based component of an expert skill in such a form that the system can generate intelligent actions and advice and can also on demand justify to the user its line of reasoning.

With reference to this definition we should stress two important aspects; firstly an expert system's task performance should be demonstrably at least as good as that of a human expert in the given domain, and secondly the system must be capable of explaining its line of reasoning on demand. We should note that this definition does not exclude problem domains too complex for human specialists to have acquired mastery. If, by some means, a program were constructed which gave an account of its faultless play of the chess end-game of King and two Bishops against King and Knight in terms which end-game specialists could understand and apply, then the program would qualify as an expert (superexpert?) system. Expert systems are the knowledge-based subcategory of a larger class of computer programs which display good task performance based upon large amounts of information (for a definition of the technical meaning of the term *knowledge* used here, see (Michie, 1982)). This larger category we will call *domain-specific problem solvers*. The information for the latter class may be held implicitly, as in search-driven game-playing programs, or explicitly as look-up. For the above mentioned expert-inscrutable Bishop-Bishop-Knight ending a tabulation by K. Thompson (Roycroft, 1983; Thompson, 1985) of the complete space of several hundred million legal positions can be made to yield optimal play. Note that this does not constitute an expert system but does exemplify the larger class.

Both domain-specific problem solvers and expert systems can be partitioned into the following two distinct classes.

a) **Classification systems.** In a classification system like MYCIN (Shortliffe and Buchanan, 1975), it is assumed that the state of the world being classified does not change during the operation of the system. Thus, if some statement could be made about the situation being classified at the beginning of the process of classification, the same statement would hold throughout.

b) **Control systems.** In a control system such as VM (Fagan et al, 1979), no such steady-state assumption holds, and indeed it is generally part of the function of the system to carry out operations which change the state of the world.

1.2. How expertise is acquired: the debug cycle

Many of the problems of building and using an expert system involve the human interface. The reasons for this, over and above the usual problems of providing user interfaces for normal programs, lie in the fact that an expert system, like an expert's knowledge, is never complete. Once the system has been shown to perform well in some domain, its owner typically seeks either to improve on this proficiency or expand the scope of expertise. Thus expert systems are often continually being debugged. Moreover, this debugging cycle does not only involve a programmer or knowledge engineer but also a human expert. Experts, though knowledgeable in their own field cannot be expected to be acquainted with programmer-oriented tools. Thus in the study of expert systems we need to be able to answer at least the following questions.

1) **How does the expert assess the system's present knowledge?** The answer seems to lie in allowing the expert to use example test-cases. The expert checks a natural language representation of the system's reasoning in particular example cases and uses this to infer the reasoning process. This was first shown to be effective in the MYCIN project (Shortliffe and Buchanan, 1975).

2) **How does the expert alter the systems knowledge?** By symmetric analogy to the first answer, we would expect that experts should be allowed to express, by example, their reasoning to the system in a natural format. The system could then infer how its own

reasoning process must adapt to fit that of the expert. This methodology was first tested by Michalski and Chilausky (Michalski and Chilausky, 1980) who found it gave excellent results.

Algorithms, like that used by Michalski et al., which infer generalised descriptions from example material are collectively termed *inductive inference* algorithms. If the outputs of such algorithms are in user-intelligible and mentally checkable form, then it is customary to speak of *inductive learning* algorithms. The study of such algorithms and their application in expert system technology forms the main thrust of this work.

1.3. Induction

In the following sections we give a brief overview of the uses of inductive algorithms. A fuller review of the subject of inductive inference is given in chapter 2.

1.3.1. Static induction

Logical induction is the process of generating concept descriptions which are either equivalent to or more general than some set of examples describing that concept. Typically descriptions generated by inductive algorithms are more compact than the original example set. As these descriptions can generally be executed, induction can also be viewed as an automatic programming technique. Work originating at Illinois (Michalski and Chilausky, 1980) and extended at Edinburgh (Shapiro, 1983) has shown the potential for constructing expert knowledge bases in the relaxed framework of concepts generalised from examples. Commercially available packages (McLaren, 1984; A-Razzak, Hassan and Pettipher, 1984; Michie, Muggleton, Riese and Zubrick, 1984) have already, during their short existence, proved the power of this approach, with development-time savings in building large expert systems of at least an order of magnitude over the traditional "deductivist" rule extraction technique. By the traditional method (e.g. MYCIN), rules are extracted during a long series of interviews between the knowledge engineer and the expert. However, it cannot be said that the "inductivists" have yet completely met their targets.

The ultimate learning/knowledge environment might be a "laboratory" in which the scientist (expert), unaided by knowledge engineers, uses machines to carry out experimentation and theory formation by mechanisms which learn from mistakes. Some form of inductive engine must surely lie at the core of any such environment.

Under the present inductive regime, examples represent *static descriptions* of world situations to which labels are attached indicating a classification or action to be taken. The world described often has a finite number of distinguishable states or *situations*. Rules may be developed inductively in various different formalisms ranging from decision trees (Quinlan, 1979), to propositional and predicate calculi (Michalski, 1983). The most serious problem to emerge from the development of this approach is that although inductive generators, when presented with sufficient example sets can generate efficient and correct rules, these rules can be so large and complex that they are incomprehensible to human experts (Quinlan, 1982b). Ease of comprehension is a crucial factor in the debug cycle of inductively generated knowledge. Two complementary approaches to this apparent impasse have so far been suggested

a) **Structured induction.** Shapiro (Shapiro, 1983) has shown that large expert domains can be effectively dealt with by employing the techniques of structured programming in an inductive environment. Thus, the expert is expected to structure his knowledge in a top-down fashion *manually*, and then present examples for each part of the hierarchy. These examples are then used to construct the individual rules (or decision trees) automatically. Facilities to aid this structuring process have been built into the Mugol environment which is described fully in chapter 3.

b) **Human subset languages.** Michie (Michie, 1984) has suggested that by constraining every constituent rule to take one or other of two alternative forms (either linear thresholded sum or linear decision tree) and by also constraining the form of the "calling diagram" relating linearised procedures to each other, rulebases can be constructed that are easily understood and force builders to structure their problem. Arbab (Arbab and Michie, 1985) has

implemented a new version of a rule-induction algorithm due to Bratko (Bratko, 1983) which constructs a linear rule where one exists and otherwise presents the most nearly linear tree which can be constructed from the data.

Manually structured induction is not the final solution. In any expert system building task, the structuring of the problem now becomes the new knowledge acquisition bottleneck. Attempts have been made (Michalski and Stepp, 1982; Paterson, 1983) to automatically structure domains from example material. These have used statistical clustering in an attempt to extract structural information from the example set. The results however, have not been very promising, with the machine's suggested hierarchies not necessarily having any significance to experts in the domain (Paterson, 1983). The primary reason for failure seems to lie in the fact that although the example set is a rich enough knowledge source for rule construction, additional information is necessary to indicate humanly comprehensible higher level structure. We suggest a new approach that offers a partial solution to this problem for control domains (see section 1.1, "Control systems"). The full theoretical basis for this approach is developed in chapters 5 and 7, with experimental results being given in chapters 6 and 8.

1.3.2. Sequence induction

The new approach relies on the presentation of example *sequences* to an inductive algorithm. Each element of the sequence is a situation/action pair similar in form to the static descriptions such as those described in section 1.3.1. These sequences can be taken to represent a series of world descriptions which are altered by actions operating on that world. The output of the inductive process is a finite state control structure in which each state contains a small number of the static description examples describing the actions and state transitions which should be carried out in various situations. These static description examples can be used by a static induction algorithm to produce rules or a decision tree for each state. *Although we do not produce a hierarchical structure, we achieve some of the aims of structured induction (i.e. a set of small understandable rules) by making use of the additional structural information which lies within each*

example. Whereas structured static induction seems to be of most use in classification systems, sequence induction lends itself more readily to the construction of control systems (see section 1.1). In terms of automatic programming, sequence induction is used to design the overall branching and looping structure of routines in a program, while static induction is used at a lower level to decide the internal ordering of nested if-then-else statements.

The basis for sequence induction techniques lies in the study of grammatical inference, that is the inference of grammatical structures from example sentences of a language (see chapter 5). Under the generative paradigm of computational linguistics, the grammar produced can be viewed as the control structure of a program which generated the example sentences. Some of the earliest work in the area of grammatical induction was done by Biermann and Feldman (Biermann and Feldman, 1972), who devised an algorithm to induce a finite state automaton representing a particular language from example strings contained in that language. Although their algorithm was capable of identifying any regular language given a sufficient example set, the algorithm requires an arbitrary complexity parameter and also has rather low *example efficiency* (ie. a large number of examples are needed to infer anything). Angluin (Angluin, 1982b) has described an algorithm which infers only a limited subset of the regular languages. This subset she calls the k-reversible languages. By limiting the target result set, Angluin's algorithm achieves example efficiency higher than that of Biermann and Feldman's algorithm.

The author (see chapter 7) has taken Angluin's algorithm and redesigned it to run in linear time complexity rather than Angluin's original $O(n^3)$ time. Furthermore, we have discovered an even smaller, but useful subset of the k-reversible languages, which we call the k-contextual languages. The algorithm for inferring members of the k-contextual languages is again more example efficient than even Angluin's, to the extent that inference is possible from samples containing only a single example (all other methods in the literature presuppose more than a single example). We have also discovered a method that circumvents the need for supplying the algorithm with an arbitrary complexity measure, something which is required by all other methods in the

literature (Angluin, 1982b; Biermann and Feldman, 1972; Levine, 1982; Miclet, 1980). We achieve this by making use of the semantic content of static examples in the construction of finite state schemas.

1.4. An environment for inductively acquiring expert knowledge

In order to test the hypothesis that expert control knowledge can be acquired conveniently by inductive inference, it is necessary to have an environment which is capable of being used for the routine inductive construction and execution of knowledge-bases. It is desirable that this environment have the following properties

a) **An induction engine.** Inductive apparatus both for *static* and for *sequence* induction.

b) **A rule language.** This is used to express the output of a) in a natural and comprehensible fashion. Such a language should provide support for problem structuring and the manipulation of whatever data is needed for the task at hand.

c) **An explanation facility.** According to the definition of section 1.1 this is necessary for the construction of any expert system.

In chapter 3 we discuss the Mugol environment which has been constructed to meet these requirements. The only above requirement not yet fully met is the integration of the sequence induction algorithm with other tools in the Mugol environment.

1.5. Applications

1.5.1. Applications of static induction The Mugol environment has been tested in the construction of two major expert classification systems. The techniques used for the construction of both of these were based on the structured static induction methodology first advocated and tested by Shapiro and Niblett (Shapiro and Niblett, 1982). The systems were

a) **WILLARD.** WILLARD (Zubrick, 1984) is an expert system for predicting the likelihood of severe thunderstorms occurring in the central USA. The system was written by Steve Zubrick,

a meteorologist at Radian Corporation. Extensive testing of the system (Zubrick, 1986) has shown that it is capable of producing predictions which usefully complement those of the US National Weather Service. A fuller description of WILLARD can be found in Appendix A.

b) EARL. EARL (Riese, 1984) is a system for diagnosing imminent break-down in large oil-cooled electrical transformers. The system was constructed by Charles Riese who is a software engineer working for the Hartford Steam Boiler Company. When EARL was tested against 859 test-cases, it managed a diagnostic success rate in excess of 99%. EARL is now in routine industrial use. A fuller description of EARL can be found in Appendix B.

The author gave help and advice in the structuring and example acquisition of both the above systems. Details of a smaller control system called ARCH are given in chapter 4.

1.5.2. Applications of sequence induction

The author has shown (see chapter 6) that sequence induction can be applied successfully to a diverse set of problems, including automatic VLSI circuit synthesis, user modelling in a mathematical educational environment and generalisation of robot plans.

More recently, using sequence induction, we have successfully built an expert system for playing a fragment of the chess endgame domain of King and two Bishops against King and Knight (see chapter 8). This endgame is so complex that even the chess endgame specialist, John Roycroft, has failed after months of continuous study, aided by unlimited access to machine-generated facts and variations, to acquire more than a partial and patchy understanding of it. However, when presented with the expert system built from his example move sequences, he easily recognised and agreed with the various states of the generalised structure which had been built. The most complex states (in terms of the number of static examples placed), were precisely those which Roycroft had spent most time describing in the sequence acquisition stage. The expert system although very compact, is also very easily comprehended.

1.6. Conclusion

In this chapter we introduce the topic of expert system research, following Michie's definition of an expert system. Expert system development involves continuous debugging of knowledge structures. We argue that the two most important tools in this debugging process are a) an *explanation facility* and b) an *inductive knowledge acquisition mechanism.* The major topic of interest within this thesis is that of inductive inference.

We describe two different forms of induction. *Static induction* algorithms take examples which represent descriptions of world situations to which labels are attached. These labels indicate a classification or an action to be taken. On the other hand, *sequence induction* relies on the presentation of example sequences to an inductive algorithm. Each element of the sequence is a situation/action pair similar in form to the static descriptions.

We have constructed an expert system construction environment called the Mugol environment. This comprises an *induction engine*, a *rule language* and an *explanation facility*. Although the Mugol environment in its present form only has facilities for static induction, it would be a simple matter to introduce a sequence induction package based on the techniques developed in this thesis.

Generally whereas static induction techniques have been found to be effective knowledge-acquisition methodologies in classification domains such as chess classification (Quinlan, 1982b; Shapiro and Niblett, 1982; Shapiro, 1983), weather forecasting and transformer diagnosis (see chapter 4), we believe sequence induction to be similarly promising as a strategy-acquisition methodology in control domains such as robot planning (see chapter 6) and chess endgame play (see chapter 8).

2

Inductive inference

Abstract. The notion of inductive inference is discussed with reference to its inverse, deductive inference. We point out that inductive algorithms do not necessarily carry out generalisation. Inductive algorithms take example world descriptions and produce executable rules. Properties of example material and inductive algorithms are described and discussed. By way of a simple example we show the strengths and weaknesses of various rule formalisms. The basis for the Mugol language is given, with emphasis being placed on transparency of expression and the ability to inductively generate the finite state control structure of Mugol modules.

2.1. Generalisation

Deduction is the process of deriving specific statements from more general ones. For instance, let a list be an ordered set. Thus

["John", "Mary", "Harry"]

is a list of people's names. A legitimate corresponding general statement concerning the members of a list might be the following

X is a member of the list L if X is the first element of L or if X is a member of the list formed by removing the first element of L. (1)

Many statements are deductively derivable from the statement of list membership. For instance

"John" is a member of the list L, where L = ["John", "Mary", "Harry"] since "John" is the first element of L. (2)

Note that by using deductive inference we could, in principle, replace all general statements s such as statement (1) by a (potentially unlimited) set S of particular statements such as statement (2).

The inductive algorithms described in this thesis take a set of descriptive statements E, called the example set, and propose a new set of descriptive statements R, called the rule set, where each example $e \in E$ can be deductively derived from some rule $r \in R$. In addition, there may or may not exist a descriptive statement $e' \notin E$ which can be deductively derived from a rule in R. Such a descriptive statement e' which is derivable from R but not originally given in E is a "guess". These guesses are usually introduced in order to compact R as much as computational trade-offs permit (see also later discussion of Popper's philosophical comment on inductive generalisation). It can therefore be said that these inductive algorithms generate a rule set R which is *more general or equivalent to* the given example set E, i.e. if E' is the set of all deductions from R then $E' \supseteq E$. The phrase "inductive inference" for such a process is in some ways unfortunate since it may to some readers imply the *necessary* involvement of generalisation.

The advantage of employing an inductive algorithm in developing expert systems seems to lie not necessarily in the inductive algorithm's ability to "guess" the classifications of previously unseen examples, but rather in a particular psychological fact: whereas experts can easily suggest particular situations to which they can apply their knowledge, it is much more difficult for them to formulate general rules.

2.2. Examples and rules

Expert systems carry out tasks relating to a particular though not necessarily explicit *world*. For instance, in medical expert systems the world is the set of possible states of a patient's body together with relevant anatomical and physiological laws, while in game-playing domains the world might be the set of legal arrangements of pieces on a board together with the laws of chess. Often the program will contain a model of the world, which is some simplified and abstracted representation of the world in question, containing enough detail for the program to work on.

For any particular model of the world, there exists a set of *situations* or arrangements of the model, e.g. in the chess domain, a "ground model" might have as its set of situations the complete set of different arrangements of pieces on the board. These situations can be abstracted by defining a set of *attributes* (relationships between parts of the world) in terms of which a particular situation can be described. In the chess world one possible attribute might be *the black king is in check.*

In a classification problem, such as deciding whether a chess endgame is a win-for-white (Shapiro, 1983) an *example* consists of a particular situation described in terms of the values of all relevant attributes together with the classification given. For instance, we might have

Example: win-for-white if A and B and C and D and E

where A, B, C, D, and E are a set of attributes, observable from the world model, which are used to decide whether white can force a win. A general rule derived from this example might be

Example: win-for-white if B

Note that the rule does not necessarily use all the attributes necessary to describe a particular world situation: an ideal rule uses as few attributes as possible. In the case of Shapiro's work, one such simple rule for the KPa7KR domain was that *white wins if it can safely capture the black rook.*

In general we call the input to an inductive algorithm the *example set* and the output the *rule set.* Examples and rules may take a very different form from those shown above, for instance in sequence induction (chapters 5,6,7,8).

2.2.1. Ordering

Ideally we would want the order in which examples are presented to an inductive algorithm not to affect the results given. In fact this is true for all inductive techniques dealt with in this thesis. Not only is the order of examples essentially irrelevant to these techniques but also the order of attributes given has no effect on the result. In contrast, in sequence induction the internal ordering within particular examples does, justifiably, affect results of induction (see chapters 5,6,7,8), remembering that in this context an "example" includes the specification of a particular

ordering.

2.2.2. Types of example material

Examples presented to an inductive algorithm are said to be either *positive* or *negative* in the sense that they respectively exemplify or counter-exemplify the concept being conveyed. Moreover, the nature of example material differs according to the method in which it is employed. In this thesis a source of examples is referred to as an *oracle* when it can be interrogated interactively by an inductive algorithm. Alternatively, when examples are given according to some fixed presentation scheme, the source of examples is said to be *text*. Inductive algorithms in this thesis use textual example material.

2.3. Criteria for inductively generated results

It is important to be able to make sound statements about

a) **Algorithmic effectiveness.** What results do we expect the algorithm to be able or unable to produce, or not produce.

b) **Validity of induced results.** Since any inductive process potentially involves the assertion of facts which are not originally given as being true, the question arises as to how confident one can be in the application of the resultant information.

We investigate these problems in the following sections.

2.3.1. Effectiveness

Suppose an inductive algorithm A conjectures a series of rule sets

$$R_1 \ R_2 \ R_3 \ldots$$

in response to an enumeration of instances of a given rule set R. Each R_{i+1} is proposed as soon as the rule set R_i is found to be inconsistent with the instance source (oracle or text). The algorithm A is said to *identify in the limit* the rule set R if and only after a finite amount of time A proposes the

rule set R_i, which is equivalent to R, and does not subsequently change this proposal. We discuss the notion of "identification in the limit" further in chapters 5 and 7 in the context of sequence induction.

Clearly, the ability to prove that an algorithm will identify in the limit some rule set from a class of rule sets *effectively* imbues the user with confidence in that algorithm. However, from the outsider's viewpoint, if the specification of the target total set is not available, it will never be clear that any particular guess made by such an algorithm will not be subsequently changed. Hence one cannot necessarily ever have complete confidence in any particular result generated by such an algorithm if it is known only that a correct rule-set will eventually be generated. It is therefore also necessary to have some independent means of verifying inductive results.

2.3.2. Verification of inductive results

Induction is believed by some to be logically and philosophically unsound since it is not possible in general to positively prove a generalisation of some facts, only disprove it (Popper, 1972). Thus we might, on the basis of experience of a number of example cases, have hypothesised the descriptive law that the sun will rise every morning. This hypothesis could not be positively proved, only disproved. Although these objections are perfectly valid, it is widely agreed that all human and animal knowledge is acquired on the basis of experience†. Thus it is not the usefulness of inductive inference which is in question but rather the way in which confidence can be gained in its results.

Let us separate the class of computable functions into the sub-classes F (finite) and I (infinite). Functions within F have a finite domain while functions within I have an infinite domain. Clearly hypothesised functions within F have a finite number of instances. Thus these could, in principle, be completely verified by enumerating all expressible problems and checking the solution's results against an oracle. However, functions hypothesised within I cannot be

† It might be argued that many behavioural responses, especially instinctual ones, are inherited genetically which can be interpreted as indicating that these responses are "learned" by evolution.

exhaustively proved like this since a potentially infinite number of instances exist for any function within *I*. Instead we might turn to the method of *mathematical induction** for a proof here. By this technique we attempt to find an incremental operator which if applied repetitively can deductively derive all possible instances from a set of base statements. If the base statements are found to be valid according to the oracle, and an arbitrary application of the operator to non-base statements also produces statements consistent with the oracle, then we can infer the entire set of statements to be correct. As this method of proof is often used to prove human-generated programs it seems reasonable to use the same technique for machine-generated programs. This latter technique has not been applied to inductively-generated programs in this thesis, but is believed to be a pressing topic for further research.

2.4. Languages involved in inductive inference

If inductive inference is to be used as an automatic programming tool, as is the case with the Mugol environment (see chapter 3), it should be recognised that there are three languages involved.

1) **Implementation language.** In terms of logic, this is a meta-language which is used to inductively transform examples into rules. In the Mugol environment the implementation language for the inductive rule generator is the programming language C.

2) **Example language.** This language comprises a simplified symbolic representation of decision-making in the world being modelled. In this thesis, as explained in chapter 1, we deal with two distinct forms of example material:

   a) *static* examples, represented by <situation/classification> pairs.

   b) *dynamic* or *sequence* examples, consisting of sequences of <situation/action> pairs which represent a changing situation controlled by actions effected on the world.

3) **Rule language.** The rule language expresses descriptive or prescriptive statements about the world. In an inductive programming environment, such rules are fragments of some

* Note that this is not related in any simple way to "induction" as used generally in this thesis.

executable program or knowledge-base. In the Mugol environment, classification rules are expressed as multi-branching decision trees. Control information is expressed in the format of recursive transition networks, i.e. a set of finite state automata (see section 2.6) which call each other. A desirable feature of a rule language in expert systems work is that its expressions should correspond well with the expert's own conceptual representations.

2.5. Classification learning

Simple concepts can be stated in logic as propositions in the propositional calculus. For instance we might define the concept of *bird* as follows

fred-is-bird if fred-has-wings and not(fred-is-aeroplane) (3)

Note that this could either be viewed as an *example* of a bird or as a general *rule* for recognising birds. The example language of Quinlan's ID3 inductive algorithm (Quinlan, 1979) recognises only tabulated disjunctions of conjunctions where each conjunction (or example) could be directly translated into a statement such as (3).

ID3's rules are generated as binary decision trees in a language having the equivalent of the context-free grammar

rule → TRUE

rule → FALSE

rule → IF attribute THEN rule ELSE rule

where *attribute* is simply the name of a test. Although simple, a generalised form of ID3 has been found in this thesis to be useful in the construction of large expert systems (see chapter 3 and 4 and Appendices A and B). However, a small artificial example developed in the following sections shows the limitations imposed by the expressibility of the output of such inductive inference.

2.5.1. Parity problem (unstructured)

Imagine that we want to inductively develop a rule for deciding whether a set of four truth values contains an even number of *trues*. This is normally called the problem of *even-parity*. We give ID3 a full tabulation of all examples as shown in figure 2.1. *CLASS* is the value of Even-parity for the particular situation. ID3's result is shown in figure 2.2.

Far from compacting the data in this case (as ID3 generally does), the decision tree has as many leaves as examples, and is very difficult to understand due to its bulk. This is admittedly a worst-case problem for ID3, however it should be asked how problems of this type can be solved, as similarly difficult problems may turn up in real world situations. Shapiro and Niblett (1982) have suggested the use of structuring to simplify the solution of such problems. Can structuring help in solving the parity problem?

Attribute-1	Attribute-2	Attribute-3	Attribute-4	CLASS
false	false	false	false	TRUE
false	false	false	true	FALSE
false	false	true	false	FALSE
false	false	true	true	TRUE
false	true	false	false	FALSE
false	true	false	true	TRUE
false	true	true	false	TRUE
false	true	true	true	FALSE
true	false	false	false	FALSE
true	false	false	true	TRUE
true	false	true	false	TRUE
true	false	true	true	FALSE
true	true	false	false	TRUE
true	true	false	true	FALSE
true	true	true	false	FALSE
true	true	true	true	TRUE

Figure 2.1 Examples of Even-parity

```
IF Attribute-1    THEN
        IF Attribute-2    THEN
                IF Attribute-3    THEN
                        IF Attribute-4 THEN     TRUE
                        ELSE                    FALSE
                ELSE    IF Attribute-4 THEN     FALSE
                        ELSE                    TRUE
        ELSE    IF Attribute-3    THEN
                        IF Attribute-4 THEN     FALSE
                        ELSE                    TRUE
                ELSE    IF Attribute-4 THEN     TRUE
                        ELSE                    FALSE
ELSE
        IF Attribute-2    THEN
                IF Attribute-3    THEN
                        IF Attribute-4 THEN     FALSE
                        ELSE                    TRUE
                ELSE    IF Attribute-4 THEN     TRUE
                        ELSE                    FALSE
        ELSE    IF Attribute-3    THEN
                        IF Attribute-4 THEN     TRUE
                        ELSE                    FALSE
                ELSE    IF Attribute-4 THEN     FALSE
                        ELSE                    TRUE
```

Figure 2.2 ID3 decision tree for Even-parity

2.5.2. Parity problem (structured)

One structuring method is the following. Let the top-level decision, "Even-parity" be based on two sub-attributes

1) *First-half-even.* This checks whether the attributes *Attribute-1* and *Attribute-2* have between them an even number of trues.

2) *Second-half-even.* This checks whether the attributes *Attribute-3* and *Attribute-4* have between them an even number of trues.

Figures 2.3, 2.4 and 2.5 show the examples for the new "Even-parity", "First-half-even" and "Second-half-even" sub-problems respectively.

First-half-even	Second-half-even	CLASS
false	false	TRUE
false	true	FALSE
true	false	FALSE
true	true	TRUE

Figure 2.3 Even-parity

Attribute-1	Attribute-2	CLASS
false	false	TRUE
false	true	FALSE
true	false	FALSE
true	true	TRUE

Figure 2.4 Examples of First-half-even

Attribute-3	Attribute-4	CLASS
false	false	TRUE
false	true	FALSE
true	false	FALSE
true	true	TRUE

Figure 2.5 Examples of Second-half-even

```
IF First-half-evenTHEN
        IF Second-half-even THEN        TRUE
        ELSE                            FALSE
ELSE    IF Second-half-even THEN        FALSE
        ELSE                            TRUE
```

Figure 2.6 New ID3 decision tree for Even-parity

Note that although the examples are still tabulated fully, only 12 examples are needed rather than the original 16 for figure 2.3. Note also, coincidentally, that since all three example sets are identical in terms of the examples present, we only show the new decision tree for "Even-parity" in figure 2.6, the other two decision trees having the same outline. ID3 can do no compaction in generating these trees, though this time, since the tree is small, it is easier to understand.

2.5.3. Example and description complexities

It is of interest to note the number of examples, size of description and description execution time for unstructured and structured solutions of the parity problem. Let N be the number of primitive attributes‡ used (i.e. the original number of truth-values being dealt with). No decision can be made about even-parity without considering all primitive attribute values.

Let us consider first the unstructured solution. In this solution it is necessary to present ID3 with all possible examples in order to obtain a correct decision tree (see figure 2.1), i.e. 2^N examples are required (in figure 2.1 $2^4 = 16$). Again, since all primitive attribute values must be considered in making any decision on even-parity, all leaves of the unstructured decision tree will need to be at maximum possible depth in the tree, i.e. depth N. There must be 2^N leaves in such a decision tree (one for each example), and by simple summation of nodes at different levels there must be $(2^{N+1}-1)$ nodes of them (in figure 2.2, $2^{4+1}-1 = 31$). When executing the tree the number of decisions made in gaining any particular result is always N, the maximum depth of the tree.

‡ By primitive attributes we mean those that are not described hierarchically in terms of any sub-attributes.

Thus we say that the unstructured solution requires $O(2^N)$ examples, $O(2^N)$ description space and $O(N)$ time to execute the description.

Let us now consider the structured solution. We can extend the solution shown in figures 2.3-2.5 to deal with any number of primitive attributes N, by using sub-attributes which repeatedly break the attribute set in half. For instance, given 8 attributes, "First-half-even" would have the sub-problems of "First-first-half-even" and "Second-first-half-even" instead of Attribute-1 and Attribute-2 (see figure 2.4). It works out that the number of examples needed for N primitive attributes is $4(N-1)$ (in the solution of figure 2.3-2.5, $4 \times 3 = 12$ examples). The descriptive size of the solution, i.e. the total number of nodes in all trees in the solution, is $7(N-1)$ (21 in the solution of the example set of figures 2.3-2.5). In order to decide on even-parity using the structured set of trees it is necessary to execute all internal nodes of all trees in the solution, which in general amounts to $3(N-1)$ nodes (9 nodes in the solution of figures 2.4-2.5). Therefore the unstructured solution requires $O(N)$ examples, $O(N)$ description space and $O(N)$ time to execute the description. The complexity results for various representational methods of describing the parity problem are summarised later in figure 2.8.

The author is not aware of any such quantitative analysis of worst case for structured and unstructured induction presented elsewhere. However, these results are in line with Shapiro's observation (1983) that in his chess end-game solution for the domain of KPa7KR, while the execution time for structured solutions is not noticeably different from that of unstructured solutions, considerably fewer examples are needed and the total size of description produced is smaller.

Thus structuring can be an effective technique both for compaction and for understandability. However, let us change the parity problem slightly and try to build a decision tree for checking even-parity of an arbitrarily long stream of truth-values.

Let us imagine that an inductive algorithm has a training phase in which examples of even-parity are presented and a rule set is produced. With either the unstructured or structured ID3

solution, an example set can only define a window into a finite portion of the stream of truth-values. Remembering that no decision can be made about even-parity without checking all primitive attribute values, the generated decision tree will work only for segments of the truth-value stream which are less than or equal to the length (number of attributes) described by the example set. In order to deal with sequences of arbitrary, unbounded or even infinite length, using a finite amount of training, we must turn to finite state machine theory.

2.6. Finite state automata and strategy learning

A *finite state automaton* is a mathematical model of a controller having a discrete set of inputs and outputs. This controller has a predefined fixed set of internal states through which it passes in carrying out its function. The output response of the controller to any set of inputs presented to it is determined not only by the input values, but also by the value of its internal state.

2.6.1. Finite state acceptors

A finite state *acceptor* is a limited form of finite state automaton which has an output repertoire of {accept, reject} when applied to any particular series of symbols.

2.6.2. Parity example revisited

Let us reconsider the "parity problem" of section 2.5, and pose this as a problem for a finite state acceptor. The allowable symbols for any element of a sequence presented to a parity checking automaton are chosen from the set {true, false}. The acceptor has two states, "Even-so-far" and "Odd-so-far". When in state "Even-so-far" the acceptor will report with the action "accept", and will give the answer "reject" when in the state "Odd-so-far". If the string of symbols terminates leaving the acceptor in the state "Even-so-far" we will know that there were an even number of trues, otherwise there were an odd number. Figure 2.7 illustrates such an automaton. In figure 2.7 circles represent states. A state is denoted by a double circle when it is associated with an *accept* output and by a single circle for the output being *reject*. Labelled arrowed arcs joining circles represent transitions that are taken between states on recognition of particular symbols in the input

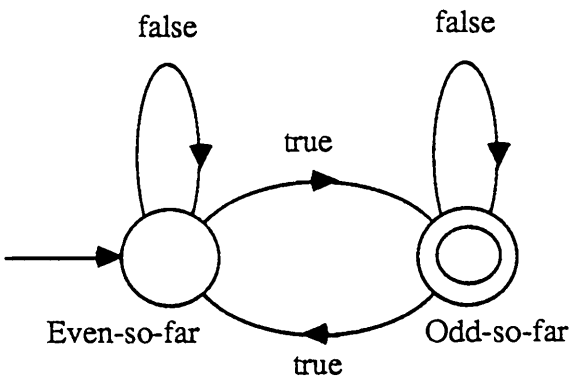


Figure 2.7 The Even-parity finite state acceptor

sequence. The unlabelled arc leading into the state "Even-so-far" from the left indicates that this is the state in which the automaton starts when presented with the first symbol in the sequence.

Finite state acceptors have especial importance in the theory of formal languages. It can be shown (Hopcroft and Ullman, 1979) that the class of languages acceptable by finite state automata is exactly that of the regular languages. In these terms strings of true/false values accepted by our even-parity acceptor would be sentences in the language of even-parity.

2.6.3. Formal definition of finite state acceptors

We formally denote a finite state acceptor by a 5-tuple $(Q, \Sigma, \delta, I, F)$ where Q is a finite set of states ({Even-so-far, Odd-so-far} in example), Σ is a finite input alphabet ({true, false} in example), $I \subseteq Q$ is the set of initial states ({Even-so-far} in example), $F \subseteq Q$ is the set of final or accepting states ({Even-so-far} in example) and δ is the transition function mapping $Q \times \Sigma$ to Q (the labelled arcs of figure 2.7). That is, $\delta(q,a)$ is a state in Q for each state q and input symbol a.

2.6.4. Complexity measure for finite state parity solution

In terms of the complexity measures of section 2.5.3, it can be seen that two decision nodes are needed in the finite state solution of figure 2.7 irrespective of the number of truth-values inspected. Thus the descriptive complexity is constant, which we denote $O(0)$. In terms of execution steps, as always we need to look at all N of the values in any particular string of values to decide on even-parity. Thus the time complexity is still $O(N)$. In chapter 5 we will show that parity acceptors can be built by the 0-reversible induction algorithm using a fixed number of examples. The example complexity using this algorithm is thus constant, or $O(0)$.

Figure 2.8 sums up the various complexity results for different representations of the even-parity problem solution.

2.6.5. Mealy and Moore machines

Although finite state acceptors are a powerful model for representing predicates which decide whether or not a string of symbols belongs to a particular set of such strings, a general purpose controller needs to be able to produce more than the two outputs {accept, reject}. There are two different formalisms used to generalise the notion of finite state acceptors to automata capable of producing a selection of more than two outputs.

	Unstructured tree	Structured trees	Finite state machine
Example complexity	$O(2^N)$	$O(N)$	$O(0)$
Description complexity	$O(2^N)$	$O(N)$	$O(0)$
Execution time	$O(N)$	$O(N)$	$O(N)$

Figure 2.8 Complexity results, N = number of binary variables dealt with

a) **Moore machines.** The output values are paired with particular states. Figure 2.9 shows the Moore machine version of an "Even-parity" finite state automaton. Note that if the machine in figure 2.9 had more than two outputs it would need more than two states.

b) **Mealy machines.** The output value is paired with particular inputs. Figure 2.10 shows the Mealy machine version of an "Even-parity" finite state automaton.

A specific form of Mealy machine, namely deterministic uniquely terminated Mealy machines (DUTMMs) are described in chapter 7. DUTMMs are the basis of the control within modules of Mugol programs (see chapter 3). These machines have a unique goal state, entry into which causes termination of the module's execution. The input and output symbol pairs which label the arcs of a Mealy machine equate to particular situations which cause actions to be fired, with a simultaneous state transition. In turn, the situation vectors represent sets of callable Mugol modules, each of which returns a value. The action is a callable Mugol module which does not return a value. In the

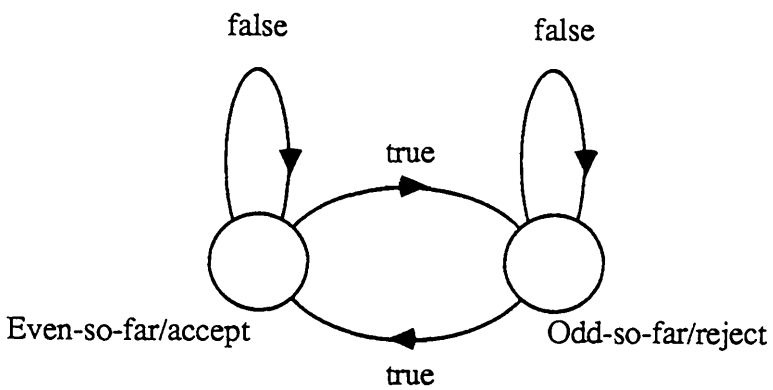


Figure 2.9 The Even-parity Moore machine

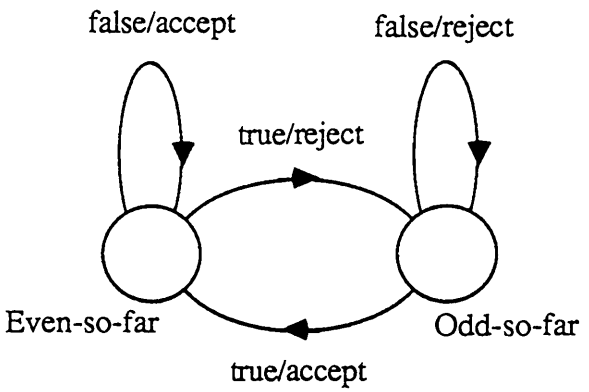


Figure 2.10 The Even-parity Mealy machine

next section we investigate the expressive power of Mugol modules.

2.6.6. Expressive power of DUTMMs

In order to show that DUTMMs have more expressive power than decision trees, it is necessary to show that they can describe all decision tree solutions to problems and that at least one other problem solution can be described using a finite state automaton but not using any structured or unstructured set of decision trees. The first condition, that DUTMMs can describe all decision tree solutions to problems, is shown to be true by figure 2.11. In this figure each situation/classification pair s_i/c_i $(1 \leq i \leq n)$ derivable from some decision tree is used to label the arcs leading from the start state of the DUTMM to the goal state, the particular decision c_i being returned on recognition of situation s_i. Clearly an automaton such as that in figure 2.11 can be constructed to be behaviourly equivalent to any given decision tree.

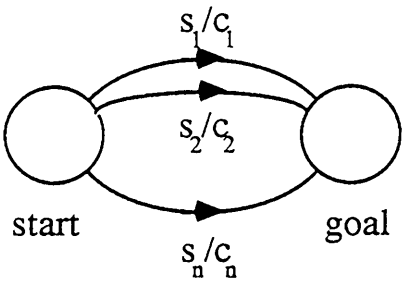


Figure 2.11 DUTMM equivalent of decision trees

The second condition is apparent from the fact that a one-way infinite lengthed string parity problem is insoluble by use of decision trees (see end of section 2.5.3), as any decision tree must be finite, though simply soluble as a Mealy machine (figure 2.10) which reports parity-so-far for any prefix of the one-way infinite string.

2.6.7. Limits on expressive power of finite state machines

Finite state automata, in turn have limits to their expressiveness, and form only one rung in the ladder of arithmetic expressiveness. Figure 2.12 describes a simple version of this hierarchy, Universal Turing Machines being the most expressive computational formalism.

In fact the only difference between a finite state machine and a Universal Turing machine is the latter's ability not only read to from a tape of symbols which can be moved backwards and forwards, but also to *write* symbols onto that tape. In our example the tape of symbols was the string of truth values which the finite state machine was allowed to read from. The extra ability to

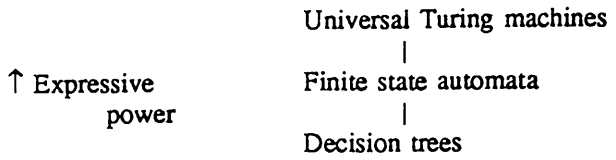


Figure 2.12 Hierarchy of arithmetic expressiveness

write to the tape equates to the use of variables and program stack in computer programs. As shown in the next section, the Mugol language (chapter 3) has these abilities, and therefore has the expressivity of a Universal Turing machine.

2.6.8. Recursion and variables

The Mugol language caters for recursively defined functions and procedures by allowing situational conditions and actions to be evaluated by the mechanism of procedure call. To complete the requisites of full computational expressiveness any high-level language must allow for the creation and manipulation of variables. Mugol permits the use of variables by the standard methods of variable declaration, value assignment and module parameterisation. The use of variables and recursion are illustrated in chapters 3 and 4.

2.7. Induction of finite state automata

Although in the last section we emphasised the expressivity of finite state representations over simpler classification formalisms, this aspect is secondary to human comprehensibility of these description formats. The transition mechanism of finite state automata has a direct parallel with the *goto* statement of many programming languages. As unconstrained use of *goto*s in programs is known to lead to highly opaque code which is difficult to create and maintain, it might be felt that we are trying to re-open a "Pandora's box" which has previously been almost successfully closed. In practice, knowledge engineers using the Mugol environment without any way of inducing finite

state control from sequence information have largely preferred to avoid the use of multiple state modules, and limited themselves to building classification systems by structuring a hierarchy of single state modules in the manner proposed and tested by Shapiro (1983).

However, human beings seem to find it easy and natural to generate "control plans" consisting of sequences of <situation/action> pairs. If such "plans" could be used by an inductive algorithm to produce finite state automata the problem of origination and maintenance would be eased considerably. Algorithms which do so are presented, together with examples of their use in chapters 5-8.

This still leaves the problem of opacity of automatically generated finite state automata. This might be approached in a similar fashion to that suggested for decision trees (Michie, 1984). Thus inductive algorithms would be limited, by some constraint, so that only humanly comprehensible finite state automata were produced. For instance the two state automaton of figure 2.10 seems quite comprehensible, though a twenty state automaton with arbitrary transitions would be very difficult to understand. One approach that suggests itself immediately is to place an upper limit upon the product of the number of states and transitions which is acceptable in an automatically generated finite state automaton. However, in this thesis we have not dealt with these aspects of comprehensibility.

2.8. Conclusion

In this chapter we characterise the nature of inductive algorithms. Inductive algorithms use various types of example material to generate hypotheses in various rule formats. By definition, inductive algorithms make "guesses" concerning unknown facts. These guesses must be shown to be sound according to some demonstrable criteria.

In sections 2.5 and 2.6 we use the "parity" problem to illustrate properties of various rule representations. In figure 2.8 we give a table of complexity results for the three chosen representations. This table shows that, for this problem at least, it is preferable to use a finite state machine representation rather than a decision-tree based one. We go on to show that finite state

machine representations have more expressive power than those of propositional calculus and decision trees. However, there exist formalisms, such as Turing machines, which have even more expressive power than finite state machines. One might ask whether formal power is the ultimate criterion for deciding between representations. We state that for expert system applications *expert comprehensibility* is more pertinent to the choice of an appropriate representation than *formal power*.

3

The Mugol environment†

Abstract. The Mugol environment implements a hierarchically ordered system of inductive learning for the acquisition of expert knowledge. The inductive component is a derivative of ID3, an algorithm which performs static induction. The environment allows the development of both classification and control oriented expert systems. A powerful facility is provided for interfacing rules to other knowledge sources.

3.1. Some issues in knowledge engineering

Expert systems differ from other computer programs in the following aspects.

1) **Explanation.** Expert systems are capable of inspecting their own reasoning in order to explain why certain factors are being investigated and how particular conclusions are reached.

2) **Problem type.** Expert systems are more suitable than an algorithmic approach for problems which involve a large amount of branching.

3) **Partial certainty.** Expert systems are usually able to deal with a set of values between true and false which represent the partial certainty of a proposition. The Mugol environment described in this chapter does not provide facilities to deal with partial certainty.

It has been stated (Feigenbaum, 1979) that the most difficult part of expert system building lies in the acquisition of knowledge from experts. A key issue here is the difference between dialogue acquisition of rules and the use of inductive learning. The latter approach relieves experts of much of the burden of authoring rules directly, allowing them to present merely instances of

† The original authors of the paper "RuleMaster: a second-generation knowledge-engineering facility" (Michie, Muggleton, Riese and Zubrick, 1984) have kindly consented to the inclusion in this chapter of material relevant to the thesis author's contribution.

correct decision-making, while the machine produces generalisations of these decisions.

One of the goals of AI programming language design (as pursued, though not yet fully attained, by the logic programming school) is that users should be able to tell the machine relevant facts, theories, advice etc. in any order that occurs to them rather than in some fixed sequence as demanded by traditional programming languages. A degree of order-independence which has so far eluded every Prolog interpreter can be supplied by a new style of programming based on inductive rather than deductive mechanisms. Strictly speaking, inductive programming involves the user in creating operational specifications which can be transformed by the inductive mechanism into well sequenced, efficient programs.

Large knowledge bases can become unwieldy and difficult to understand. Experts tend to organise their knowledge as a set of interrelated factors. By making a hierarchy of attributes explicit (Shapiro, 1983) it is possible to make the interrelationship of problem attributes easier to understand and maintain. Furthermore, Shapiro showed that structuring even paid off in terms of strict store-cost and processor cost (see also section 2.5.3). Even though this approach involves the human overhead of structure formation, its advantages outweigh this disadvantage.

As explained in section 1.1, expert systems can be broadly divided into two main categories: classification, eg. MYCIN (Shortliffe and Buchanan, 1975) and control, eg. VM (Fagan et al, 1979). Often expert systems require a combination of these abilities. In many systems the form of knowledge representation supports one of these approaches while impeding the other.

Practical expert systems require information sources other than the rules which the expert uses for decision making. For instance, a medical diagnostic system might read biomedical sensors, access patient records and do mathematical modelling of bodily processes. Facilities for linking to external routines may therefore be considered as an essential component of modern expert system software.

3.2. The Mugol environment

3.2.1. Overview

The Mugol environment is a general purpose expert system building tool. It consists of two major components: Mugol and Mugmaker. Mugmaker is an inductive generator of executable expressions in a rule language called Mugol. Mugmaker allows users to describe their knowledge in a declarative form, while Mugol executes the more procedurally oriented form generated by Mugmaker. Mugmaker is discussed in the knowledge acquisition section below, while some of the special features of Mugol are described in further sections.

It is a well recognised fact that over 50% of the time taken creating an expert system is spent on building support facilities for carrying out calculations, reading instrumentation or accessing databases. As a high-level language Mugol combines both an ability to represent conditionals rules and to carry out calculations and communicate with procedures written in other languages (see section 3.10). While other high-languages might have provided similar facilities in this respect, Mugol is unique in its ability to provide explanation of reasoning as an integral part of the program specification.

The development of the Mugol environment was motivated by a desire to solve the knowledge engineering issues involved in building large expert systems, (described in the section 3.1). The original basis for the inductive techniques used in Mugmaker are those of Quinlan's (Quinlan, 1979) ID3 algorithm. This method was later refined and improved by Shapiro and Niblett (Shapiro and Niblett, 1982) and Paterson and Niblett (Paterson and Niblett, 1982). A basis for structuring inductively generated rule sets was developed and tested by Shapiro and Niblett (Shapiro and Niblett, 1982). The explanation facility used by the Mugol interpreter is derived from a proposal by Shapiro and Michie (Shapiro and Michie, 1986). The Mugol environment is the academic counterpart of the commercial product RuleMaster (Michie, Muggleton, Riese and

Zubrick, 1984).

The Mugol environment is written as a set of interrelated C programs written under the UNIX* operating system. There are current working versions running on the following machines: DEC VAX, SUN Microsystems, IBM PC/XT and PC/AT, and AT&T UNIX machine.

3.3. Knowledge acquisition

The approach taken in the Mugol environment differs considerably from that of hand-crafting rules. It is well known that experts explain complex concepts to human apprentices implicitly by way of examples rather than explicitly by stating principles. The apprentice intuitively generalises these sample decisions to form more widely applicable rules. A computer can learn in the same way as the human apprentice if it is able to produce general rules from specific instances.

The Mugol environment allows the expert system builder to use rules authored either explicitly by an expert or by the machine from examples. The machine builds rules by a process called *rule induction*. In induction of classification rules, rules are induced by generalisation over *examples* of expert decision-making. An example is expressed as a vector of values pertaining to attributes of the decision, together with the expert's classification. For instance, in a very simple case, if we are trying to build a rule to classify animals, the attributes of the decision might be *colour* and *size*. A possible classification is *ELEPHANT*. Given the example:

Colour	*Size*		*Class*
grey	big	=>	ELEPHANT

The induction algorithm would generalise this example to the rule:

irrespective of the animal's colour or size, it is an ELEPHANT

* UNIX is a trademark of Bell Laboratories.

In order to get a more accurate generalisation, more examples would need to be added, and a more complex rule would be induced. For instance, with the following example set:

Colour	*Size*		*Class*
grey	big	=>	ELEPHANT
yellow	big	=>	GIRAFFE
grey	small	=>	TORTOISE

the following decision tree is generated:

If the animal's colour is

a) yellow, then it is a GIRAFFE

b) grey, then if the animal's size is

i) big, then it is an ELEPHANT

ii) small, then it is a TORTOISE

In the Mugol environment, a class is composed of an action and a next-state. The action specifies what to do in the example situation, and the next-state says which context must be entered after the action has been carried out. The induction subsystem which supports this is known as Mugmaker. The syntax and semantics of Mugmaker are described in section 3.7.

An illustration of the power of inductive inference to weed out irrelevance is shown in the following example taken from the WILLARD expert system (see Appendix A). Contained within a widely used meteorological manual (NTIS, 1969) is a table (Figure 3.1) of three attributes used to determine the expected change of lapse rate.

Vertical Divergence	Vertical Motion	Vertical Thickness		Change in Lapse Rate
divergent	descending	shrinking	=>	more stable
divergent	ascending	shrinking	=>	more stable
divergent	ascending	stretching	=>	less stable
convergent	descending	shrinking	=>	more stable
convergent	descending	stretching	=>	less stable
convergent	ascending	stretching	=>	less stable
divergent	none	shrinking	=>	more stable
convergent	none	stretching	=>	less stable
divergent	ascending	no change	=>	no change
convergent	descending	no change	=>	no change
none	ascending	stretching	=>	less stable
none	descending	shrinking	=>	more stable

Figure 3.1 Table found in (NTIS, 1969) of examples used in lapse rate determination

The rule generated from these examples used only one of the three given attributes. This rule was as follows:

if the thickness (distance between the two constant pressure surfaces)

a) shrinks then the lapse rate becomes more stable

b) stretches then the lapse rate becomes less stable

c) does not change then the lapse does not change

This simple relation was never spotted by meteorologists although the table had appeared for years in standard texts. The rule was found to be correct and consistent with a physical model of the atmosphere (based on the hypsometric equations).

Entering rules by examples has several distinct advantages over writing production rules. When the example set has insufficient information to cover the entire problem space, Mugmaker will generalise these examples in order to produce a decision tree which covers all the possible

situations. If the knowledge is entered directly as rules no generalisation is carried out. When too many attributes are present (as in the case shown in figure 3.1), redundant information is ignored by Mugmaker. Again, production systems do not have this ability to compact knowledge.

The knowledge is given as examples in a more implicit declarative form than production systems, and this is automatically transformed into a more explicit procedural form than that of production systems. Thus experts can enter and revise knowledge without regard to order, while reviewing and executing an economical procedural form constructed for them by the system. In logic programming the equivalent of Mugmaker's example set would be an arbitrarily ordered set of Horn clauses in a propositional subset of first order logic.

3.4. Types of expert systems supported

A wide range of approaches may be taken in the construction of expert systems. These approaches employ varying knowledge representations, including production systems (eg. Shortliffe and Buchanan, 1975), first-order predicate logic (eg. Niblett, 1985), frames (Minsky, 1975), inference networks (eg. Duda et al., 1979), causal models (eg. Mozetic, Bratko and Lavrac, 1984), object-oriented models (eg. Bobrow and Stefik, 1983) or hybrid approaches (eg. Intellicorp, 1984). The best understood of these are classificatory in nature. However, expert system packages made by removing the knowledge component from an expert classification system, as was done with EMYCIN (Van Melle, 1980), often have difficulty handling procedural actions. The Mugol environment was designed from the start to allow a wide range of control strategies, so that the system could be applied to a broad set of problem types.

An expert system application built with the Mugol environment consists of a set of Mugol modules. Each module consists of a transition network of states, each of which contains a single decision tree. When invoked, each decision tree carries out a sequence of tests until a decision is reached to perform an action. After execution of this action, control is passed to a new state within the calling module. Control only moves from one module to another via the mechanism of procedure call. The ability to do conditional branching together with that of calling modules

recursively allows the building of arbitrarily complex control structures (see sections 2.6.6 and 2.6.7 for a discussion of the expressivity of the Mugol language). More details concerning the Mugol language are given in section 3.5.

The two large expert systems which have been built with the Mugol environment are primarily classificatory and only use a subset of the Mugol language (see Appendices A and B). However, small but effective control expert systems have been built. For example, an expert system which builds an arch out of a set of blocks from an arbitrary starting position to a given goal position was inductively constructed from sample arch-building action-decisions (see chapter 4). Although this was only a demonstration of the Mugol environment's capabilities, the domain contains several of the features of non-classificatory applications (e.g. simple design and scheduling).

3.5. The Mugol language

Mugol is a language for the run-time orchestration of an incremental library of C-coded procedures. It accordingly leaves all actual computations to these, including I/O. One can say that all that a Mugol program ever does is to execute rules/control statements and assign strings to variables.

3.5.1. Finite automata

The language has its formal basis in finite automata theory (see section 2.6). The subclass of finite state automata which are of interest here, DUTMMs (see section 2.6 and 7.3), consists of machines whose output signal is entirely dependent on the combination of their input signals and their internal machine state.

More formally, the behaviour of the machine is described by the 6-tuple $\{Q, \Sigma, \Delta, \delta, q_0, q_g\}$ in which

(1) $Q = \{q_0, q_1, ...\}$ is a set of machine *states*.

(2) Σ is a set of legal *inputs*.

(3) Δ is a set of legal *outputs*.

(4) δ is the *next state function* which gives the next state based on the current state and the current situation (in the case of a machine measuring the situation on the basis of two binary values we have $\delta : Q \times \{0, 1\} \times \{0, 1\} \rightarrow Q$)

(5) q_0 is the *start state* of the automaton.

(6) q_g is the unique *goal state* of the automaton.

δ can be described by a tabulation of inputs and outputs related to particular states. This is usually called the *state transition table.* However, a finite state function is often more clearly represented in terms of a *state transition diagram.* Figure 3.2 depicts a finite state diagram of an automaton capable of taking two streams of binary input digits and producing one stream representing their sum using the states q_0 and q_1 to represent the "carry." The initial state is indicated by a start arrow, and the state transition arcs are labelled with the two input digits read-in, together with the one output digit produced. At any moment in time, the machine's situation is the pair of values being read in, eg. <0,1>. The action chosen will either be to output "0" or to output "1".

By analogy with the above, we could imagine a sequential procedure being represented in the style of figure 3.3, with conditional tests replacing the input symbols, and procedure calls replacing

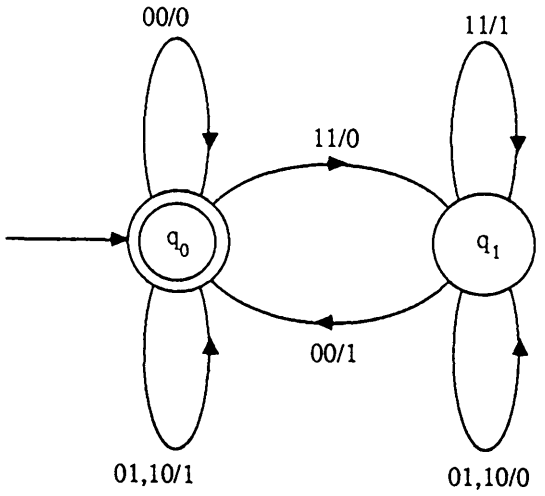


Figure 3.2 A binary adder as a finite state machine transition diagram.

the output symbols. The figure is a diagrammatic representation of a possible Mugol program module. The module while in state "in bed" decides to do the action "sleep" and stay in state "in bed" if conditions "tired" and "not-hungry" are true. Alternatively, if tired is untrue or hungry is true, then action "get up" is carried out, and the module enters state "up." It is left to the reader to follow the remaining transitions.

An approximate correspondence between this formalism and the production-rule formalisms more familiar to knowledge engineers is shown in the table of figure 3.4. We shall follow the EMYCIN use of the term "context" to denote a self-contained bundle of rules which can be entered from another context as a result of firing an action which contains a "goto".

In terms of algorithmic programming (an inductively generated application can be viewed as an efficient block-structured program) we have the correspondence of figure 3.5.

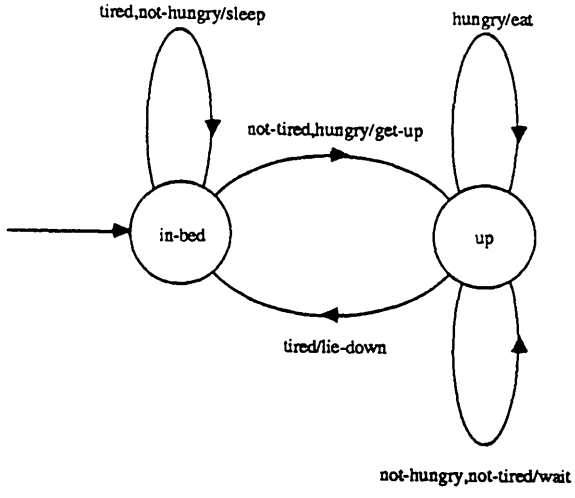


Figure 3.3 A simple daily routine. ' , ' means 'or'

Finite state automata	Production rule systems
state	context
situation	antecedent
action	consequent
transition arc	goto <context>

Figure 3.4 Relationship between Finite state automata and Production rules

Mugol	Algorithmic
module	routine (may or may not return value)
state	labelled if-then-else block
situation	set of callable value-returning routines
action	non-value-returning routine
transition arc	goto <label>

Figure 3.5 Relationship between Mugol and Algorithmic languages

3.5.2. Mugol syntax

Figure 3.6 shows the syntax of Mugol as a syntax diagram. The following sections use illustrative examples to describe the structure of Mugol programs.

3.5.3. Mugol program structure

A Mugol program consists of a collection of inter-related modules. An individual module can represent either an executable procedure or a piece of data. In order to aid the imposition of a structure on these modules, they are arranged in a tree. The scope of referencing a module from any other is limited by a recursive scope rule called *visibility*. Visibility is defined (figure 3.7) as follows:

```
module m2 is visible to module m1 iff
        m2 is a child of m1 or
        m2 is visible to the parent of m1
```

Figure 3.7 represents a hypothetical program tree with modules named by the letters of the alphabet. Note that the highest module in the tree, root, by the definition of visibility cannot be referenced by any module in the tree. The circular modules are those visible to module "m". Rather than using block bracketing to indicate the tree structure of a Mugol program, each module is identified using its unique path from the root in the program tree, e.g. module m's complete name is "b.g.m."

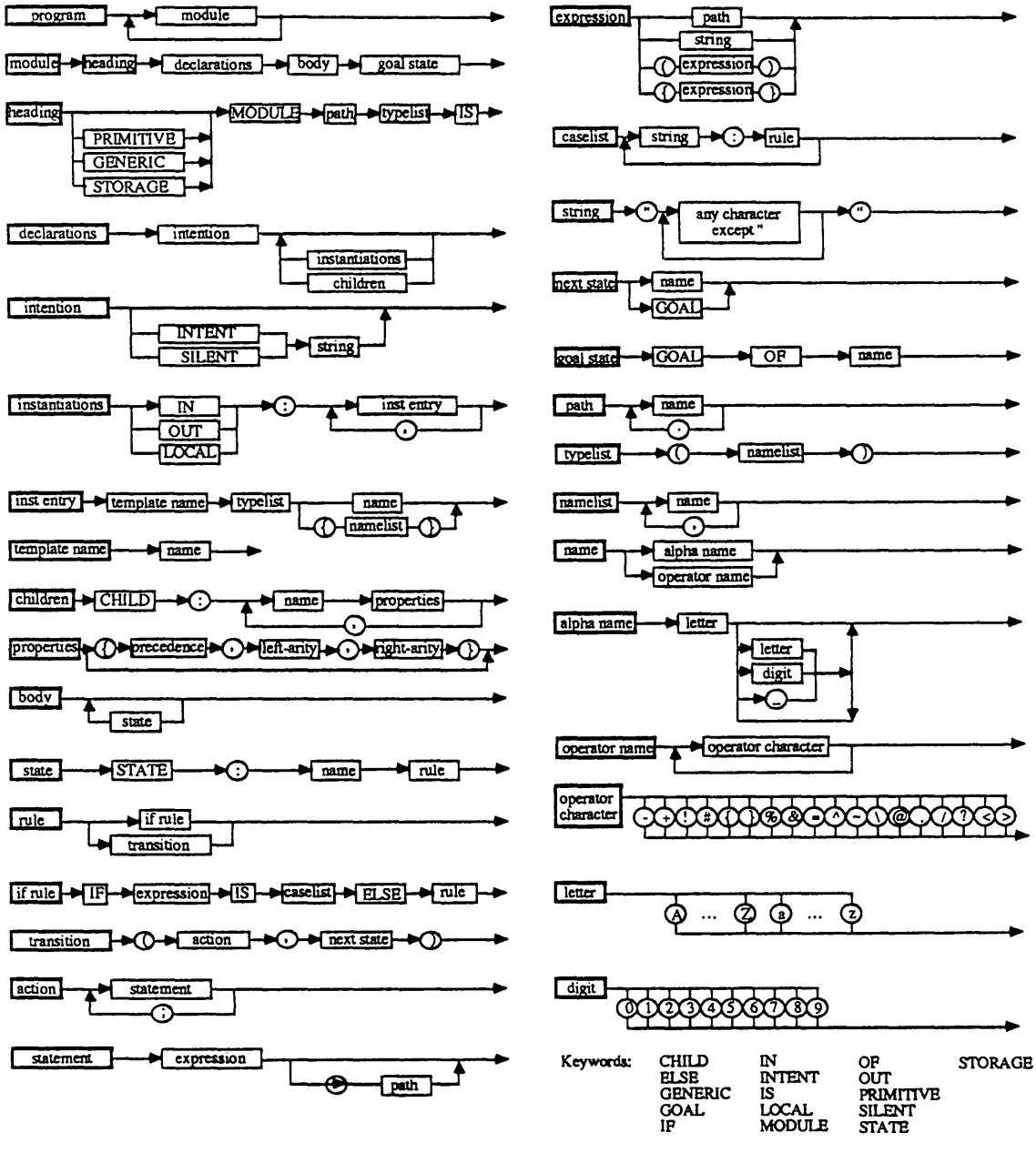


Figure 3.6 The syntax of Mugol

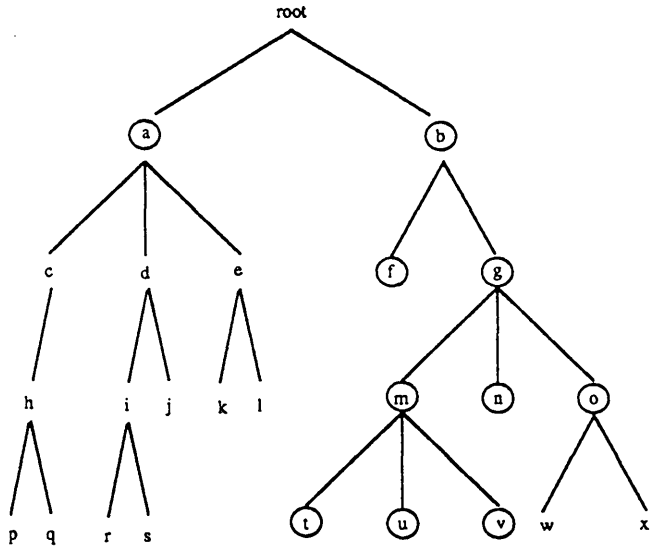


Figure 3.7 The visibility of module m

3.5.4. Form of individual states

Each Mugol module consists of a declaration section together with a number of named states. Each state in a module has a single decision tree associated with it which decides, on the basis of a number of tests, what action should be taken, and which state within the present module to enter subsequently. The decision trees have tests at internal nodes and action/next state pairs at the leaves.

The state shown in figure 3.8 is from a module to decide whether to use an umbrella.

```
STATE: decide
IF (weather) IS
      "wet" : IF (inside) IS
            "yes" : ("DONTUSE" → result, goal)
            ELSE IF (soaked) IS
                  "yes": ("DONTUSE" → result, goal)
                  ELSE ("USE" → result, goal)
        "sunny" : ("DONTUSE" → result, goal)
        ELSE ("DONTUSE" → result, goal)
```

Figure 3.8 Decision tree within state which decides whether to use an umbrella

The decision tree is expressed as nested "case statements" each of form

IF <test> IS

<val 1> : <tree 1>

<val 2> : <tree 2>

...

ELSE <tree N>

where each <tree x> is either of the same form or of the form

(<action>, <next-state>)

In figure 3.8:

(1) weather, inside and soaked are tests producing the quoted string values shown (e.g., weather can be "wet," "sunny" or some other value).

(2) '"USE" → result' and '"DONTUSE" → result' are actions indicating the assignment of the string literals "USE" and "DONTUSE" to the variable 'result'.

(3) goal is the name of a special state which is found in every Mugol module. When entered, it merely returns control to the calling module.

3.6. Individual Mugol modules

A Mugol module can be labelled with certains combination of:

(1) **PRIMITIVE.**

(2) **GENERIC.**

(3) **STORAGE.**

Figure 3.9 tabulates the meaning of the 8 different combinations of these three labels. These various module types are explained in the following sections.

3.6.1. Type 0 modules - normal Mugol modules

There are two basic classes of type 0 module: modules which return a value to the calling module and modules which do not. When used in expert system work, this distinction is between modules which carry out some diagnostic value-returning test that does not affect the state of the world and modules which carry out a non-value-returning control action which is intended to affect

Type	Module labelling	Meaning
0	MODULE	A normal callable procedure
1	PRIMITIVE MODULE	Callable C-coded procedure
2	GENERIC MODULE	A module which can be instantiated for operation on different types
3	PRIMITIVE GENERIC MODULE	Unused combination
4	STORAGE MODULE	Unused combination
5	PRIMITIVE STORAGE MODULE	Unused combination
6	GENERIC STORAGE MODULE	User-defined abstract data type
7	PRIMITIVE GENERIC STORAGE MODULE	System-defined abstract data type

Figure 3.9 The interpretation of the 8 different types of Mugol module

the world (see section 1.1 for distinction between classification and control). Figure 3.10 shows an example of the former type of module (value-returning) while figure 3.11 shows an example of the latter (non-value-returning). Figure 3.10 is the complete Mugol module from which the state shown in figure 3.8 was excerpted. The module *rain* exists in the program tree as a child of *main* (hence

```
MODULE main.rain IS
   INTENT: "decide whether to use an umbrella"
   CHILD: weather, inside, soaked
   OUT: string result
   STATE: decide
   IF (weather) IS
        "wet" : IF (inside) IS
           "yes" : ("DONTUSE" → result, GOAL)
           ELSE IF (soaked) IS
                "yes": ("DONTUSE" → result, GOAL)
                ELSE ("USE" → result, GOAL)
        "sunny" : ("DONTUSE" → result, GOAL)
        ELSE ("DONTUSE" → result, GOAL)
GOAL OF rain
```

Figure 3.10 A value-returning Mugol module

```
MODULE oax IS
   INTENT: "play noughts and crosses"
   STATE: decide
    IF (ask "is board full" "yes,no") IS
      "yes" : (advise "Board full - end of game", GOAL)
      ELSE IF (ask "can O win immediately" "yes,no") IS
         "yes" : (advise "complete line to win", GOAL)
         ELSE IF (ask "can X win immediately" "yes,no") IS
            "yes" : (advise "block X", decide)
            ELSE IF (ask "is middle free" "yes,no") IS
               "yes" : (advise "take centre", decide)
               ELSE IF (ask "Is there a corner free" "yes,no" ) IS
                  "yes" : (advise "take the corner with most WEIGHT", decide)
                  ELSE (advise "take any free space", decide)
GOAL OF oax
```

Figure 3.11 A non-value-returning Mugol module which plays noughts-and-crosses

the path name is *main.rain*). The system will use the string following the keyword *INTENT* when referring in explanation to the execution of this module. *rain* is a single-state module the state being called 'decide'. Examples of modules containing multiple states are given in section 3.9. Module *rain* has five sub-modules, *weather*, *inside*, *soaked*, *incar* and *result*. *result* is an instantiation of the primitive generic storage module string and is used as the output variable of *rain*.

The module *oax* of figure 3.11 plays the noughts' side of the game noughts-and-crosses by asking a number of questions about the board state, such as "is board full". On the basis of these questions it advises an action, such as "take any free space". Paired with this action is the next state which can either be the present state, *decide*, which causes looping, or the special module exit state, *GOAL*, which causes termination of play. Note that the major difference between the module shown in figures 3.10 and 3.11 is the respective presence and absence of the OUT declarations in these modules.

3.6.2. Type 1 modules - C-coded procedures

Mugol programs can call programs written in other languages (presently C and FORTRAN) by the mechanism of PRIMITIVE MODULES. Figure 3.12 shows the interface to the primitive module *ask* which is called by the noughts-and-crosses playing module of figure 3.11. Note that the explanation string for *ask* is labelled with the keyword *SILENT* here rather than *INTENT* since we do not want any of the workings of *ask* to be shown in any explanation of execution (see section 3.8). The input prompt string (*IN: string prompt*) is substituted for the $1 in the explanation string. Any occurrence of a '$' followed by some number *N* causes the runtime substitution of the *N*th input argument into the explanation string when an explanation of execution is given.

The Mugol interpreter executes PRIMITIVE modules either by mapping each call via a look-up table to a unique C-function compiled into the interpreter, or failing this, by requesting "remote" execution of the procedure in a "slave" process which runs as a concurrent process connected by a UNIX pipe to the Mugol interpreter.

3.6.3. Type 2 modules - generic modules

Generic modules are code segments which can be multiply instantiated to act on different data types. For instance the module *square* of figure 3.13 can be multiply instantiated as in modules *sqri* and *sqrf*. to operate on either integers or floating point numbers. The interpreter automatically

---

```
PRIMITIVE MODULE ask IS
   SILENT: "the answer to '$1'"
   IN: string prompt
   OUT: string answer
GOAL OF ask
```

Figure 3.12 The primitive module 'ask'

---

infers the appropriate meaning for "*" in the instantiated modules from the types of its operands.

3.6.4. Type 6 - user-defined abstract data types

When an expert describes his problem, he would prefer to state it using terminology pertinent to his own subject. Thus a thrust force of 5000 N is to a turbo engineer more than the integer value "5000". Abstract data types allow the expert to invent his own data types together with operators for manipulating them. Users can define their own abstract data types within a Mugol program by the declaration of GENERIC STORAGE MODULEs. The direct children of a generic storage module are taken as being operators which act only on data of the corresponding type. Thus figure 3.14 describes a new type of object called a "coordinate" which consists of three values, *x*, *y* and *z*. The operations which can be carried out on a coordinate are "read" (which reads a coordinate value from the user), "print" (which prints a coordinate value onto the screen) and "offset" (which adds a 2D offset to a coordinate position). Note that since this is a storage module, it has no executable body.

```
GENERIC MODULE square (type) IS
   IN: type val_in
   OUT: type val_out
   STATE: calculate
        (val_in * val_in -> val_out, GOAL)
GOAL OF square

MODULE m IS
   LOCAL: square (integer) sqri, square (float) sqrf
   STATE: show
        (print sqri 5 ; print sqrf 5.0, GOAL)
GOAL OF m
```

Figure 3.13 Generic modules and instantiations

```
GENERIC STORAGE MODULE coordinate IS
   LOCAL: float {x,y,z}
   CHILD: read, print
GOAL OF coordinate

MODULE m IS
   LOCAL: coordinate {top,bottom}
   STATE: getshow
        (read "Bottom? " -> bottom; read "Top? " -> top; print bottom, GOAL)
GOAL OF m
```

Figure 3.14 The abstract data type 'coordinate'

3.6.5. Type 7 - system-defined abstract data types

The Mugol language has no inherent data types built into its syntax, all variables being stored internally as strings for the convenience of inter-process communication. However, certain data types are supplied to the interpreter along with any application. Such base types are declared as PRIMITIVE GENERIC STORAGE MODULEs. In all other ways primitive generic storage modules are identical to the GENERIC STORAGE MODULEs of section 3.6.4. The Mugol interpreter at present supports the following data types

string

integer

float

list

3.7. Operator definitions

In several languages, including Prolog, programmers are provided with the ability to declare procedures (or predicates in Prolog's case) in such a way that they can be called using infix, prefix or postfix notation within expressions. Such a facility is also provided in the Mugol language by allowing the declaration of "operator properties" together with the definition of modules. Figure 3.15 illustrates the use of such operator properties in the definition of the integer data type operators. In the definition *+ {20,1,1}* of figure 3.15, the three numbers stand respectively for *precedence*, *left arity* and *right arity*. The first of these, precedence, indicates +'s binding strength within an expression. Imagine an expression depicted in the standard form as a tree, hanging downwards from the root. The lower the precedence, the lower the operator will be in the tree. Left and right arity indicate the number of arguments expected on the left and right side of the operator. Clearly the sum of the left and right arities should be equal to the number of input

```
PRIMITIVE GENERIC STORAGE MODULE integer IS
        child:  + {20,1,1},
                - {20,1,1},
                * {30,1,1},
                ** {50,1,1},
                / {30,1,1},
                < {15,1,1},
                <= {15,1,1},
                > {15,1,1},
                >= {15,1,1},
                == {10,1,1},
                != {10,1,1},
                i_to_s,
                s_to_i,
                real,
                read,
                print {8,0,1}
GOAL OF integer
```

Figure 3.15 Integer data-type operators

arguments expected by the corresponding procedure. Note that in figure 3.15 the operators *i_to_s*, *s_to_i*, *real* and *read* have no declared operator properties. In such a case the interpreter assigns a default precedence of 100, a left arity of 0 and a right arity equal to the number of input arguments. Operators have a maximum precedence of 511.

3.8. Explanation

The Mugol interpreter has the ability to explain its line of reasoning at any time during a session. When an expert system is consulted, the reasoning behind a piece of advice may influence its acceptance. For example, if the explanation indicates that a critical factor has been ignored, the user may decide to reject the advice. Requests for explanation during hypothetical test cases can also be used to instruct novices. Explanation has additional value at development time. In explanation-driven development the expert checks that correct decisions are reached, and that they are reached for the right reasons. This increases the likelihood that situations outside the training set will be dealt with correctly. A point in case occurred during the validation of the EARL expert system (see Appendix B), in which it was discovered that in 4 cases out of 859 tested EARL reached the correct conclusion for the wrong reasons. Without the ability to validate both the result and the explanation against the expert, this would never have been realised.

Our implementation of explanation follows that described in Shapiro and Michie (Shapiro and Michie, 1986), though we have not incorporated the irrelevance-suppression option which Shapiro and Michie regarded as important; instead we present the explanation in rule-sized chunks so as not to overburden the user.

Each Mugol module requires a text segment containing optional slots for run-time substitution of the input arguments of that module (such as the *INTENT* statement of figure 3.10 and the *SILENT* statement of figure 3.12). These pieces of text are combined in a standard set of masks to produce English phrases. An algorithm orders the individual phrases to form a proof which justifies the reasoning by building from axiomatic facts towards the final conclusion. The explanation is given piecemeal, the most relevant portion being presented first, with further

elaboration on demand. Actions and tests are dealt with differently by the presentation algorithm. Users can request explanation at any time that they are asked a question or given advice. In addition the users can interrupt the system at any time to find out what is happening. Furthermore, a full report of the line of reasoning leading to some final conclusion can be produced at the end of a session. Although the ordering is revised, this form of explanation is similar to that given by MYCIN.

An example of automatically generated explanation is given in figure 3.16 (this figure is identical to figure A.2 in Appendix A and has been reprinted for the reader's convenience).

It should be noted that the bracketed numbers in figure 3.16 are not produced by the present version of the Mugol environment, but are included in the diagram for ease of reading. This would be a simple and desirable addition to the system's capabilities.

The following example illustrates the mechanism by which explanation is ordered. The Mugol interpreter saves a full trace of a program's execution as a proof tree. When an explanation is demanded, the proof tree is presented in postfix order with keywords (such as *Since*) being inserted. Thus if in figure 3.17 the tree represents the execution proof tree, with *A* being proved by the conjunction of *B* and *C* being true etc., the explanation would be presented bottom-up, as shown in the same diagram. This explanation shows how axioms are presented before lemmas, which are in turn presented before the final theorem being proved. This ordering seems very close to that used by human beings in explaining logical proofs.

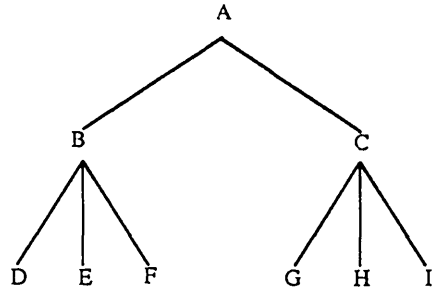
3.9. The Mugmaker code generator

Experts using the Mugol environment have the freedom to build their expert system without ever having to write actual Mugol code, generating it instead from example decisions. For this purpose Mugmaker uses the ACLS algorithm (Paterson and Niblett, 1982) to generate a single rule for each state of the module. Mugmaker distinguishes between "test" and "action" modules, in that whereas the former returns a value, the latter does not.

FULL EXPLANATION OF THE FORECAST:

Since upper level cold air advection causing increased
upwards vertical velocities is present
it follows that the upper-level destabilisation
potential is sufficient (1)
Since the K Index is strong
when the Lifted Index is strong
it follows that the stability indices condition
is favourable (2)
Since daytime heating acting as a possible trigger mechanism
for potential instability release is strong
when (2) the stability indices condition is favourable
it follows that low-level destabilisation potential
is favourable (3)
Since an approaching 500 millibar short wave trough is present
it follows that the vertical velocity field
is favourable (4)
Since a high 850 mb dew point is present
when surface dew point classification is moderate
it follows that the low-level moisture field
is marginal (5)
Since (1) the upper-level destabilisation potential is sufficient
when (3) low-level destabilisation potential is favourable
and (4) the vertical velocity field is favourable
and (5) the low-level moisture field is marginal
it was necessary to advise:
"There's a MODERATE CHANCE that thunderstorms occurring
12 hours from now will be severe at this location."
in order to actually forecast the chance of severe thunderstorms

Figure 3.16 Sample WILLARD forecast explanation



gives:

```
Since D
   when E
   and F
   it follows that B
Since G
   when H
   and I
   it follows that C
Since B
   when C
   it follows that A
```

Figure 3.17 Method of ordering explanation.

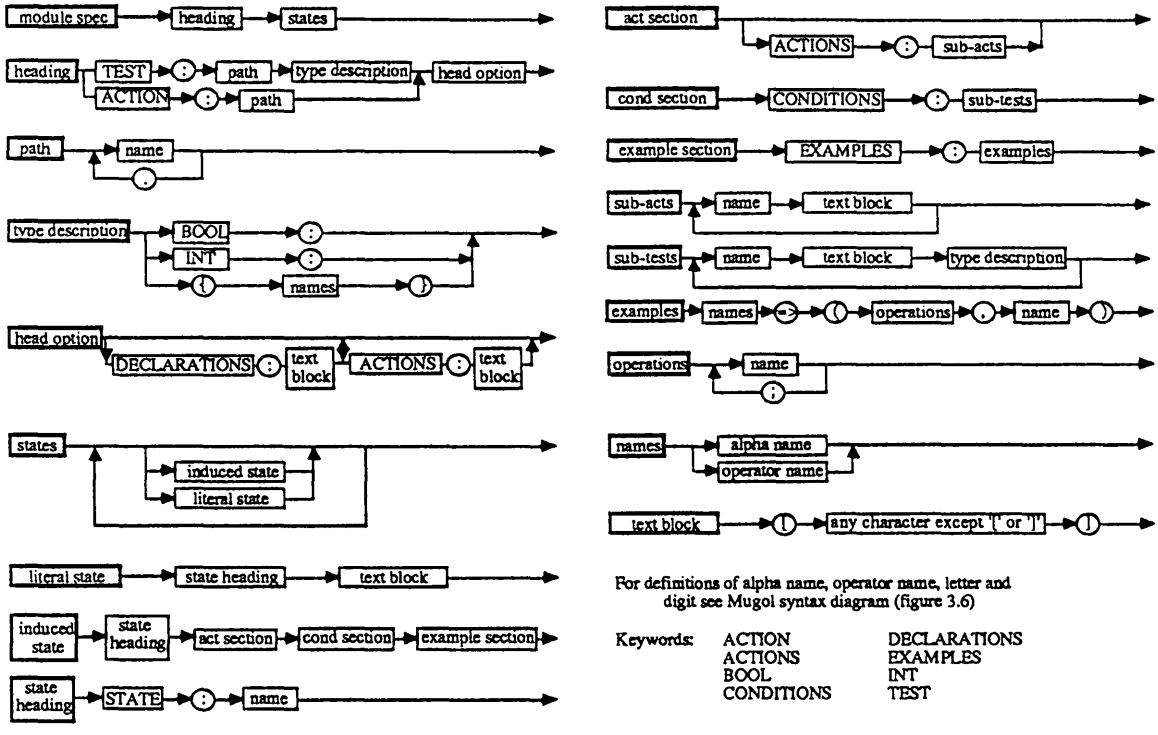


Figure 3.18 The syntax of Mugmaker

```
TEST: main.rain { USE DONTUSE }

DECLARATIONS:

        [ CHILD: weather, inside, soaked, incar
          OUT:   string result                          ]

STATE: decide

        ACTIONS:
                DONTUSE             ["DONTUSE" -> result]
                USE                 ["USE" -> result]

        CONDITIONS:

        weather                     [weather]     {wet sunny blustery}
                inside              [inside]      {yes no}
                        soaked      [soaked]      {yes no}
                                incar [incar]     {yes no}

        EXAMPLES:

        sunny    -     -     -     =>   (DONTUSE, goal)
        blustery -     -     -     =>   (DONTUSE, goal)
        wet      yes   -     -     =>   (DONTUSE, goal)
        wet      no    yes   no    =>   (DONTUSE, goal)
        wet      no    no    yes   =>   (DONTUSE, goal)
        wet      no    no    no    =>   (USE, goal)
```

Figure 3.19 Mugmaker induction file for Mugol module of figure 3.10

3.9.1. Mugmaker syntax

For every Mugol module there exists a Mugmaker file which was used to generate that module. Figure 3.18 shows the syntax of Mugmaker files. In the following sections we use illustrative examples to describe the structure of these files.

3.9.2. Single-state module

The Mugol module of figure 3.10 was generated automatically from the induction file of figure 3.19. The module has only one state where the action is to return a string giving the

decision. Below we give an informal description of the keywords and major sections of this Mugmaker file.

(a) **TEST:** is followed by the path of the Mugol module to be induced, "main.rain". Following this are return value options. In this case the return value is a string with two possible values: "USE" and "DONTUSE".

(b) **DECLARATIONS:** introduces a piece of text (between square brackets) to be placed at the top of the module produced. The declarations indicate the modules associated with "rain".

(c) **STATE:** is followed by the name of a state in the object module. The actions, conditions, and examples define the rule for that state.

(d) **ACTIONS:** indicates a set of action-names found in the implication of example clauses. Each action name is followed by a piece of Mugol code in brackets, which is substituted for the action name in the object program. In the umbrella example the actions are simple assignments of strings to the output argument "result."

(e) **CONDITIONS:** are a set of value producing expressions. Like actions, each condition name is followed by a piece of Mugol substitution code in square brackets. Following this in curly brackets is the set of values that can be produced by this conditional expression. The conditions have a one-to-one correspondence with the example value vectors. For this reason, the condition descriptions are stepped in order to line up with the columns of the example set.

(f) **EXAMPLES:** are vectors of values, one value from each condition expression, each followed by an (=>) and an (action, next state) pair. These are used to induce the decision tree for this state. A "-" in an attribute column is called a *don't care* value. *Don't care* values are interpreted as representing all possible values which the corresponding attribute could take.

Thus an example containing one *don't care* value for an attribute with N possible values, actually represents N distinct examples. An examples with more than one *don't care* value represents a number of examples equal to the product of all the numbers of attribute values for which the *don't care* values are present.

3.9.3. Multiple-state module

The next example file (Figure 3.20) defines a routine with two states. The rules instruct an individual on the "Heimlich method" actions to be taken in case of an adult choking victim who is standing up. The conditions to be checked are the victim's airway, consciousness, pulse, and whether the patient is breathing.

One new feature seen here is compound Mugol statements, joined by ";", as the action implicated by examples. The statements may be expressions or assignments. Mugol modules named in expressions will often be primitives written in C. For example, "prints" and "reads" are C utility routines for printing and reading strings. Each application will typically add its own set of domain-dependent utility routines.

```
MODULE: main.choke

        DECLARATIONS:

                [ CHILD: ar ]
        ACTIONS: /* global actions */
                hit         [prints "hit victim's back 4 times\n"]
                sweep       [prints "sweep victim's mouth with finger\n"]
                ok          [prints "comfort the victim\n"]

        STATE: primary

                ACTIONS:
                        squeeze     [prints "squeeze victim's chest 4 times\n"]
                        brace       [prints "brace victim to prevent falling\n"]
                        amb         [prints "call an ambulance\n"]

                CONDITIONS:

                freed                           [reads "Is the obstruction clear?"]
                                                        {yes no}
                        falling                 [read "Is the victim falling? "]
                                                        {yes no}
                                conscious [reads "Is the victim conscious? "]
                                                        {yes no}

                EXAMPLES:

                yes     yes     yes     => (brace; ok, GOAL)
                no      no      yes     => (hit; squeeze; sweep, primary)
                no      yes     yes     => (brace; hit; squeeze; sweep, primary)
                -       -       no      => (amb, unconscious)
                yes     no      yes     => (ok, GOAL)

        STATE: unconscious

                ACTIONS:
                        thrust      [prints "apply 4 chest thrusts\n"]
                        ar          [ar]        /* Artificial respiration */
                        cpr         [prints "apply cpr: 15 compressions, 2 breaths\n"]
                        emt         [prints "let EMT staff take over\n"]
                        ok          [prints "You just helped save a life\n"]

                CONDITIONS:

freed                                           [reads "Is the obstruction clear?"]
                                                        {yes no}
        breathing                       [read "Is the victim breathing? "]
                                                        {yes no}
                pulse                           [reads "Is there a pulse? "]
                                                        {yes no}
                        conscious       [reads "Is the victim conscious? "]
                                                        {yes no}
                                amb             [reads "Has the ambulance arrived? "]
                                                        {yes no}

                EXAMPLES:

no      yes     yes     no      no      => (hit; thrust; sweep, unconscious)
no      no      yes     no      no      => (ar, unconscious)
yes     no      yes     no      no      => (ar, unconscious)
yes     no      no      no      no      => (cpr, unconscious)
no      no      no      no      no      => (hit; thrust; sweep; cpr, unconscious)
yes     yes     yes     yes     no      => (ok, primary)
-       -       -       -       yes     => (emt, GOAL)
```

Figure 3.20 Induction file for the choke problem

```
MODULE main.choke IS
 CHILD: ar
  STATE: primary
    IF (reads "Is the victim conscious? ") IS
     "yes" : IF (reads "Is the obstruction clear? ") IS
         "yes" : IF (reads "Is the victim falling? ") IS
          "yes" : (prints "brace victim to prevent falling\n";
                          prints "comfort the victim\n", GOAL)
           ELSE (prints "comfort the victim\n", GOAL)
        ELSE IF (reads "Is the victim falling? ") IS
          "yes" : (prints "brace victim to prevent falling\n";
                          prints "hit victim's back 4 times\n";
                          prints "squeeze victim's chest 4 times\n";
                          prints "sweep victim's mouth with finger\n", primary)
           ELSE (prints "hit victim's back 4 times\n";
                          prints "squeeze victim's chest 4 times\n";
                          prints "sweep victim's mouth with finger\n", primary)
     ELSE (prints "call an ambulance\n", unconscious)

  STATE: unconscious
    IF (reads "Has ambulance arrived? ") IS
     "yes" : (prints "let EMT staff take over\n", GOAL)
     ELSE IF (reads "Is the obstruction clear? ") IS
         "yes" : IF (reads "Is there a pulse? ") IS
          "yes" : IF (reads "Is the victim breathing? ") IS
           "yes" : (prints "You just helped save a life!\n", primary)
           ELSE (ar, unconscious)
          ELSE (prints "apply cpr : 15 compressions, 2 breaths\n", unconscious)
        ELSE IF (reads "Is the victim breathing? ") IS
          "yes" : (prints "hit victim's back 4 times\n";
                          prints "apply 4 chest thrusts\n";
                          prints "sweep victim's mouth with finger\n", unconscious)
          ELSE IF (reads "Is there a pulse? ") IS
           "yes" : (ar, unconscious)
           ELSE (prints "hit victim's back 4 times\n";
                          prints "apply 4 chest thrusts\n";
                          prints "sweep victim's mouth with finger\n", unconscious)

GOAL OF choke
```

Figure 3.21 Mugol module produced from the Mugmaker file of figure 3.19

Each of the two induced states passes control sometimes to the other induced state, and sometimes to the goal state. Thus looping control algorithms can be induced from a set of

example files. In figure 3.11:

(a) Under DECLARATIONS, "ar" (artificial respiration) is a child of choke, and is called as one of the ACTIONs of choke in the STATE called "unconscious".

(b) There are three ACTIONS sections. The first describes actions which are global to both the STATEs "conscious" and "unconscious." Each STATE also has local ACTIONS to be used only for the rule in that STATE. All ACTIONS apart from "ar" call the C-coded primitive "prints" to print a message to the user.

(c). No global CONDITIONS are represented in the induction file. However, both states have local CONDITIONS. These all involve calling "reads" to prompt the user for an answer. However, a more complex example might call a sub TEST module to test the condition.

The induction file presented in figure 3.20 is transformed by Mugmaker into the Mugol module of figure 3.21.

3.9.4. Induction of a hierarchy of rules

Much of the power of the Mugol environment to solve industrial-scale problems derives from the ability to induce a hierarchy of rules from a set of example files. A list of actions, conditions, and examples is supplied for each rule in the system, and the automatic induction process generates both the individual rules for each state, as well as the connections between states in a module, and between modules.

The choke example above uses a sub-module called "ar" (artificial respiration) as a sub action module. The example file of figure 3.22 defines the ar routine. The induction file was transformed by Mugmaker into the Mugol program shown in figure 3.23.

```
ACTION: main.choke.ar  /* artificial respiration */

        STATE: arproc
                ACTIONS:
                        airway   [prints "tilt head - open airway\n"]
                        breath   [prints "give 1 breath/5 sec\n"]
                        ok       [prints "reassure victim\n"]
                        stop     [prints "stop applying ar\n"]
                        monitor  [prints "monitor victim\n"]
                        cpr      [prints "apply cpr -15/2\n"]
        CONDITIONS:

        pulse                    [reads "Is there a pulse?"]
                                                {yes no}
                breath           [read "Is victim breathing?"]
                                                {yes no}
                        breath   [read "Is victim conscious?"]
                                                {yes no}
        EXAMPLES:

        yes     no      no       => (airway;breath, arproc)
        yes     yes     no       => (stop;monitor, arproc)
        yes     yes     yes      => (ok, GOAL)
        no      no      no       => (cpr, arproc)
```

Figure 3.22 Induction file for artificial respiration

```
MODULE main.choke.ar IS

 STATE: arproc
  IF (reads "Is victim breathing?") IS
   "yes" : IF (reads "Is victim conscious?") IS
    "yes" : (prints "reassure victim\n", GOAL)
    ELSE (prints "stop applying ar\n";
          prints "monitor victim\n", arproc")
   ELSE IF (reads "Is there a pulse?") IS
    "yes" : (prints "tilt head - open airway\n";
          prints "give 1 breath /5 sec\n", arproc)
    ELSE (prints "apply cpr - 15/2\n", arproc)

GOAL OF ar
```

Figure 3.23 Mugol module produced from the Mugmaker file of figure 3.22

3.10. External information sources

Virtually all large expert system applications will require access to external information sources, such as sensors, files, data bases, and specially written or existing programs. External resources can also be used to incorporate alternate reasoning approaches into a system. External output to control devices to update data bases may also be desired.

To deal with these demands, the Mugol environment allows the developer to set up separate processes under the operating system. Communication with these other processes is defined by a simple interface which allows external programs to be called in the same manner as Mugol modules (see section 3.6.2). At execution time instructions and data are passed across a UNIX pipe between the Mugol environment and the external programs. These programs can be written in any language supported by UNIX (eg. FORTRAN, C, LISP, Prolog).

3.11. Conclusion

The Mugol environment is an expert system building package intended to solve many of the problems involved in the construction of large knowledge based programs. An inductive learning

system (Mugmaker) allows rapid and effective acquisition of expert knowledge. The Mugol language allows structured organisation of large quantities of knowledge acquired in such a manner. Mugol also provides a facility for presenting ordered explanation of reasoning to the level of elaboration required. However, the Mugol environment does not explicitly support any form of reasoning based on partial certainty.

In comparison to other expert system approaches we believe that although our knowledge representation, in the form of decision trees, is no better than that of production systems, the fact that knowledge can be presented in the form of examples from which rules can be refined means that the process of knowledge acquisition is greatly eased. It has been noted often during the construction of Mugol-based applications that whereas designers using dialogue acquisition methodologies talk of constructing prototype systems in terms of years, Mugol-based applications have been consistently prototyped in around six person months.

Typical expert system applications contain aspects of both classification and control tasks. The Mugol environment provides a consistent knowledge representation for these disparate problem elements. Furthermore, an interface to external sources and sinks of information is provided.

A method of inducing the state transition structure of Mugol modules from trace information would be desirable. The theoretical basis for such a mechanism is given in chapters 5 and 7.

4

ARCH

Abstract. A small robot planning system, ARCH, was built by the author and an augmented Mugol version has subsequently been successfully tested by Barry Shepherd using a Rhino robot (see Appendix H). Barry Shepherd's system builds a specified arch out of children's blocks.

4.1. Introduction

In this chapter we describe how a planner for blocks world problems can be inductively generated using the Mugol environment. The problem chosen is identical to that attempted by Dechter and Michie (Dechter and Michie, 1984). Whereas Dechter and Michie used "Expert-Ease" (McLaren, 1984), the choice of Mugol for our re-implementation overcame several of the problems which they encountered.

The domain of planning deals with finding a sequence of tests and conditional operations which transforms some initial situation into a goal situation. The problem domain is described by a situation-space, containing a set of legal situations together with a set of actions. Each action is usually described in terms of a precondition that should be satisfied by a situation before the action can be applied, and a postcondition which should hold following the action's application. When using the inductive algorithm we formulate the set of situations as the set of objects. The set of attributes is the set of features by which the situations are described. The set of actions are the set of classes.

For every goal, there is a set of plans, each corresponding to an initial situation, for achieving this goal. Given such a goal, a situation is classified to the first action in the plan for achieving this

goal from this situation. Accordingly, for every goal, a set of examples which partially describe the above relationships between situations and actions can be created. For this set of examples, a conditional rule is induced. Thus given an initial situation, the induced rules can be used to select the appropriate action.

4.2. The problem: building a five block arch

The problem involves generating a plan for the simple activity of building an arch out of five blocks named *A*, *B*, *C*, *D* and *beam*. Five platforms are used to hold the blocks. These platforms are called *pile1*, *pile2*, *beam store*, *right arch* and *left arch*. In the initial world situation, the blocks A, B, C, D are stacked on pile1 and pile2. The beam is placed on the beam store. Figure 4.1 illustrates a typical initial situation. The goal is to build an arch on the *right arch* and *left arch* platforms in the configuration shown in figure 4.2.

The problem was divided into a hierarchy of sub-problems for the Mugol environment. This hierarchy is shown in figure 4.3. Dechter and Michie (1984) showed that without such problem decomposition, inductively generated decision tree solutions require an unmanageable number of

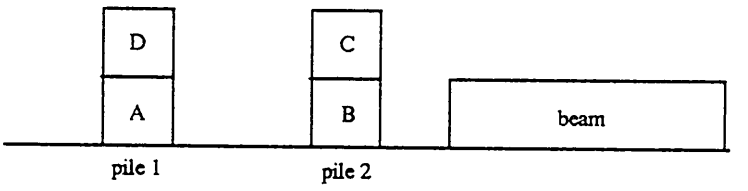


Figure 4.1 An initial situation

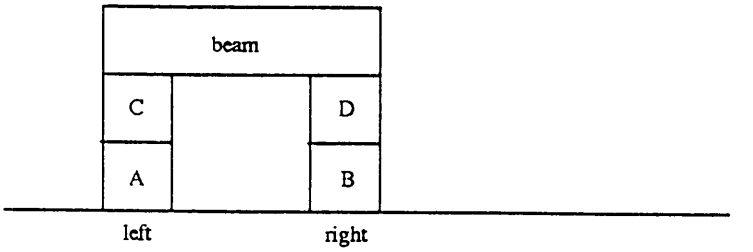


Figure 4.2 The goal situation

examples. However, whereas the package used by Dechter and Michie did not allow the creation and iterative execution of a hierarchy of decision trees, the Mugol environment does (see section 3.9.4). The action *main* merely calls the top level action *arch*.

4.3. The action arch

The top level action, *arch*, defines the order in which five different goals must be reached, ie. A, B, C and D must be moved to their respective positions and the beam must be placed across the top. In figure 4.4 we give the Mugmaker file which describes this top level goal. Note that at this level no examples are needed as the task can be described by this simple sequence of goals.

main.arch is the path leading from the root of the problem hierarchy to this action node. *arch* has three children in the problem hierarchy. These are *onto*, *from* and *to*. *onto* produces a plan which will move a particular block onto its goal position. *from* and *to* are actions which, given that a block is clear of other blocks, respectively pick up a block from a given position and place it on some other given position. The action names are used as infix operators (see section 3.7) in expressions like

"A" onto "the left arch"

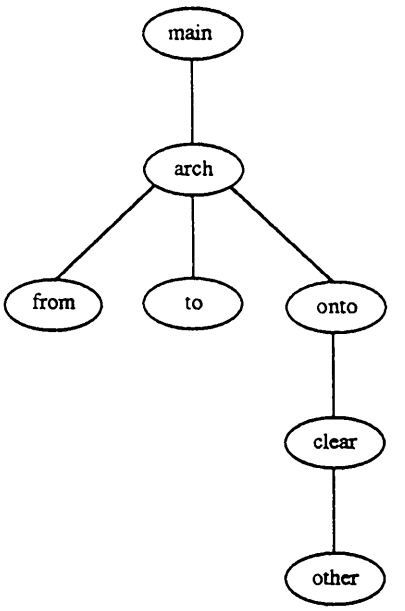


Figure 4.3 Hierarchical breakdown of the problem

```
ACTION: main.arch

        DECLARATIONS:
                [ CHILD: onto {100,1,1}, from {110,1,1}, to {100,1,1} ]

    STATE: only
        [("A" onto "the left arch";
          "B" onto "the right arch";
          "C" onto "A (left arch)";
          "D" onto "B (right arch)";
          "BEAM" from "beam store" to "C and D",
          GOAL)]
```

Figure 4.4 Mugmaker file describing the top level goal

and

"BEAM" from "beam store" to "C and D"

Thus the declaration

CHILD: onto {100,1,1}, ...

says that *onto* has a precedence of 100 and takes 1 argument on the left, and one on the right. This allows a more English-like statement of the required activity than that produced by Dechter and Michie (1984).

4.4. The action onto

The action *onto* is described by the Mugmaker file of figure 4.5. *onto* has two INput parameters called *block* and *place*, as described in the previous section. It also has one child in the problem hierarchy. *onto*, like all the modules within ARCH is a single state module. The examples use a number of ACTIONS which are described between square brackets as small pieces of Mugol code. eg.

block from "pile1" to place

```
ACTION: main.arch.onto

        DECLARATIONS:
                [ IN: string {block, place}
                  CHILD: clear]

        STATE: decide

                ACTIONS:

                        x1toplace       [block from "pile1" to place]
                        x2toplace       [block from "pile2" to place]
                        clearx1  [clear block "pile1"]
                        clearx2  [clear block "pile2"]
                        null            [null]

                CONDITIONS:

                xon
                                        [reads "is " # block # " on " #
                                                place # "? (yes/no) "]
                                        {yes    no }
                        clearx
                                        [reads "is " # block #
                                                " clear of blocks? (yes/no) "]
                                        {yes    no }
                                pilex
                                        [reads "which pile is " #
                                                block #
                                                " on? (pile1/pile2) "]
                                        {pile1 pile2}
                EXAMPLES:

                yes     -       -       => (null, GOAL)
                no      yes     pile1   => (x1toplace, GOAL)
                no      yes     pile2   => (x2toplace, GOAL)
                no      no      pile1   => (clearx1; x1toplace, GOAL)
                no      no      pile2   => (clearx2; x2toplace, GOAL)
```

Figure 4.5 The Mugmaker file describing the action 'onto'

says that the block given to *onto* as a first parameter must be picked up from pile1 and put onto the place given as *onto*'s second parameter. The possible situations are described using the CONDITIONS xon, clearx and pilex. These merely invoke questions which are directed at the user.

In terms of these three CONDITIONS, a set of EXAMPLES are given which specify what to do under different circumstances. The action done for each circumstance is paired with the next state to enter, which for each of these is GOAL. Entering the GOAL state returns control from the action being carried out.

Note the use of parameterisation (i.e. block and place) in this module. This was found by Dechter and Michie (1984) to be particularly awkward to simulate using Expert-Ease (McLaren, 1984).

4.5. The action clear

The action *clear* is described by the Mugmaker file of figure 4.6. In the examples for *clear*, if there is nothing on the block being cleared, then the goal has been reached. If one of the blocks A-D is on the block being cleared, then *clear* is recursively called for the upper block. Once the upper block has been cleared, it is moved to the other pile.

The use of recursion in an inductively generated solution for clearing blocks was suggested by Dechter and Michie (1984). However, again because of the limitations of Expert-Ease, a separate decision tree needed to be developed for clearing "A", clearing "B", clearing "C" and clearing "D". As shown, this can be avoided when using the Mugol environment by careful use of parameterisation.

However, figure 4.6 illustrates a weakness of the Mugol environment which was noted by Dechter and Michie with reference to Expert-Ease i.e. actions and condition values cannot be parameterised. With the solution shown in figure 4.6 it would have been useful to be able to write examples of the form

```
ACTION: main.arch.onto.clear

        DECLARATIONS:
                [ IN: string {block, place}
                  CHILD: other {120,0,1}]
    STATE: decide
        ACTIONS:
                Aother          ["A" from place to other place ]
                Bother          ["B" from place to other place ]
                Cother          ["C" from place to other place ]
                Dother          ["D" from place to other place ]
                clearA          [clear "A" place]
                clearB          [clear "B" place]
                clearC          [clear "C" place]
                clearD          [clear "D" place]
                null            [null]
        CONDITIONS:
                onx
                                [reads "which block is on " # block
                                        # "? (A/B/C/D/nothing) "]
                                {nothing A B C D}

        EXAMPLES:
                nothing => (null, GOAL)
                A               => (clearA; Aother, GOAL)
                B               => (clearB; Bother, GOAL)
                C               => (clearC; Cother, GOAL)
                D               => (clearD; Dother, GOAL)
```

Figure 4.6 Mugmaker file describing the action 'clear'

```
CONDITIONS:

        onX     ...

EXAMPLES:

        nothing         => (null, GOAL)

        Y               => (clear(Y); to_other(Y), GOAL)
```

4.6. A session

Given the initial situation presented in figure 4.1, the Mugol program produced from the Mugmaker files for this problem produces the interaction shown in figure 4.7. User answers are shown in italics and primitive operations in the plan are shown in bold print.

Clearly, although rather a lot of questions are asked, a plan is produced which satisfies the goal.

4.7. Conclusion

Arch building is a classic Artificial Intelligence problem in which search-based planners are often employed. The Mugol environment facilitates the development of an elegant inductive solution to the ARCH problem by supporting hierarchical problem decomposition, the use of variables, parameterisation and user definable expression syntax. Whereas a classical search-based planning solution to this problem might be able to deal with four or five brick problems before computational overheads became too high, Barry Shepherd has shown that the solution presented here can be extended to thirty or forty brick problems without such overwhelming overheads (see Appendix H). The advantage of search-based planning over merely programming a solution lies in the fact that the specification is simple, declarative and compact. However, inductive specifications also have these advantages.

Further work needs to be done to allow the use of variable condition values in examples.

is A on the left arch? (yes/no) *no*
is A clear of blocks? (yes/no) *no*
which pile is A on? (pile1/pile2) *pile1*
which block is on A? (A/B/C/D/nothing) *C*
which block is on C? (A/B/C/D/nothing) *nothing*

!! pick up C from pile1
!! put C onto pile2
!! pick up A from pile1
!! put A onto the left arch

is B on the right arch? (yes/no) *no*
is B clear of blocks? (yes/no) *no*
which pile is B on? (pile1/pile2) *pile2*
which block is on B? (A/B/C/D/nothing) *D*
which block is on D? (A/B/C/D/nothing) *C*
which block is on C? (A/B/C/D/nothing) *nothing*

!! pick up C from pile2
!! put C onto pile1
!! pick up D from pile2
!! put D onto pile1
!! pick up B from pile2
!! put B onto the right arch

is C on A (left arch)? (yes/no) *no*
is C clear of blocks? (yes/no) *no*
which pile is C on? (pile1/pile2) *pile1*
which block is on C? (A/B/C/D/nothing) *D*
which block is on D? (A/B/C/D/nothing) *nothing*

!! pick up D from pile1
!! put D onto pile2
!! pick up C from pile1
!! put C onto A (left arch)

is D on B (right arch)? (yes/no) *no*
is D clear of blocks? (yes/no) *yes*
which pile is D on? (pile1/pile2) *pile2*

!! pick up D from pile2
!! put D onto B (right arch)

!! pick up BEAM from beam store
!! put BEAM onto C and D

Figure 4.7 User interaction for blocks problem

5

Overview of grammatical induction theory

Abstract. The Mugol environment in its present form demands that the control structure of Mugol finite state machines be hand-coded. In this chapter and chapter 7 we investigate techniques for automatically constructing finite state structures from trace information. The techniques are based on "grammatical induction", i.e. discovery of grammar from example sentences. First we present a survey of algorithms which infer a regular language from a given subset of that language. We introduce a general algorithm for this task, which has a low order polynomial time complexity. Several previously devised algorithms are demonstrated by way of adaptations to this general algorithm.

5.1. Introduction

This chapter deals with *grammatical induction* techniques, that is, methods of hypothesising the grammar rules of a language from example "sentences". For the reader's convenience we will repeat some of the details concerning finite state machines found in chapter 2.

We have limited the scope of investigation to the inference of *regular languages* (for a general survey of inductive inference methods see (Angluin and Smith, 1982)). As an example of the kind of problem which we intend to solve, let us suppose that we present the inductive inference program with the following sample of strings

aaabbb

ab

abb

b

a

We might expect it to return with the rule

a^+b^+

This represents the regular language

one or more a's followed by one or more b's

Alternatively we can describe a machine which accepts strings of this kind diagrammatically as a finite state acceptor. Indeed it has been shown (Hopcroft and Ullman, 1979) that any regular language can be recognised by some finite state acceptor. As the converse is also true, i.e. any finite state acceptor can be expressed as a regular expression, these representations are equivalent.

The aim of this investigation is to develop an algorithm for inducing Mugol modules (see chapters 2 and 3) from traces of their intended execution (i.e. sequences of calls to predefined tests and actions). Finite state acceptors differ from the type of finite state automata which represent Mugol modules, in that the arcs of finite state automata are labelled with pairs of tokens rather than singlets. Although the methods of induction presented here are for finite state acceptors, adaptations

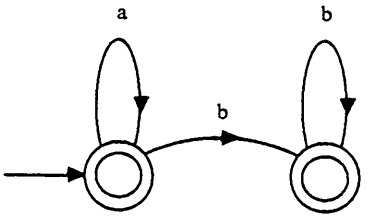


Figure 5.1 The finite state acceptor representing the language a*b*

of these algorithms to produce finite state automata are presented in chapter 7. These adapted algorithms build automata in Mealy machine form from trace information. We call this adapted form of grammatical induction *sequence induction.*

Most papers on this subject suggest particular solutions to the problem. The algorithms presented are tailored to be as efficient as possible for the heuristic being used; for the sense in which *heuristic* is here used, see section 5.4.5 and following. We attempt to show that by devising a general algorithm ("IM1", section 5.4.3 below) which can be specialised to any one of a number of existing grammatical inference schemes eases comparison of the properties of the latter. In section 5.2 and 5.3 we present a brief resumé of the background of this research. In section 5.4 we give the theoretical results of grammatical induction from positive samples. In section 5.5 the issues of section 5.4 are discussed in an informal fashion by use of examples which illustrate the behaviour of various algorithms.

5.2. Language Identification

Gold's theoretical study of *language learnability* (Gold, 1967) introduced an abstract setting for the problem of grammatical induction. The grammatical induction problem is that of deciding which language L from a class of languages C is characterised by a set of examples E. An example from the set E can be positive or negative in the sense that it is stated whether it is inside or outside L. Thus supposing C is the set of regular languages over the symbol set $\Sigma = \{0,1\}$, then

E = { <00,*in*>, <1,*out*>, <11,*in*>, <00011,*in*>, <01000,*out*>,..}

might exemplify the regular language $L = (0,10^*1)^*$ containing only binary strings of even parity (this is the same problem as that of section 2.6.2 with 0 and 1 replacing false and true respectively). In Gold's work, inference is carried out on an infinite list of examples containing one or more occurrences of every possible string along with an indication of whether it is in the target language. Gold defined an inference algorithm I as *identifying a language in the limit* if and only if after a certain number of examples are provided, I chooses the correct explanation and does not subsequently change this explanation as more examples are presented.

Next Gold introduced the general inference technique of *identification by enumeration* in which a generator exhaustively postulates in some fixed order L_1, L_2, L_3, ... all languages L_i from the set C and returns the first which is consistent with the examples so far.

Gold went on to show that for any class of languages C containing all finite languages (those with a finite number of legal sentences) and at least one infinite language, it is impossible for an inference algorithm to identify an arbitrary element of C using only positive examples. This can easily be seen by taking C to be the set of regular languages over the symbol set Σ and showing that for any positive example set E there are at least two languages which can be postulated; namely the universal language Σ^* and the finite language containing only the members of E.

Gold distinguishes between two types of presentation of mixed positive/negative examples. A presentation can be *text*, in which case the inference algorithm I is presented with a passive list of facts. Alternatively, I can be supplied with an *informant* or *oracle*, an agent which answers membership questions about the unknown language.

5.3. Mixed Positive/Negative Presentations

Although regular sets can be identified from positive and negative *text*, Angluin (Angluin, 1978) has shown that the problem of finding a minimal regular expression from such samples is NP-hard. Furthermore, Gold (Gold, 1978) showed that the corresponding problem of finding a minimal finite acceptor from positive and negative samples is also NP-hard.

Given an *oracle*, Moore (Moore, 1956) has shown that it is possible to identify a language only if we are also given additional information about L. Moore's algorithm has an NP complexity bound, and requires, as additional information, an upper bound on the number of states in the canonical (state-minimal) acceptor of L. Pao and Carr (Pao and Carr, 1978) and later Angluin (Angluin, 1982a) suggest the use of a representative sample of L, that is, a finite subset of L that exercises every transition in the canonical acceptor of L. Whereas Pao and Carr's enumerative algorithm is NP in the number of queries made of the oracle, Angluin's algorithm requires only a polynomial number of queries for the same problem.

5.4. Theoretical results of grammatical induction from positive samples

In section 5.2 we stated Gold's theorem which says that particular classes of languages cannot be identified in the limit from positive examples only. This does not imply necessarily that negative examples are an imperative, only that some form of additional constraint must be used in order to guarantee identification in the limit. In this section we investigate algorithms which use parameterised constraint predicates which allow identification of languages in the limit from positive example sets. As might be expected, all these algorithms have the property that the proposed language L does at least contain the sample set S.

As far as the author is aware, there are only four algorithms in the literature for inducing finite state automata from positive examples. A general algorithm is given in § 5.4.3 which, with suitable alteration of the driving heuristic produces the same results as all of the existing algorithms except Angluin's (Angluin, 1982b).

5.4.1. Definitions

Below we present some basic definitions from set theory and formal language theory which will be used both in this chapter and chapter 7. The notation used roughly follows that of Angluin in (Angluin, 1982b).

$|S|$ - the *cardinality* of the set S.

2^S - the *power set* of S, $2^S = \{ S' : S' \subseteq S \}$. $|2^S| = 2^{|S|}$.

Σ - a finite alphabet with cardinality $|\Sigma| \geq 2$.

Σ^* - the infinite set of strings made up of zero or more letters from Σ.

λ - the empty string.

uv - the concatenation of the strings u and v.

$|u|$ - the length of string u.

w^r - the reverse of the string w.

L - a *language* L is any subset of Σ^*.

L^r - the reverse of L, $L^r = \{w^r: w \in L\}$.

$\mathrm{Pr}(L)$ - the prefixes of elements of L, $\mathrm{Pr}(L) = \{u$: for some $v, uv \in L\}$.

$T_L(u)$ - the left-quotient of u in L, $T_L(u) = \{v: uv \in L\}$.

$T_L^k(u)$ - the k-tails of u in L, $T_L^k(u) = \{v: v \in T_L(u), |v| \leq k\}$.

S^+ - a *positive sample* S^+ of L is any finite subset of L.

π_S - a *partition* of some set S, π_S, is a set of pairwise disjoint nonempty subsets of S such that the union of all sets in π_S is equal to S.

$B(s,\pi_S)$ - the unique *block* (element) of π_S containing s, where $s \in S$.

refines - given two partitions, π and π', π *refines* π' if and only if every block of π' is a union of blocks of π.

$\chi_{\pi_S}(u,v)$ - the characteristic predicate function of a partition over S is defined as

$$\chi_{\pi_S}(s,s') = \begin{cases} true & \text{if } s,s' \in S,\ B(s,\pi_S) = B(s',\pi_S) \\ false & \text{otherwise} \end{cases}$$

χ_{π_S} can easily be shown to be an *equivalence relation.* A relation R is an equivalence relation if and only if it has the properties of being *reflexive* (for all $s \in S^+$, $\chi_{\pi_S}(s,s)$ is true), *transitive* ($\chi_{\pi_S}(s,s')$ and $\chi_{\pi_S}(s',s'')$ implies $\chi_{\pi_S}(s,s'')$) and *symmetric* ($\chi_{\pi_S}(s,s')$ implies $\chi_{\pi_S}(s',s)$).

A - an acceptor is a tuple $A = (Q, \Sigma, \delta, I, F)$, where Q is the non-empty finite *state set*, Σ is the input alphabet, δ: $2^Q \times \Sigma \rightarrow 2^Q$, is the *transition function*. The transition function for strings δ^*: $2^Q \times \Sigma^* \rightarrow 2^Q$, is defined using the recursive definition

$$\delta^*(Q',\lambda) = Q'$$
$$\delta^*(Q',bu) = \delta^*(\delta(Q',b),u)$$

where $q \in Q$, $b \in \Sigma$ and $u \in \Sigma^*$, $I \subseteq Q$ is the set of *initial states*, and $F \subseteq Q$ is the set of *final states*. A *deterministic acceptor* is defined similarly, the difference being that I, δ and δ^* represent single element sets. When dealing with deterministic acceptors, we will write q_0 for the initial state set $I = \{q_0\}$, $\delta(q,b) = q'$ for $\delta(\{q\},b) = \{q'\}$ and $\delta^*(q,u) = q'$ for $\delta^*(\{q\},u) = \{q'\}$.

$L(A)$ - the regular language $L(A)$ recognised by A consists of strings u which are accepted by A, that is $\delta^*(I,u) \in F$.

δ^r - the reverse transition function δ^r is defined as

$$\delta^r(Q',a) = \{q' : q \in \delta(q',a)\} \qquad \textit{for all } a \in \Sigma,\ q \in Q.$$

A^r - the reverse of the acceptor A is $A^r = (Q, \Sigma, \delta^r, I, F)$. Diagrammatically, A^r is A with the initial and final states swapped and all transition arcs reversed in direction. It can easily be shown that $L(A^r) = (L(A))^r$.

In the following, let $A = (Q, \Sigma, \delta, I, F)$ and $A' = (Q', \Sigma', \delta', I', F')$ be two acceptors.

a–successor- for some $q, q' \in Q$, $a \in \Sigma$, q is an *a-successor* of q' if and only if $q \in \delta(q',a)$.

k–follower - a string u is said to be a k-follower of a state $q \in Q$ if and only if $|u| = k$ and $\delta(q,u) \neq \emptyset$. Every state has exactly one 0-follower, namely λ.

k–leader - a string u is a k-leader of a state $q \in Q$ if and only if $\delta^r(q,u^r) \neq \emptyset$. Every state also has exactly one 0-leader, λ.

isomorphic - we say that A is *isomorphic* to A' if and only if there exists a bijective mapping h: $Q \rightarrow Q'$ such that $h(I) = I'$, $h(F) = F'$, and for every $q \in Q$ and $b \in \Sigma$, $h(\delta(q,b)) = \delta'(h(q),b)$. In other words, two acceptors are isomorphic if a renaming of their states makes them identical.

subacceptor - A' is a *subacceptor* of A if and only if $Q' \subseteq Q$, $I' \subseteq I$, $F' \subseteq F$ and for every $q' \in Q'$ and $b \in \Sigma$, $\delta'(q',b) \subseteq \delta(q',b)$. Alternatively, A' is a subacceptor of A if and only if $L(A') \subseteq L(A)$. Diagrammatically a subacceptor is formed from an acceptor by removing some nodes and arcs from the transition diagram of the original acceptor.

live - if $q \in Q$, then q is called *live* if and only if for some u,v, $q \in \delta^*(I,u)$ and $\delta(q,v) \cap F \neq \emptyset$. A state is called *dead* if it is not live. A' is called a *stripped subacceptor* of A if and only if $Q' = \{q' : q' \in Q \text{ and } q' \text{ is live}\}$.

A/π_Q - let π_Q be some partition of Q, the state set of A. $A' = A/\pi_Q$, the *quotient* of A and π_Q is defined as follows. Q' is the set of blocks of π_Q. I' is the set of blocks of π_Q that contain at least one element of I. Similarly, F' is the set of blocks of π_Q that contain at least one element of F. Block B_2 is a member of $\delta'(B_1,a)$ if and only if there exists $q_1 \in B_1$ and $q_2 \in B_2$ such that $q_2 \in \delta(q_1,a)$.

$A(L)$ - the *canonical* or *minimal acceptor* for a language L, $A(L) = (Q, \Sigma, \delta, I, F)$ is defined as follows

$$Q = \{T_L(u): u \in Pr(L)\},$$

$$I = \{T_L(\lambda)\} \quad \text{if } L \neq \emptyset, \text{ otherwise } I = \emptyset,$$

$$F = \{T_L(u): u \in L\},$$

$$\delta(T_L(u),a) = T_L(ua) \quad \text{if } u,ua \in Pr(L).$$

Note that the canonical acceptor $A(L)$ has the minimum number of states possible for an acceptor of L. None of these states is dead, thus $A(L)$ is *stripped*. Any acceptor A' which is *isomorphic* to $A(L)$ is called canonical.

$PT(S^+)$ - if S^+ is a positive sample of L, we define the *prefix tree acceptor* of S^+, $PT(S^+) = (Q, \Sigma, \delta, I, F)$, as

$$Q = Pr(S^+),$$

$$I = \{\lambda\} \text{ if } S^+ \neq \emptyset, \text{ otherwise } I = \emptyset,$$

$$F = S^+,$$

$$\delta(u,a) = ua \text{ whenever } u,ua \in Pr(S^+).$$

representative sample

- S^+ is a *representative sample* of L if and only if for every transition (q,b) in $A(L)$ there is a string $u \in S^+$ which exercises (q,b).

acceptor for S^+. Let $\pi_{Pr(S^+)}$ be the partition π_L restricted to $Pr(S^+)$. Then $A_0/\pi_{Pr(S^+)}$ is isomorphic to a subacceptor of $A(L)$. Thus $L(A_0/\pi_{Pr(S^+)}) \subseteq L$.

Corollary 5.2. $L(A_0/\pi_{Pr(S^+)})$ is contained in L.

The following Lemma is due to Fu and Booth (Fu and Booth, 1975)

Lemma 5.3. Every acceptor $A/\pi_{Pr(S^+)}$ derived from the prefix set S^+ is a valid solution.

5.4.3. Algorithm IM1

We now present a simple, though general, algorithm for carrying out inference by merging the states of $PT(S^+)$. Many of the algorithms in the literature are special cases of this algorithm. Describing these algorithms in terms of our algorithm, IM1, facilitates their presentation and comparison. To the author's knowledge no algorithm similar to IM1 has appeared in any publication previously.

IM1 applies the characteristic predicate $\chi_{\pi_{Pr(S^+)}}(u,v)$ (hereafter called $\chi(u,v)$) to every pair $u,v \in Pr(S^+)$. If χ returns *true*, IM1 merges the blocks containing u and v. The resultant acceptor A_0/π_f is non-deterministic. Hopcroft and Ullman (Hopcroft and Ullman, 1979) give an algorithm which can be used to convert this to the equivalent minimal deterministic acceptor.

Algorithm IM1

Input: a nonempty positive sample S^+

Output: the acceptor $A_0/\pi_{Pr(S^+)}$

* Initialisation Let $A_0 = (Q_0, \Sigma, \delta_0, I_0, F_0)$ be $PT(S^+)$. Let π_0 be the trivial partition of Q_0. Let i = 0.

* Merging For all pairs (u,v) in Q_0 do **begin**

If $\chi(u,v)$ then **begin**

Let $B_1 = B(u,\pi_i)$, $B_2 = B(v,\pi_i)$.

Let π_{i+1} be π_i with B_1 and B_2 merged.

Increase i by 1. **end end**

*Termination

Let $f = i$ and output the acceptor A_0/π_f.

5.4.4. Time complexity of IM1

As every pairwise test of elements of Q_0 is made, χ is applied $n(n-1)/2$ times, where $n = |Q_0|$. Thus the time complexity of the algorithm is $O(n^2)$.

5.4.5. Heuristics used in the literature

Although the heuristics described in (Angluin, 1982b; Biermann and Feldman, 1972; Levine 1982; Miclet 1980) were not originally described in terms of the function χ of IM1, predicates

giving equivalent results can easily be described and compared in this manner.

5.4.5.1. Biermann and Feldman's k-tail predicate

Biermann and Feldman's heuristic (Biermann and Feldman, 1972) is functionally equivalent to the predicate

$$\chi(u,v) = \begin{cases} true & \text{if } T^k_{S^*}(u) = T^k_{S^*}(v) \\ false & \text{otherwise} \end{cases}$$

where k is some positive integer supplied by the user. The resultant acceptor A is more compact the smaller k is. Biermann and Feldman proved that given the correct value of k for the target language, their algorithm will identify in the limit an acceptor A which when minimised is *isomorphic* to $A(L)$. However, the correct value of k cannot be determined without first knowing what $A(L)$ is. Biermann and Feldman show that by using a hashing function to merge states it is possible to carry out induction using this predicate in $O(n)$ time.

5.4.5.2. Levine's heuristic

Although Levine (Levine, 1982) applied his heuristic algorithm primarily to inference of tree systems, he shows that it is also possible to use it for inference of finite acceptors. Levine defines a strength function which measures the maximum overlap between pairs of tail sets.

$$Stren(u,v) = \max_i \left[\frac{2|T^i_{S^*}(u) \cap T^i_{S^*}(v)|}{|T^i_{S^*}(u)| + |T^i_{S^*}(v)|} \right] \qquad , i \geq 0$$

The heuristic predicate he uses is

$$\chi(u,v) = \begin{cases} true & \text{if } Stren(u,v) \geq Strn \\ false & \text{otherwise} \end{cases}$$

where $Strn$ is a user defined parameter in the range 0 to 1. As with Biermann and Feldman's k parameter, the acceptor has a compactness which is roughly proportional to the value of $Strn$. The calculation of $Stren$ itself has an upper bound time complexity of $O(n)$, thus giving the complete algorithm a time complexity of $O(n^3)$ when using IM1.

5.4.5.3. Miclet's algorithm

Miclet (Miclet, 1980) designed a heuristic algorithm based on statistical clustering techniques. The algorithm uses a distance function to do successive clustering and merging of states. Although this is a fairly general methodology, in his examples he uses a heuristic which approximates in its results to one described fully by Angluin (Angluin, 1982b). The heuristic identifies in the limit the maximally sized *zero-reversible* language containing the input sample. The heuristic is the simplest of all those presented here. The zero-reversible heuristic described by Angluin is equivalent to

$$\chi(u,v) = \begin{cases} true & \text{if } T_{S^+}(u) \cap T_{S^+}(v) \neq \varnothing \\ false & \text{otherwise} \end{cases}$$

Angluin in (Angluin, 1982b) presents a method of computing A using this heuristic in approximately $O(n)$ time.

5.4.5.4. Angluin's heuristic algorithm for k-reversible languages

Angluin (Angluin, 1982b) has shown that there are a class of languages, that she calls *k-reversible*, which can be identified in the limit. The acceptor A is defined to be "deterministic with lookahead k" if and only if for any pair of distinct states q_1 and q_2, if $q_1, q_2 \in I$ or $q_1, q_2 \in \delta(q_3,a)$ for some $q_3 \in Q$ and $a \in \Sigma$, then there is no string that is a k-follower of both q_1 and q_2. This guarantees that any nondeterministic choice in the operation of A can be resolved by looking ahead k symbols past the current one.

An acceptor A is defined to be *k-reversible* if and only if A is deterministic and A^r is deterministic with lookahead k. A language L is defined to be *k-reversible* if and only if there exists a *k-reversible* acceptor A such that $L = L(A)$.

Angluin presents an algorithm which, starting with the prefix tree acceptor, successively refines acceptors by merging any two states q_1 and q_2 which violate the condition of k-reversibility. The algorithm continues this process until no such pair of states q_1 and q_2 exist. As no more than n mergers can be made (the prefix tree acceptor contains only n nodes), and $O(n^2)$ comparisons must be made for each merger, the time complexity of the algorithm is $O(n^3)$. Angluin shows that her

heuristic will identify in the limit any particular language in any of the classes of k-reversible languages. However, she also shows that not all regular languages are members of a k-reversible language class.

5.4.6. Limitations of existing heuristics

It may be seen from inspection that a common factor of all the heuristics listed above is that $T_{S^+}(u) \cap T_{S^+}(v) \neq \emptyset$ must at least hold for $\chi(u,v)$ to be true. The following theorem shows the limitation of such a requirement.

Theorem 5.4. For any $\chi(u,v)$ which implies $T_{S^+}(u) \cap T_{S^+}(v) = \emptyset$, the induced partition $\pi_{Pr(S^+)}$ is the trivial partition π_0 whenever $|S^+| = 1$.

Proof. Let $S^+ = \{w\}$. Let two distinct prefixes of w be u_1 and u_2. Let $T_{S^+}(u_1) = \{v_1\}$ and $T_{S^+}(u_2) = \{v_2\}$. As u_1 and u_2 are distinct prefixes of w, $|u_1| \neq |u_2|$ and $|w| = |u_1| + |v_1| = |u_2| + |v_2|$. Thus $v_1 \neq v_2$, $T_{S^+}(u_1) \neq T_{S^+}(u_2)$ and $\chi(u_1,u_2)$ will always be false. As no mergers would ever be made, $\pi_{Pr(S^+)} = \pi_0$. QED.

Human beings are capable of making inferential "guesses" about regular languages from single pieces of evidence. For instance, given the string

aaabbb

one might suspect L to be

a*b*

The author's k-contextual algorithm presented in chapter 7 avoids this limitation.

5.5. Informal presentation of results

Having been presented with a sample of a particular regular language, the first step in our general method of finding an appropriate candidate acceptor is to form the unique *prefix tree acceptor* corresponding to the sample. This prefix tree acceptor is itself a finite acceptor. It is

formed by taking each string in the sample and using it to extend a path from the tree root to one of the leaves. The individual segments of the string are used as the labels of the arcs along this path. Moreover, any state at which a string terminates is marked in a special manner with a double circle, and called an *accepting state*. Note that whereas all leaves are accepting states, accepting states can also be found at some internal nodes of the tree. Figure 5.2 illustrates the relationship between the sample and the prefix tree. Clearly this finite acceptor will accept no more and no less than the strings presented in the sample. As with any tree, we can name each node uniquely by describing the path from the root to that node. In the case of the prefix tree acceptor shown above, we can represent the states as the set of all prefixes of strings in the sample,

$Pr(S^+) = \{\lambda,a,b,aa,ab,bb,aab,abb\}$

where λ is the empty string representing the root node, or start state.

Sample, S^+: {ab,bb,aab,abb}

Prefix tree acceptor, $PT(S^+)$:

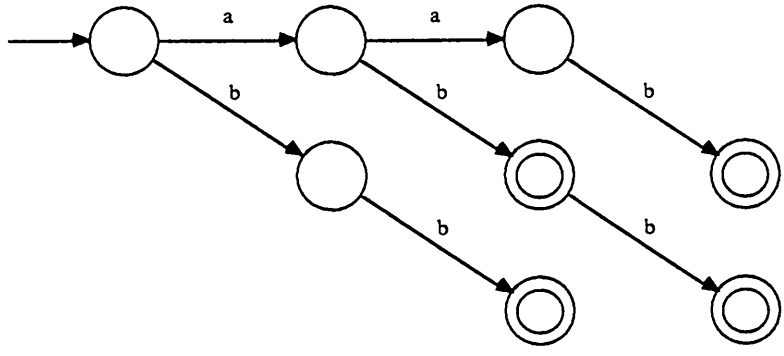


Figure 5.2 A positive sample and its corresponding prefix tree acceptor

By *merging* some of the nodes of the acceptor of figure 5.2 it is possible to form a smaller acceptor which will still accept only the strings represented in the sample. This new acceptor is shown in figure 5.3. The acceptor of figure 5.3 is in fact the smallest, or *canonical* acceptor which will accept only the sample (this has been confirmed algorithmically). By further merger of the states of the acceptor of figure 5.3 we produce acceptors which accept successively more and more strings. In this way it is possible to infer languages which are generalisations of the original sample, and of which the sample is a proper subset. To illustrate this figure 5.4 shows an acceptor formed by the merger of three of the states of the acceptor of figure 5.3. This process of merger, if carried on in an arbitrary manner will in the limit produce an acceptor containing a single state and single arc. Such an acceptor, called a *universal* acceptor, accepts *any* string consisting of symbols present in the original sample. This is shown in figure 5.5. This result is almost certainly an *over-generalisation* of the target grammar. Thus it is necessary to introduce a restraining factor into the inference process. This is done by using a predicate to qualify the merger of candidate states. This

Language accepted, S^+: {ab,aab,abb,bb}

Canonical acceptor, $A(S^+)$:

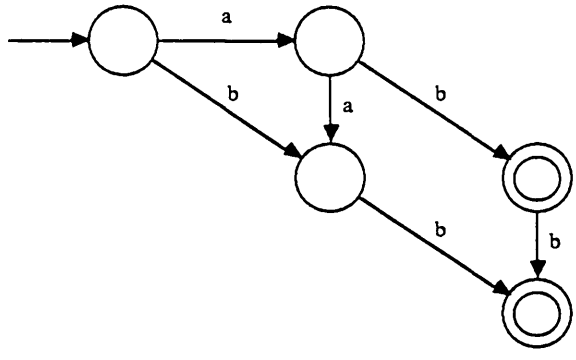


Figure 5.3 The canonical acceptor of the sample

Acceptor of L:

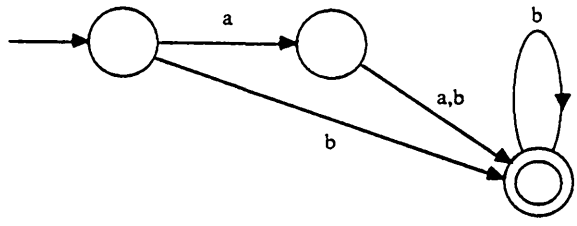


Figure 5.4 A new acceptor derived from that of figure 5.3

Acceptor of L:

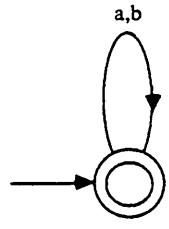


Figure 5.5 The universal acceptor for the symbol set {a,b}

predicate is called the *characteristic predicate* and is often merely a *heuristic*. During the process of inference every possible pair of nodes in the original prefix tree acceptor is tested using the heuristic to decide whether they should be merged in the resultant acceptor.

5.5.1. Various heuristics

All heuristics developed so far for this problem have depended on matching some local properties of pairs of candidate nodes. If the heuristic does find a match then the nodes are merged. Below we sketch informally how four of these matching heuristics work.

5.5.1.1. Biermann and Feldman's k-tail heuristic

Biermann and Feldman (Biermann and Feldman, 1972) describe a heuristic which merges states having identical "k-tail" sets. A k-tail of a node is a string of length k or less formed by taking a directed path from that node to an accepting state in the prefix tree acceptor of the sample. We will refer to the states of the prefix tree acceptor in figure 5.2 by way of the unique prefix of each node (these are given immediately below figure 5.2). Below we denote the k-tail set of a particular node by T^{k}_{S+}(prefix). k is some integer value chosen by the user. Thus for the prefix tree acceptor of figure 5.2, with k=1,

$T^{1}_{S+}(\lambda) = \varnothing$

$T^{1}_{S+}(a) = \{b\}$

$T^{1}_{S+}(b) = \{b\}$

$T^{1}_{S+}(aa) = \{b\}$

$T^{1}_{S+}(ab) = \{\lambda,b\}$

$T^{1}_{S+}(bb) = \{\lambda\}$

$T^{1}_{S+}(aab) = \{\lambda\}$

$T^{1}_{S+}(abb) = \{\lambda\}$

We can now partition the original prefix set into subsets of prefixes with matching tail sets

{{λ},{a,b,aa},{ab},{bb,aab,abb}}

The effect of having merged these nodes is shown in figure 5.6. The reader may notice that two

Language accepted, L: $(a,b)a^*(b,bb)$

Prefix tree acceptor, $PT(S^+)$:

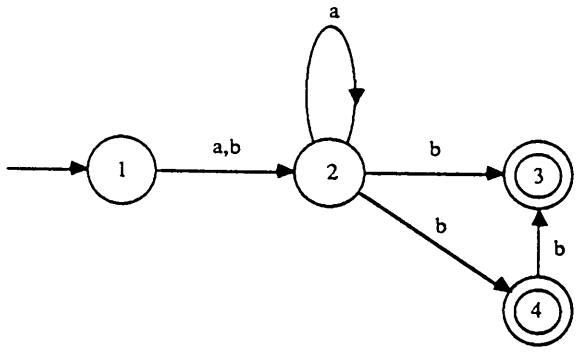


Figure 5.6 Effect of k-tail inference, k=2, on prefix tree acceptor of figure 5.2

arcs labelled with a "b" emanate from the state labelled "2" in the diagram above. This implies that a *non-deterministic* decision must be made at this point when exercising the acceptor. Such an acceptor is called a *non-deterministic acceptor* and can transformed to an equivalent *deterministic* acceptor using a procedure described in (Hopcroft and Ullman, 1979).

5.5.1.2. Levine's heuristic

Levine's (Levine, 1982) heuristic is based on maximising and thresholding a function on each pair of states in the prefix tree acceptor. For each pair of states (u,v) in the prefix tree acceptor we compute the function

$$Stren(u,v) = \max_i \left[\frac{2|T^i_{S^+}(u) \cap T^i_{S^+}(v)|}{|T^i_{S^+}(u)|+|T^i_{S^+}(v)|} \right] \qquad , i \geq 0$$

In order to demonstrate the algorithm, we present below the tail sets of all states in the prefix tree acceptor of figure 5.2. These tail sets are equivalent to k-tail sets with k set to infinity. In

figure 5.7 we show the 2-dimensional matrix representing the computation of *Stren* for all pairs of states.

$T_{S^+}(\lambda)$ = {ab,bb,aab,abb}

$T_{S^+}(a)$ = {b,ab,bb}

$T_{S^+}(b)$ = {b}

$T_{S^+}(aa)$ = {b}

$T_{S^+}(ab)$ = {λ,b}

$T_{S^+}(bb)$ = {λ}

$T_{S^+}(aab)$ = {λ}

$T_{S^+}(abb)$ = {λ}

For purposes of thresholding, the user provides a value *Strn* between 0 and 1. If $Stren(u,v) \geq Strn$ for any pair (u,v) then this pair is merged. Thus if we choose *Strn* to be 2/3 we get the partition representing the universal acceptor (figure 5.5). By setting *Strn* to 4/5 rather, we produce the following partition

{{λ,a,b,aa},{ab,bb,aab,abb}}

	λ	a	b	aa	ab	bb	aab	abb
λ	1	4/5	0	0	0	0	0	0
a	4/5	1	1	1	2/3	0	0	0
b	0	1	1	1	2/3	0	0	0
aa	0	1	1	1	2/3	0	0	0
ab	0	2/3	2/3	2/3	1	1	1	1
bb	0	0	0	0	1	1	1	1
aab	0	0	0	0	1	1	1	1
abb	0	0	0	0	1	1	1	1

Figure 5.7 Matrix of Stren for all pairs of states

Figure 5.8 shows the acceptor representing this partition.

5.5.1.3. Miclet's heuristic algorithm

Miclet (Miclet, 1980) gives an algorithm which is general in the sense that it can be used with a variety of heuristics. However, he uses it with a heuristic which is equivalent to applying Levine's heuristic with *Strn* always set to be the lowest non-zero value represented in the matrix. As shown above, this leads to production of the universal language with our particular example.

5.5.1.4. Angluin's heuristic algorithm for k-reversible languages

Angluin's algorithm (Angluin, 1982b), like others described, uses a parameter k provided by the user. The algorithm operates by successively merging any two states q_1 and q_2 for which one of the conditions represented in figure 5.9 holds. In words these conditions are

---

Language accepted, L: $(a,b)^*b^+$

Acceptor of L:

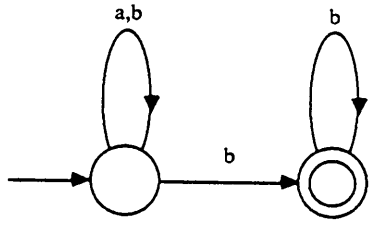


Figure 5.8 Acceptor produced from sample using Levine's algorithm, $Strn$=4/5

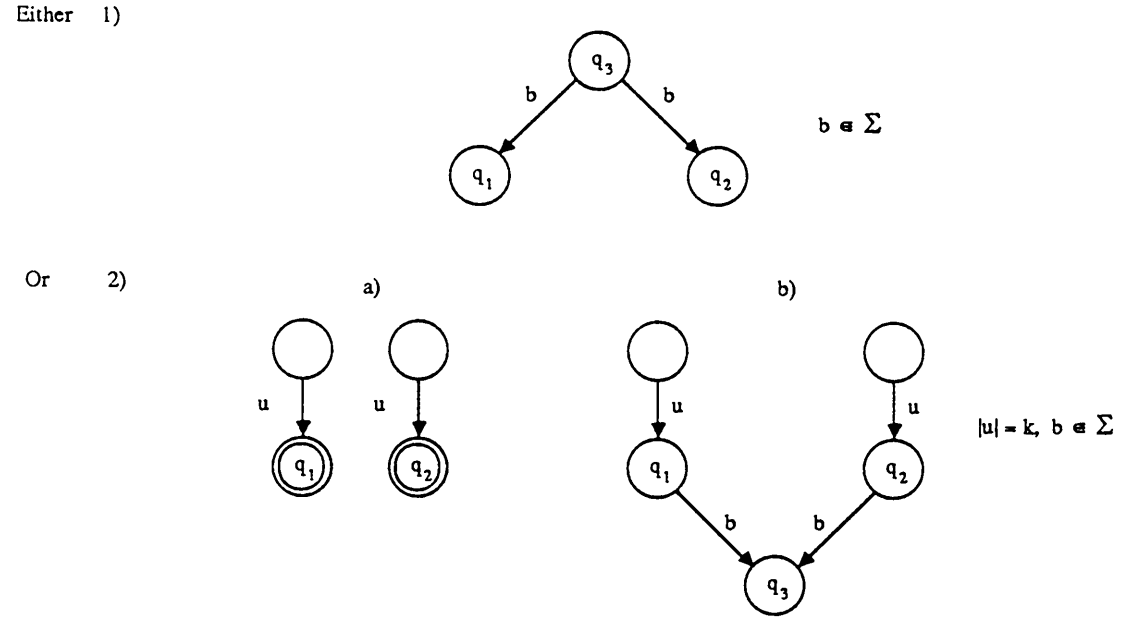


Figure 5.9 Graphical representation of conditions for merger of q_1 and q_2

1) There exist two arcs labelled with a common symbol leading out from state q_3 to q_1 and q_2.

2) Two paths labelled with a common string of length k lead to q_1 and q_2, where q_1 and q_2 are either a) both accepting states or b) both have paths labelled with a common string of length 1 leading to some state q_3.

Figure 5.10 shows the result of applying Angluin's heuristic with k=1 to the prefix tree acceptor of figure 5.2. When minimised, this acceptor represents the language a^*b^+. Of all the results from heuristic predicates presented so far, this seems to be the most intuitively correct guess for the sample S^+. However, as Angluin's algorithm has a time complexity of $O(n^3)$ this algorithm is not practical for large samples.

Language accepted, L: a^*b^+

Acceptor of L:

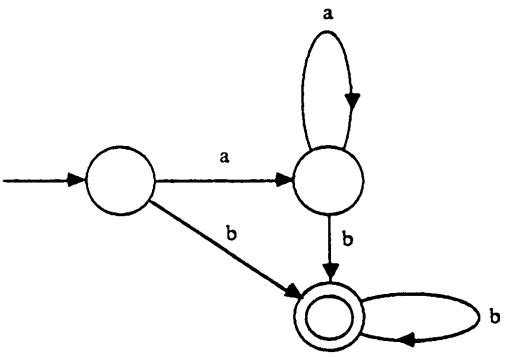


Figure 5.10 The result of applying Angluin's algorithm, k=1

5.6. Conclusion

We have presented a general efficient algorithm for computing and enabling comparison of a very large class of heuristic algorithms. It is evident that an increasing number of heuristic approaches exist for inferring regular languages. One of the goals of this chapter is to show that by use of a general framework for testing and comparing existing approaches re-implementation of large numbers of algorithms can be avoided.

6

Sequence induction applications

Abstract. In this chapter we describe six small but varied applications of the KR and SKR induction algorithms of chapter 7.

6.1. Introduction

Inductive algorithms, such as ID3 (Quinlan, 1979), take sample descriptions of a static world and produce generalisations of these descriptions. For many real world problems it is more appropriate for descriptions of activities to be given as sequences of static descriptions changing over time.

In this chapter we describe the application of sequence induction (see chapters 5 and 7) in a varied set of domains. Our intention is to investigate the applicability of sequence induction techniques within the Mugol environment. In chapter 5 we described a number of algorithms for carrying out grammatical induction. In that chapter we showed that one of these algorithms, that of Angluin (section 5.4.5.4), gave better results than any of the others (section 5.5.1.4). However, we noted that Angluin's k-reversible algorithm runs in time $O(n^3)$ and is thus not practical for large samples. Nevertheless, in the following chapter (section 7.2) we give an algorithm (KR) which is input/output equivalent to Angluin's k-reversible algorithm, but runs in time $O(n)$. In this chapter we use both KR (used in section 6.2) and a sequence induction version of KR called SKR (described in 7.4.5 and used in sections 6.3 - 6.7).

6.2. A simple grammar

This experiment was to see if the KR algorithm could induce the grammar a^*b^*, (zero or more a's followed by zero or more b's). The algorithm requires a small integer value, k, to be given to it to tell it how much generalisation is necessary. Generally the smaller k is, the more compact its guess. In this experiment $k = 1$ since $k = 0$ leads to an over-generalisation, the automaton having only a single state.

Given the set of sentences

$$S^+ = \{ab,bb,aab,abb\}$$

the KR algorithm infers the automaton which is shown as a state transition table in figure 6.1.

τ is merely a termination symbol. Thus *state 2* is shown to be an acceptor state by the fact that a termination symbol can be accepted. *State 0* is the start state of the automaton. As in Mugol (see chapter 3), the unique goal state has no outgoing arcs. In descriptions of automata given in later sections of this chapter the symbols are situation/action pairs, and in any system

Present State	Input symbol	Next State
0	a	1
0	b	2
1	a	1
1	b	2
2	b	2
2	τ	GOAL

Figure 6.1 Induced state transition table for sentences {ab,bb,aab,abb}

producing state example information for Mugmaker, τ would be given by the user as the situation/action pair used when control is returned from the Mugol module.

The automaton shown above, is not the target language a^*b^* (0 or more a's followed by 0 or more b's). In order to get the algorithm to find a^*b^* it is necessary to give it the strings a and λ (empty sentence) in addition to the sentences provided. Thus the sample sentences given to the algorithm would be

$$S^+ = \{\lambda, a, ab, bb, aab, abb\}$$

The resultant automaton is shown as a state transition table in figure 6.2 and as a state transition diagram in figure 6.3.

This represents the desired automaton, although it is not minimal. Minimisation of automata is a well understood process, and a standard algorithm could be used for this purpose.

Present State	Input symbol	Next State
0	a	1
0	b	2
0	τ	GOAL
1	a	1
1	b	2
1	τ	GOAL
2	b	2
2	τ	GOAL

Figure 6.2 Induced state transition table for sentences {λ,a,ab,bb,aab,abb}

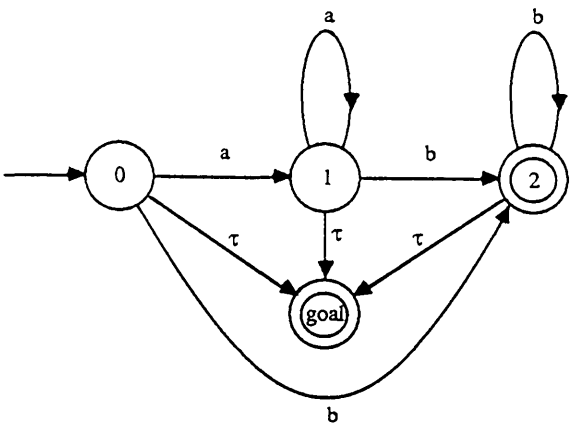


Figure 6.3 Diagrammatic representation of figure 6.2

6.3. 1 bit binary adder

In the next experiment we try to infer a 1 bit binary adder. Such a piece of circuitry can be produced automatically as a VLSI layout once the underlying finite state machine has been designed. The algorithm was used with the parameter setting of $k = 1$.

Figure 6.4 shows the sequences given, together with their binary sums.

Sequences are separated in the table by double lines. Instead of the input symbols used in the previous experiment, we have used situation/action tuples. The two binary numbers to be added are given in 1 bit situation pairs, the lower order bits being presented first. The result after each input pair is given as the action.

In figure 6.5 we give the algorithm's solution as a transition table.

Situation (Input1,Input2)	Action Output	Comment
		A null sequence is legal
(0,0)	0	0 + 0 = 0
(0,1)	1	0 + 1 = 1
(1,0)	1	1 + 0 = 1
(1,1) (0,0)	0 1	1 + 1 = 10
(1,1) (1,1) (0,0)	0 1 1	11 + 11 = 110
(1,1) (0,1) (0,0)	0 0 1	1 + 11 = 100
(1,1) (1,0) (0,0)	0 0 1	11 + 1 = 100

Figure 6.4 Example situation/action sequences describing 1 bit binary adder

Present State	Situation (Input1,Input2)	Action Output	Next State
0	τ	NULL	GOAL
0	(0,0)	0	0
0	(0,1)	1	0
0	(1,0)	1	0
0	(1,1)	0	1
1	(0,0)	1	0
1	(0,1)	0	1
1	(1,0)	0	1
1	(1,1)	1	1

Figure 6.5 Inductively generated state transition table for a 1 bit binary adder

This solution is complete and correct. The two states correspond to the carry and non-carry states. Thus from 7 example sums, the algorithm found a solution to an indefinite precision adder.

6.4. Traffic light controller

This example came from the book "Introduction to VLSI systems" by Mead and Conway. The book is a standard reference book for VLSI technology and contains an example of a finite state circuit for controlling traffic. Here is some of the description of the problem taken directly from the book.

The following simple example will help to illustrate the basic concepts of finite-state machines and their implementations in *n*MOS circuitry. A busy highway is intersected by a little-used farmroad. Detectors are installed that cause the signal C to go *high* in the presence of a car or cars on the farmroad... We wish to control traffic lights at the intersection, so that in the absence of any cars waiting to cross or turn left on the highway from the farmroad, the highway lights will remain green. If any cars are detected ..., we wish the highway lights to cycle through caution to red and the farmroad lights then to turn green. The farmroad lights are to remain green only while the detectors signal the presence of a car or cars, but never longer than some fraction of a minute. The farmroad lights are then to cycle through caution to red and the highway lights then to turn green. The highway lights are not to be interruptible again by the farmroad traffic until some fraction of a minute has passed.

Figure 6.6 shows the meanings of actions used in figures 6.7 and 6.8. Figure 6.7 shows sequences given to the algorithm for this problem. It is assumed that the problem starts with the highway traffic lights being green.

The symbols *y* and *n* stand for *yes* and *no* respectively. The '-' symbol indicates that *any* non-clashing value of the attribute can be taken at this point.

Abbreviation	Meaning
wait	Null action
ST + HY	Start the timer and turn the highway lights yellow
ST + HR + FG	Start the timer, turn the highway lights red and turn the farmroad lights green
ST + FY	Start the timer and turn the farmroad lights yellow
ST + FR + HG	Start the timer, turn the farmroad lights red and turn the highway lights green

Figure 6.6 Meanings of action abbreviations used in figures 6.7 and 6.8

Situation (Farmroad cars, Long time-out, Short time-out)	Action	Comment
		Null sequence acceptable
(n, -, -)	wait	Waiting for farmroad cars
(y, y, -) (-, -, y) (-, -, y) (-, -, y)	ST + HY ST + HR + FG ST + FY ST + FR + HG	A complete cycle of changing the lights with no waiting
(y, y, -) (-, -, n) (-, -, y) (-, -, y) (-, -, y)	ST + HY wait ST + HR + FG ST + FY ST + FR + HG	A complete cycle of changing the lights with one wait
(y, y, -) (-, -, n) (-, -, y) (y, n, n) (-, -, y) (-, -, y)	ST + HY wait ST + HR + FG wait ST + FY ST + FR + HG	A complete cycle of changing the lights with two waits
(y, y, -) (-, -, n) (-, -, y) (y, n, n) (-, -, y) (-, -, n) (-, -, y)	ST + HY wait ST + HR + FG wait ST + FY wait ST + FR + HG	A complete cycle of changing the lights with three waits

Figure 6.7 Situation/action sequences describing a traffic light controller

Figure 6.8 shows the transition table of the automaton produced with parameter setting k=1. Again the automaton is complete and correct according to the book. The states correspond to

0) Highway lights are green. Traffic is travelling along the main highway.

1) Highway lights have changed to yellow. The timer has been started and the automaton is waiting for the short timeout.

2) Highway lights have turned yellow. The farmroad lights are green. the timer has been restarted. The automaton is waiting for either the long timeout or for cars to stop flowing along the farmroad.

3) The farmroad lights have turned yellow. The timer has been restarted again and the automaton is waiting for the short timeout.

It is interesting to note that the authors of the book from which this example was taken, in order to show how the automaton works, describe it in terms of example sequences of events.

Present State	Situation (Farmroad cars, Long time-out, Stime-out)	Action	Next State
0	τ	NULL	GOAL
0	(n, -, -)	wait	0
0	(y, y, -)	ST + HY	1
1	(-, -, y)	ST + HR + FG	2
1	(-, -, n)	wait	1
2	(-, -, y)	ST + FY	3
2	(y, n, n)	wait	2
3	(-, -, y)	ST + FR + HG	0
3	(-, -, n)	wait	3

Figure 6.8 Induced state transition table for traffic light controller

6.5. Reverse motor problem

This problem is conceptually very simple. A large electric motor is controlled by three buttons. The buttons are marked "left", "right" and "stop". When the motor is not moving, pressing the left button will cause the motor to start turning to the left, the right button to the right. The only complication is that if the motor is turning in a particular direction, any attempt to force it to turn in the opposite direction will cause the motor to stop, rather than immediately changing polarity on the motor. This is necessary due to the momentum of the motor. Obviously the stop button is used to stop the motor if it is turning in either direction. It is assumed that the motor starts not turning in either direction. Figure 6.9 shows the sequences given to the algorithm. Figure 6.10 shows the transition table of the automaton produced with parameter setting k=1.

As with the grammar example, using a k setting of 1, the automaton, although correct, is not minimal. The reader may notice that state(0) and state(1) are identical and should have been merged. It is generally true that the automata produced are only minimal if the k setting is 0. States 0 and 1 correspond to the motor being stopped, states 2 and 3 correspond to the motor turning left and right respectively.

6.6. Algebra problem

For this experiment various different solutions of simple linear equations were presented as sequences of situation/action tuples. The same problem was tackled previously by Andrew Paterson using the static induction package ACLS (Paterson, 1984) (ACLS is based on ID3 (Quinlan, 1979)). The sequences are presented along with the automaton produced. Whereas Paterson used 7 attributes, it was found that only 4 attributes were needed when using the grammatical induction algorithm. Figure 6.11 shows the meanings of situational attributes used in figures 6.13 and 6.14, the table of sequences. Figure 6.12 gives the meanings of the actions used in figures 6.13 and 6.14.

Figure 6.14 shows the transition table of the automaton produced with parameter setting $k = 1$.

Situation Button pushed	Action	Comment
		Null sequence acceptable
-	motor stop	Pushing the stop button leaves the motor at rest
- -	motor stop motor stop	It does not matter how many times it is pushed
left -	motor left motor stop	Any button other than left stops the motor when it is turning left
left left -	motor left motor left motor stop	Pushing left when it is turning left keeps it turning left
- left -	motor stop motor left motor stop	Similar
right -	motor right motor stop	Any button other than right stops the motor when it is turning right
right right -	motor right motor right motor stop	Pushing right when it is turning right keeps it turning right
- right -	motor stop motor right motor stop	Similar

Figure 6.9 Situation/action sequences describing a motor controller

Present State	Situation Button pushed	Action	Next State
0	τ	NULL	GOAL
0	-	motor stop	1
0	left	motor left	2
0	right	motor right	3
1	τ	NULL	GOAL
1	-	motor stop	1
1	left	motor left	2
1	right	motor right	3
2	-	motor stop	1
2	left	motor left	2
3	-	motor stop	1
3	right	motor right	3

Figure 6.10 Induced state transition table for motor controller

Attribute	Meaning
Brackets	The equation contains at least one bracketed term
X on the right	There is a term in x on the right-hand side of the equation
Const on left	There is a constant term on the left-hand side of the equation
Similar terms	Either side of the equation contains two or more constants or terms in x
Ok	There is a single term in x on the left

Figure 6.11 Meanings of situational attributes used in figures 6.13 and 6.14

Action	Meaning
Divide both	Divide through both sides of the equation by the coefficient of x
Add similar	Add together any similar terms (see 'Similar terms' in figure 6.11)
Multiply brackets	Multiply out any bracketed term by its coefficient
X to left	Move a term in x from the right to the left of the equation
Const to right	Move a constant term from the left to the right of the equation

Figure 6.12 Meanings of actions used in figures 6.13 and 6.14

Situation (Brackets,X on the right, Const on left,Similar terms,Ok)	Action	Equation
(n,n,n,n,y)	Divide both	3x = 6 x = 2
(n,n,n,y,-) (n,n,n,n,y)	Add similar Divide both	3x + 4x = 6 7x = 6 x = 6/7
(n,n,n,y,-) (n,n,n,y,-) (n,n,n,n,y)	Add similar Add similar Divide both	3x + 4x + 5x = 6 7x + 5x = 6 12x = 6 x = 1/2
(y,-,-,-,-) (n,n,n,n,y)	Multiply brackets Divide both	5(3x) = 7 15x = 7 x = 7/15
(y,-,-,-,-) (n,n,n,y,-) (n,n,n,n,y)	Multiply brackets Add similar Divide both	5(3x + 4x) = 7 15x + 20x = 7 35x = 7 x = 1/5
(y,-,-,-,-) (y,-,-,-,-) (n,n,n,y,-) (n,n,n,n,y)	Multiply brackets Multiply brackets Add similar Divide both	5(3x) + 6(4x) = 7 15x + 6(4x) = 7 15x + 24x = 7 39x = 7 x = 7/39
(y,-,-,-,-) (n,y,-,-,-) (n,n,n,y,-) (n,n,n,n,y)	Multiply brackets X to left Add similar Divide both	5(3x) = 2x + 7 15x = 2x + 7 15x - 2x = 7 13x = 7 x = 7/13

Figure 6.13 continued over page

Figure 6.13 (contd)

Situation (Brackets,X on the right, Const on left,Similar terms,Ok)	Action	Equation
(y,-,-,-,-)	Multiply brackets	5(3x) + 7 = 2
(n,n,y,-,-)	C to right	15x + 7 = 2
(n,n,n,y,-)	Add similar	15x = 2 - 7
(n,n,n,n,y)	Divide both	15x = -5
		x = -1/3
(n,y,-,-,-)	X to left	5x = 7 + 2x
(n,n,n,y,-)	Add similar	5x - 2x = 7
(n,n,n,n,y)	Divide both	3x = 7
		x = 3/7
(n,y,-,-,-)	X to left	5x = 7 + 4x + 3x
(n,y,-,-,-)	X to left	5x - 4x = 7 + 3x
(n,n,n,y,-)	Add similar	5x - 4x - 3x = 7
(n,n,n,y,-)	Add similar	x - 3x = 7
(n,n,n,n,y)	Divide both	-2x = 7
		x = -2/7
(n,y,-,-,-)	X to left	5x + 7 = 3x + 5
(n,n,y,-,-)	C to right	5x - 3x + 7 = 5
(n,n,n,y,-)	Add similar	5x - 3x = 5 - 7
(n,n,n,y,-)	Add similar	8x = 5 - 7
(n,n,n,n,y)	Divide both	8x = -2
		x = -1/4
(n,n,y,-,-)	C to right	5x + 2 = 7
(n,n,n,y,-)	Add similar	5x = 7 - 2
(n,n,n,n,y)	Divide both	5x = 5
		x = 1
(n,n,y,-,-)	C to right	5x + 2 + 3 = 7
(n,n,y,-,-)	C to right	5x + 3 = 7 - 2
(n,n,n,y,-)	Add similar	5x = 7 - 2 - 3
(n,n,n,y,-)	Add similar	5x = 5 - 3
(n,n,n,n,y)	Divide both	5x = 2
		x = 2/5

Figure 6.13 Situation/action sequences describing algebraic equation solver

Present State	Situation (Brackets,X on the right, Const on left,Similar terms,Ok)	Action	Next State
0	(n,n,n,n,y)	Divide both	3
0	(n,n,n,y,-)	Add similar	4
0	(y,-,-,-,-)	Multiply brackets	1
0	(n,y,-,-,-)	X to left	5
0	(n,n,y,-,-)	C to right	2
1	(n,n,n,n,y)	Divide both	3
1	(n,n,n,y,-)	Add similar	4
1	(y,-,-,-,-)	Multiply brackets	1
1	(n,y,-,-,-)	X to left	5
1	(n,n,y,-,-)	C to right	2
2	(n,n,n,y,-)	Add similar	4
2	(n,n,y,-,-)	C to right	2
3	τ	NULL	GOAL
4	(n,n,n,n,y)	Divide both	3
4	(n,n,n,y,-)	Add similar	4
5	(n,n,n,y,-)	Add similar	4
5	(n,y,-,-,-)	X to left	5
5	(n,n,y,-,-)	C to right	2

Figure 6.14 Inductively generated state transition table for the equation solver

Again state 0 and 1 should be the same state. States 0 and 1 deal with repetitively multiplying out the brackets. State 5 then repetitively moves all "x" terms to the left hand side. State 2 repetitively moves all "constant" terms from the right hand side of the equation to the left. State 4 repetitively adds up similar terms and divides both sides through by the divisor of "x". It should be obvious that by dividing the task up into these smaller tasks, the user will not need to be asked as many questions when executing the automaton (given that it is being done interactively), as in each context it can be assumed that the jobs of the preceding contexts have been carried out satisfactorily. In order for this saving however, much more example information needed to be given than in Paterson's solution, which only required 9 ACLS examples.

6.7. Hanging pictures in a room

This last example is more typical of the usual situation/action problems posed for robot-like worlds. The problem is as follows. A robot is in a room which contains a door and some pictures placed on the floor against walls on which they should be hung. The robot can start off facing in any direction and must hang all the pictures on the appropriate walls. The robot is able to see objects and knows its position (either at a wall or 'other'). The robot uses the ability to see the door to make sure it has hung all the pictures before stopping. It is able to 'turn', which involves rotating in a clockwise direction until its situation vector changes in some way. It can also move forward, again until the situation vector changes. Sub-problems such as actually hanging the picture on the chosen wall could have been developed as individual, simple automata. Figure 6.15 gives the meanings of the actions used in figures 6.16 and 6.17. Figure 6.16 shows sequences given to the algorithm. Figure 6.17 shows the transition table of the automaton produced with parameter setting $k = 0$.

Action	Meaning
Forward	Keep moving forward until the situational vector changes
Turn	Keep turning clockwise until the situational vector changes
Hang picture	Hang the picture which is on the floor on the wall
Lie down	Lie down on the ground

Figure 6.15 Meanings of actions used in figures 6.16 and 6.17

Situation (See, At)	Action	Comment
(door, -) (other, -) (door, -)	Turn Turn Lie down	All pictures hung, start at the door
(other, -) (door, -) (other, -) (door, -)	Turn Turn Turn Lie down	All pictures hung, start elsewhere
(door, -) (other, -) (picture on the floor, -) (picture on the floor, wall) (other, wall) (other, -) (door, -)	Turn Turn Forward Hang picture Turn Turn Lie down	1 picture to hang start at door
(other, -) (picture on the floor, -) (picture on the floor, wall) (other, wall) (door, -) (other, -) (door, -)	Turn Forward Hang picture Turn Turn Turn Lie down	Start elsewhere

Figure 6.16 Situation/action sequences describing the robots actions

Present State	Situation (See, At)	Action	Next State
0	(door, -)	Turn	1
0	(other, -)	Turn	0
0	(picture on the floor, -)	Forward	6
1	(other, -)	Turn	2
2	(door, -)	Lie down	3
2	(picture on the floor, -)	Forward	4
3	τ	NULL	GOAL
4	(picture on the floor, wall)	Hang picture	5
5	(other, wall)	Turn	1
6	(picture on the floor, wall)	Hang picture	7
7	(other, wall)	Turn	0

Figure 6.17 Inductively generated state transition table for the robot controller

The states have an interpretation as follows

0) The robot starts in this state and must decide what line of action is appropriate.

a) If the robot sees the door, it will turn, and go to state 1 from which it will do a single pass around the room looking for pictures until it sees the door again.

b) If it sees something other than the door or a picture on the floor, it turns in order to find one of these two.

c) If it sees a picture on the floor it will move forward to the picture, go to state 6 and proceed by hanging the picture and returning to state 0.

1) Having entered state 1, the robot must have seen the door at least once. Thus it is only

necessary to mop up all remaining pictures and keep turning until it sees the door. This will be done in either repeating the sequence state 1 - state 2 - state 4 - state 5 - state 1, or by seeing the door in state 1 and stopping.

2) This state is part of the loop described for state 1, and contains the terminating condition that the door can be seen.

3) This state merely terminates the module unconditionally.

4) This state is part of the loop starting in state 1.

5) This state is part of the loop starting in state 1.

6) This state is part of the loop starting in state 0.

7) This state is part of the loop starting in state 0.

Dufay and Latombe (1984) describe a similar method of automatically programming robots. They use an inductive algorithm which is essentially the same as that of Miclet (1980) (see section 5.4.5.3). In their system, low-level robot sequences are generated by a planner and fed into an inductive algorithm. The resultant generalised finite state automaton is represented in a robot programming language for execution. The robot program contains not only manipulator directives but also tests to be carried out on the world state.

6.8. Conclusion

The problems described in sections 6.3 - 6.5 are conceptually different from those in sections 6.6 - 6.7. The difference lies in the fact that whereas in the first three, a particular world situation is assumed for the start state (eg. the highway lights start off being green in the traffic light example) the latter problems make no such assumptions (eg. the robot can start anywhere in the room, facing in any direction). Although the first examples could be developed with this "any situation starts" approach, it seems not typical as a whole of problems occurring in engineering. It is also interesting to note that whereas each problem is fairly difficult, it was automatically broken

into a number of smaller and simpler problems.

Angluin's k-reversible algorithm (section 5.4.5.4) seems to be very powerful, and capable of dealing with building complex automata from a skimpy presentation of sample sequences. Moreover our efficient version of this algorithm (section 7.2) runs at quite acceptable speeds, typically around 10 - 20 seconds for the automata presented in this chapter.

As stated in section 6.7, the method of constructing robot plans from example sequences has also been investigated by Dufay and Latombe (1984). However, they used a simpler inductive algorithm, essentially the same as that described by Miclet (5.4.5.3). Angluin (1982b) has shown Miclet's algorithm to be merely a special case of *k*-reversible induction. We therefore conclude that our method has a wider scope than that of Dufay and Latombe.

7

New sequence induction theory

Abstract. A new algorithm implements Angluin's k-reversible induction in time O(n) rather than Angluin's time O(n^3). The new algorithm is shown to identify k-reversible languages in the limit and can be modified to use negative data rather than a k value. We also propose and demonstrate a new method of grammatical induction called k-contextual induction. This has the advantage over all other methods in the literature of being capable of generating natural solutions from samples as small as a single example. Next we show how these algorithms can be used to generate control structures for Mugol modules in the form of Mealy machines.

7.1. Introduction

In this chapter we describe a new approach to the automatic construction of control strategies from example material. Our intention is to build control or strategic expert systems from example sequences. Each element of the sequence is a static example of the ID3 (Quinlan, 1979) variety. The output of the inductive process is a finite state structure in which each state contains a small number of the static examples. These can in turn be used by ID3-like induction schemes to produce rules or decision trees for each state. *Thus although we do not produce a hierarchical structure, we achieve some of the aims of structured induction (i.e. a set of small understandable rules) by using example material which contains additional structural information within each example.*

The basis for these techniques lies in the study of grammatical induction, that is the inference of grammatical structures from example sentences of a language (see chapter 5). The grammar produced can be viewed as the control structure of a program which generated the example sentences. As explained in chapter 5, some of the earliest work in this area was done by Biermann

and Feldman (Biermann and Feldman, 1972) who devised an algorithm to induce finite state automata from strings of a language. Although their algorithm was capable of finding any regular language given a sufficient example set, the algorithm requires an arbitrary complexity parameter. Angluin (Angluin, 1982b) has described an algorithm which infers only a limited subset of the regular languages. This subset she calls the *k*-reversible languages. By limiting the target language class, Angluin's algorithm is capable of finding the correct language using fewer examples than Biermann and Feldman's algorithm.

The author has taken Angluin's algorithm and redesigned it to run with linear time complexity rather than Angluin's original $O(n^3)$ time (see section 7.2). Furthermore, we have discovered an even smaller, but useful subset of the *k*-reversible languages, which we call the *k*-contextual languages (section 7.3). The algorithm for inferring members of the *k*-contextual languages requires even fewer examples to infer any particular *k*-contextual language than Angluin's, to the extent that sensible inference is possible from samples containing only a single example. All other methods in the literature (Angluin, 1982b; Biermann and Feldman, 1972; Levine, 1982; Miclet, 1980) presuppose more than a single example.

7.2. An efficient algorithm for induction of k-reversible languages

In section 5.4.6.1 we introduced Angluin's k-reversible algorithm. This algorithm has time complexity $O(n^3)$. In this section we describe a new algorithm, *KR*, which carries out Angluin's *k*-reversible induction in time $O(n)$. The definitions given in section 5.4.1 are assumed as precursors to the following discussion.

7.2.1. Uniquely terminated acceptors

Let the finite state acceptor (*FSA*) A be described by the n-tuple $A = (Q, \Sigma, \delta, I, F)$. We say that A is a *τ–terminated acceptor* (*TTA*), (where $\tau \in \Sigma$ is a unique termination symbol) if and only if for any state $q \in Q$, $\delta(q, \tau) = q'$ implies $q' \in F$. Otherwise $\delta(q, \tau) = \varnothing$. i.e. we call a finite state acceptor a *TTA* if it has the property that any transition arc is labelled with the termination

symbol τ if and only if it leads into an acceptor state. It should be clear that any string w accepted by a *TTA* will have the form $w = u\tau v$ if and only if v is the empty string, λ. i.e. the symbol τ can only be found as the last symbol of w.

The acceptor A is a *goal state acceptor* (*GSA*) if and only if it has a single accepting state q_g (called the *goal* state) and the set of states reached by a single transition from q_g, is empty. In other words, a *GSA* has a unique *goal* state which has no outgoing arcs.

We call any acceptor that is both a *TTA* and a *GSA*, *uniquely terminated.*

Theorem 7.1 There exists a bijection h_τ such that for any acceptor $A = \{Q, \Sigma, \delta, I, F\}$ in which $\tau \notin \Sigma$, $h_\tau(A)$ is a *uniquely terminated* acceptor that accepts the language $L(A).\{\tau\}$.

Proof. The mapping function h_τ is as follows. Let $A = (Q, \Sigma, \delta, I, F)$. Now we construct the uniquely terminated acceptor $A_u = h_\tau(A) = (Q_u, \Sigma_u, \delta_u, I_u, F_u)$ with $Q_u = Q \cup \{q_g\}$ (where q_g is the *goal* state), $\Sigma_u = \Sigma \cup \tau$, $\delta_u(q_f, \tau) = \{q_g\}$ for all $q_f \in F$, $\delta_u(q, b) = \delta(q,b)$ for all $q \in Q$, $b \in \Sigma$ and $\delta_u(q_g, b') = \varnothing$, $b' \in \Sigma_u$, $I_u = I$, $F = \{q_g\}$. Clearly A_u is a *TTA* since all arcs leading to q_g are labelled with τ. It is also a *GSA* since q_g is unique and $\delta_u(q_g, b') = \varnothing$, $b' \in \Sigma_u$.

In order to show that h_τ is a bijection, we need to prove the existence of the inverse function h_τ^{-1}. Let A_u be a uniquely terminated acceptor $A_u = (Q_u, \Sigma_u, \delta_u, I_u, F_u)$ where $F_u = \{q_g\}$. Now we construct the finite state acceptor $A = h_\tau^{-1}(A_u) = (Q, \Sigma, \delta, I, F)$ with $Q = Q_u - \{q_g\}$, $\Sigma = \Sigma_u - \tau$, $\delta(q, \tau) = \varnothing$ for all $q \in Q$, $\delta(q, b) = \delta_u(q,b)$ for all $q \in Q - F$, $\delta(q_f, b) = \delta_u(q_f,b)$ for all $q_f \in F$ where $b \in \Sigma$. $I = I_u$, $F = \{q: q \in Q_u, \delta_u(q,\tau) = \{q_g\}\}$. Given that A_u is uniquely terminated, clearly A is by definition a finite state acceptor since it is fully specified and does not accept any symbols other than those of Σ. QED.

We say that an acceptor A is *ku*-reversible if and only if

1) A is *uniquely terminated* and

2) $h_\tau^{-1}(A)$ is k-reversible (see definition of k-reversibility in section 5.4.5.4).

7.2.2. The KR algorithm

The following algorithm constructs a *ku*-reversible acceptor by augmenting the sample set S^+ to $S^+.\{\tau\}$ and using a process similar to that of Angluin's ZR algorithm. The final result is normalised to being a k-reversible acceptor using h_τ^{-1}.

Algorithm KR

Input: a nonempty positive sample S^+ and a parameter k.

Output: a k-reversible acceptor A.

* Initialisation
Let S_u^+ be $S^+.\{\tau\}$
Let $A_0 = (Q_0, \Sigma_0, \delta_0, I_0, F_0)$ be $PT(S_u^+)$.
Let π_0 be the trivial partition of Q_0.
For each $b \in \Sigma_0$ and $q \in Q_0$ let $s(\{q\},b) = \delta_0(q,b)$ and $p(\{q\},b) = \delta_0'(q,b)$.
Choose some $q' \in F_0$.
Let LIST contain all pairs (q',q) such that $q \in F_0 - \{q'\}$.
Let $i = 0$.

* Merging
While LIST $\neq \emptyset$ do
begin
  Remove some element (q_1,q_2) from LIST.
  Let $B_1 = B(q_1,\pi_i), B_2 = B(q_2,\pi_i)$.
  If $B_1 \neq B_2$ then
  begin
    Let B_3 be B_1 and B_2 merged.
    Let π_{i+1} be π_i with B_3 replacing B_1 and B_2.
    For each $b \in \Sigma_0$, s-UPDATE(B_1,B_2,B_3,b) and pk-UPDATE(B_1,B_2,B_3,b,k).
    Increase i by 1.
  end
end

*Termination
Let $f = i$
Output $h_\tau^{-1}(A_0/\pi_f)$.

Although s-UPDATE remains the same as that described by Angluin (Angluin, 1982b), we include it here for the sake of completeness.

Algorithm s-UPDATE

Input: blocks B_1, B_2 and B_3, and a symbol $b \in \Sigma_0$.

If $s(B_1,b)$ and (B_2,b) are nonempty then
begin
  Place $s((B_1,b),(B_2,b))$ on LIST.
end

If $s(B_1,b)$ is nonempty
  then let $p(B_3,b) = p(B_1,b)$
  else let $p(B_3,b) = p(B_2,b)$.

Angluin's *p*-UPDATE is replaced by *pk*-UPDATE which is described below.

Algorithm pk-UPDATE

Input: blocks B_1,B_2 and B_3, a symbol $b \in \Sigma_0$ and a k parameter.

For each $q_1 \in p(B_1,b)$ and $q_2 \in p(B_2,b)$
begin
  If q_1 and q_2 have a common k-leader in A_0/π_i then (1)
  begin
    Place (q_1,q_2) on LIST.
  end
end

If p(B_1,b) is nonempty
  then let $p(B_3,b) = p(B_1,b)$
  else let $p(B_3,b) = p(B_2,b)$.

Lemma 7.2. Let S^+ be a non-empty positive sample, k a non-negative integer, and π_i the partition formed by KR on input S^+ and k after i steps. If some u_1v_1 and u_2v_2 are in the same non-goal block B of π_i (i.e. $B(u_1v_1,\pi_i) = B(u_2v_2,\pi_i)$, $v_1 \neq w_1\tau$, $v_2 \neq w_2\tau$), where $|v_1| = k = |v_2|$, then $v_1 = v_2$.

Proof. From inspection of KR, two strings u_1v_1 and u_2v_2 are in the same block B of π_i only if at some step j, previous to i, $B(u_1v_1,\pi_j)$ was merged with $B(u_2v_2,\pi_j)$. Let the pair (q_1,q_2) be the pair of states representing $B(u_1v_1,\pi_{j'})$ and $B(u_2v_2,\pi_{j'})$, placed on LIST during some step j', previous to j. (q_1,q_2) can have been placed on LIST only either

a) during initialisation, in which case v_1 and v_2 are terminated by a τ symbol (i.e. $v_1 = w_1\tau$, $v_2 = w_2\tau$ and are within a goal block B_g of π_i). However, Lemma 7.2 only applies to non-goal blocks *or*

b) by *pk*-UPDATE. *pk*-UPDATE would only merge q_1 and q_2 if they had a common k-leader in A_0, i.e. $v = v_1 = v_2$ and $|v| = k$ *or*

c) by *s*-UPDATE. As A_0 is $PT(S_u^+)$, which is by definition deterministic, *s*-UPDATE would only merge q_1 and q_2 if they were both *b*-successors ($b \in \Sigma_0 - \{\tau\}$) of some state q_3. Also, as A_0 is deterministic, q_3 must have been formed by a similar chain of 0 or more merges by *s*-UPDATE preceded by a *pk*-UPDATE. Thus all strings leading into q_1 and q_2 must have a common tail of at least length *k*. QED.

The condition that q_1 and q_2 have a common *k*-leader in A_0/π_i from statement (1) of the algorithm *pk*-UPDATE can be computed efficiently and simply as follows. Let q_1 and q_2 correspond to $B(u_1v_1,\pi_i)$ and $B(u_2v_2,\pi_i)$, where u_1v_1, $u_2v_2 \in S^+$. To check whether q_1 and q_2 have a common *k*-leader in A_0/π_i, we need merely check that $|v_1| = k = |v_2|$ and $v_1 = v_2$. It can be seen from Lemma 7.2 that it does not matter which u_1v_1 and u_2v_2 are taken as representatives of the two blocks.

Lemma 7.3. Let S^+ be a non-empty positive sample and *k* a non-negative integer. The output of algorithm *KR* on input S^+ and *k* is isomorphic to the prefix tree acceptor $PT(S^+)$ whenever *k* is greater than the length of the longest string within S^+.

Proof. Let π_f be the partition formed by *KR* on input S^+ and *k*, and let $u_1v_1w_1$ and $u_2v_2w_2$ be two members of S^+. During initialisation A_0 is set to be $PT(S_u^+)$ where S_u^+ is $S^+.\{\tau\}$. By Lemma 7.2 u_1v and u_2v are only within the same non-goal block *B* of π_f when $|v_1| = k = |v_2|$ and $v_1 = v_2$. However, since *k* is greater than the longest string within S^+, there can exist no such substrings v_1 and v_2 of length *k*. Thus no non-goal state of A_0 will be merged. However, all and only goal states are placed on LIST during initialisation, thus all such goal states are merged into a single goal state. Therefore the output of *KR*, $h_\tau^{-1}(A_0/\pi_f)$ must be isomorphic to $PT(S^+)$ by the definition of h_u^{-1} (proof of 7.1). QED.

7.2.3. The correctness of KR

Angluin (Angluin, 1982b) describes an algorithm, *k*-RI, for inducing *k*-reversible languages which repetitively merges any two blocks $B(q_1,\pi_i)$ and $B(q_2,\pi_i)$ from successive partitions π_i of the

original prefix tree $PT(S^+)$ if and only if they violate the conditions of k-reversibility. We now define the conditions of ku-reversibility in a similar manner to those of Angluin's.

1) No two arcs labelled with a common symbol b ($b \in \Sigma_0$) leading out from any state q_3 lead to any other two states q_1 and q_2. i.e. a ku-reversible acceptor is deterministic.

2) Given that there exists two paths labelled with a common string u of length k leading to two states q_1 and q_2, there must not also be two arcs labelled with a common symbol b ($b \in \Sigma_0$) leading from q_1 and q_2 to some other state q_3.

If either of these two conditions is present, then the states q_1 and q_2 mentioned should be merged. Diagrammatically we can represent the conditions as those shown in figure 7.1.

Lemma 7.4. Let S^+ be a non-empty positive sample, k a non-negative integer, A_0 the prefix tree acceptor of S^+_u, and π_f the final partition found by KR on input S^+. Then π_f is the finest partition of the states of A_0 such that A_0/π_f is ku-reversible.

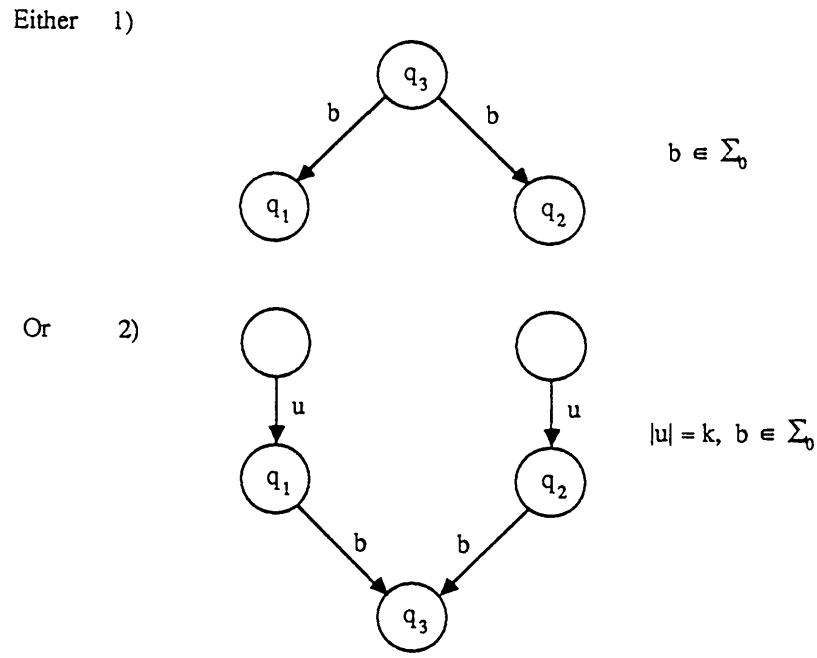


Figure 7.1 Graphical representation of conditions for merger of q_1 and q_2

Proof. If the pair (q_1,q_2) is ever placed on LIST, then q_1 and q_2 must be in the same block of the final partition, that is, $B(q_1,\pi_f) = B(q_2,\pi_f)$. Thus in order to prove that KR always produces a *ku*-reversible acceptor, it suffices to show that two states q_1 and q_2 are always placed on LIST if and only if they violate the conditions of *ku*-reversibility. From inspection of KR, it can be seen that (q_1,q_2) can have been placed on LIST only either

a) during initialisation.

   i) This corresponds to all those occurrences of condition 2) (figure 7.1) in which $b = \tau$.

   ii) Owing to the initialisation of LIST all occurrences of condition 2) (figure 7.1) in

which $b = \tau$ will be found and merged.

b) by *pk*-UPDATE.

i) This corresponds to all those occurrences of condition 2) (figure 7.1) in which $b \neq \tau$.

ii) As A_0 has the graphical form of a tree (each state has a maximum of one arc leading into it), and condition 2) depicts a graph containing a state (q_3) with two arcs leading into it, q_3 must have been formed as the product of a merger. Following this merger, q_1 and q_2 would have been placed on LIST. Thus such conditions will always be found.

c) by *s*-UPDATE.

i) This corresponds to all those occurrences of condition 1) (figure 7.1).

ii) Since A_0 is deterministic, the state q_3 depicted in condition 1) of figure 7.1 must have been formed as the product of a merger. Following this merger, q_1 and q_2 would have been placed on LIST by *s*-UPDATE. Again, such conditions will always be found.

We have shown that the states (q_1,q_2) will be merged in cases a-c i) *only if* the conditions of *ku*-reversibility are violated. Also we have shown in all cases a-c ii) that (q_1,q_2) are always placed on LIST *if* the conditions of *ku*-reversibility are violated. Thus $A_u = A_0/\pi_f$ is *ku*-reversible.

It remains to show that if π is any partition of Q_0 such that A_0/π is *ku*-reversible then π_f refines π. We prove by induction that π_i refines π for $i = 0,1,..f$. Clearly π_0 refines π. Suppose π_0, π_1, .. π_i all refine π and π_{i+1} is obtained from π_i in the course of processing (q_1,q_2) for LIST. Since π_i refines π, $B(q_1,\pi_i)$ is a subset of $B(q_1,\pi)$ and $B(q_2,\pi_i)$ is a subset of $B(q_2,\pi)$, so to show that π_{i+1} refines π, it suffices to show that $B(q_1,\pi) = B(q_2,\pi)$.

Either (q_1,q_2) was first placed on LIST during the initialisation stage or not. If so, then q_1 and q_2 are both accepting states, and since A_0/π is *ku*-reversible and thus by definition is a *GSA*, it has only one accepting state, so $B(q_1,\pi) = B(q_2,\pi)$. Otherwise, (q_1,q_2) was first placed on LIST in consequence of some previous merge, let us say the merge to produce π_j from π_{j-1}, where $0 < j \leq i$. Then $(q_1,q_2) = (s(B_1,b),s(B_2,b))$ (resp. $(p(B_1,b),p(B_2,b))$) where B_1 and B_2 are the blocks of π_{j-1} merged in forming π_j and b is some symbol. Then q_1 and q_2 are b-successors (resp. b-predecessors) of two states in some block B of π_j. Since π_j refines π by the induction hypothesis, q_1 and q_2 are b-successors (resp. b-successors) of some block B' in π, and since A_0/π is *ku*-reversible, $B(q_1,\pi) = B(q_2,\pi)$. Thus in either case π_{i+1} refines π, and by induction we conclude that π_f refines π. QED.

Lemma 7.5. Let S^+ be a non-empty positive sample, k a non-negative integer, A_0 the prefix tree acceptor of S^+_u, π_f the final partition found by KR on input S^+ and k, and $A = h_\tau^{-1}(A_0/\pi_f)$ the output automata. Then A is isomorphic to the automata $A' = PT(S^+)/\pi$, where π is the finest partition of the states of $PT(S^+)$ such that A' is k-reversible.

Proof. From the definitions of k-reversibility and the mapping h_τ^{-1}, since A_0/π_f is *ku*-reversibility, it follows that $h_\tau^{-1}(A_0/\pi_f)$ is k-reversible.

It is necessary to show that if $B(w_1,\pi_f) = B(w_2,\pi_f)$ then $B(w_1,\pi) = B(w_2,\pi)$, $w \neq u\tau v$. Mergers made under condition (figure 7.1)

1) are the same

2) $b \in \Sigma_0 - \{\tau\}$ merges are for k-reversible reasons. $b = \tau$, q_1 and q_2 would be accepting states.

Thus merges are made in the same way as those for conditions of Angluin's k-reversibility for all states other than goal states. All and only necessary states are merged (proof of 7.4). Thus A is isomorphic to the automata $A' = PT(S^+)/\pi$, where π is the finest partition of the states of $PT(S^+)$ such that A' is k-reversible. QED.

We have thus shown that the KR algorithm is input/output equivalent to Angluin's algorithm (Angluin, 1982b).

Theorem 7.6. Let S^+ be a nonempty positive sample, k a natural number and let $A_u = h_\tau^{-1}(A_0/\pi_f)$ be the acceptor output by *KR* on input S^+ and k. Then $L(A)$ is the smallest k-reversible language containing S^+.

Proof. As *KR* is input/output equivalent to Angluin's algorithm, k-RI (Angluin, 1982b), and Angluin proves this to be true for k-RI, it clearly holds for *KR*.

Theorem 7.7. Let L be a nonempty k-reversible language and w_1, w_2, w_3, ... any positive presentation of L. On this input, the output A_1, A_2, A_3, ... of *KR* converges to $A(L)$ (i.e. *KR* identifies L in the limit).

Proof. As *KR* is input/output equivalent to Angluin's algorithm, k-RI (Angluin, 1982b), and Angluin also proves this to be true for k-RI, it clearly holds for *KR*.

7.2.4. Time complexity of KR

Theorem 7.8. Let S^+ be a non-empty positive sample, k a non-negative integer. The algorithm KR may be implemented to run in time $O(n)$ where n is $(\sum^{u \in S^+} |u|) + |S^+| + 1$.

Proof. During initialisation, S_u^+ is composed as $S^+.\{\tau\}$. Let $n = (\sum^{u\tau \in S_u^+} |u\tau|) + 1 = (\sum^{u \in S^+} |u|) + |S^+| + 1$. The prefix tree acceptor $A_0 = PT(S_u^+)$, which has exactly n states can be constructed in time $O(n)$. Similarly the time taken to output the final acceptor $h_\tau^{-1}(A_0/\pi_f)$ is $O(n)$. As A_0 is a tree, it contains $n - 1$ transition arcs and thus there are exactly $n - 1$ s and p relations. Blocks are merged if they are distinct, which can happen at most $n - 1$ times. Similarly s-UPDATE and pk-UPDATE can effectively merge a total maximum of $n - 2$ pairs of s and p relations respectively. Thus assuming block mergers and s and p mergers take constant time the time complexity of KR is $O(n)$. QED.

7.2.5. Updating a k-reversible guess

Angluin (Angluin, 1982b) shows how her ZR algorithm can be modified to have good incremental behaviour. We now demonstrate how the *KR* algorithm described here can be modified for the same ends. Given the *ku*-reversible automaton $A_u = A_0/\pi_f$ computed by *KR* on input S^+, and given a new string w, we may easily update A_u to be the *ku*-reversible acceptor computed by *KR* on input $S^{+\prime} = S^+ \cup \{w\}$. The method for doing this is to start at the initial state of A_u and follow the transitions A_f makes on the input string $w\tau$. If no undefined transitions are encountered and the last state reached is the *goal* state, then A_u already accepts $w\tau$ and nothing need be done. Otherwise, add new states and transitions for each symbol of w starting with the first undefined transition (if any). Mark the last state reached by $w\tau$ as accepting, and place the pair consisting of this state and the *goal* state of A_u on LIST. Continue the merging portion of the algorithm *KR* until LIST is empty, and output the *k*-reversible acceptor $h_\tau^{-1}(A_u/\pi')$, where π' is the final partition of the states of A_u. The correctness of this procedure is verified in the same way as that of the original algorithm *KR*, since the order of detecting and performing required merges is immaterial.

Example 7.9 If we run *KR* with a setting of $k = 0$ on the input {0,00,11,1100}, we obtain the acceptor shown in figure 7.2. If we then add the string 101 to the sample and perform the updating procedure just described, we first obtain the acceptor shown in figure 7.2b. This is then "folded up" as shown in Figure 7.2c and d to obtain as a final result an acceptor for strings with an even number of 1's.

7.2.6. Using negative data

Negative data can be used in the same way as that described by Angluin (Angluin, 1982b). That is, we are given a positive and negative example set (S^+,S^-), such that S^+ and S^- are disjoint finite sets of strings. We compute the *k*-reversible languages for $k = 0,1,2,...$ using the positive examples, S^+, until we find some k for which the inferred language does not contain any of the strings from the negative set S^-.

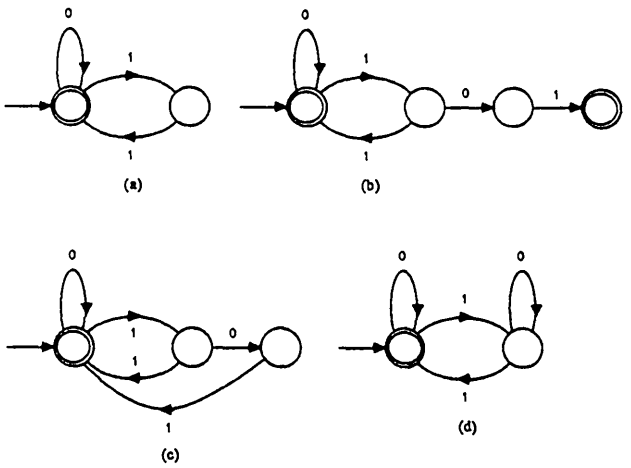


Figure 7.2 Updating a guess

7.3. k-contextual languages

According to theorem 5.4, all three of the algorithms reviewed in chapter 5 (Biermann and Feldman, 1972; Levine, 1982; Miclet, 1980) have the common property that they require at least two examples in order to carry out any generalisation. It can also be easily shown that Angluin's *k*-reversible method has exactly the same limitation. However, human beings have little difficulty in hypothesising grammars from sufficiently long single strings.

The *k*-contextual language class described in this section has the property that the smallest *k*-contextual language which is consistent with a single example may contain more than one string (see Example 7.17), i.e. algorithms which hypothesise *k*-contextual languages can carry out generalisation using only one example.

7.3.1. k-contextuality

First we give a language characterisation of k-contextual sets.

Definition 7.10. Let L be a regular language. Then L is k-contextual if and only if whenever u_1vw_1 and u_2vw_2 are in L and $|v| = k$, $T_L(u_1v) = T_L(u_2v)$.

We extend the notion of k-contextuality to cover not only languages but their corresponding acceptors.

Definition 7.11. An acceptor A is k-contextual if and only if $L(A)$ is k-contextual.

Remark 7.12. If a language L is k-contextual and contains two not necessarily distinct strings u_1vw_1 and u_2vw_2, where $|v| = k$, then L also contains u_1vw_2 and u_2vw_1. This is merely a particularisation of Definition 7.10.

Remark 7.13. Any 0-contextual language L containing two not necessarily distinct strings u_1w_1 and u_2w_2 also contains u_1w_2 and u_2w_1. This is a particularisation of Remark 7.12.

Lemma 7.14. Any 0-contextual non-empty language L is equal to Σ^* the *universal language* where $b \in \Sigma$ if and only if there is some $ubv \in L$.

Proof. Let L be a non-empty 0-contextual language. We prove by induction that $L = \Sigma^*$ where $b \in \Sigma$ if and only if there is some $ubv \in L$. Let w be an element of L. Since $w = \lambda.w = w.\lambda$ it follows from Remark 7.13 that λ is an element of L. By the inductive hypothesis we suppose that L contains all members of Σ^* of length less than or equal to n. Now suppose that $|ubv| = n + 1$. The strings u, v and b are all members of L since they are all of length less then n. Since $u.\lambda$, $\lambda.b \in L$, by Remark 7.13 $ub \in L$. Similarly since ub, $\lambda.v \in L$, by Remark 7.13 $ubv \in L$, which completes the inductive step and the proof. QED.

As shown in section 5.5 inductive algorithms which use positive data to identify a language must avoid *overgeneralisation*, that is choosing a language which is a superset of the target

language. For this purpose Angluin (Angluin, 1982b) has shown the need to define a *characteristic sample* for any class of languages. A characteristic sample of a k-contextual language L is a sample S^+ of L with the property that L is the smallest k-contextual language that contains S^+. If a characteristic sample for L is found in the sample, then proposing L is not an overgeneralisation.

Remark 7.15. $A = (Q, \Sigma, \delta, \{q_0\}, F)$ is k-contextual if and only if for all strings u_1vw_1 and u_2vw_2 accepted by A, where $|v| = k$, there is a unique state q such that $\delta(q_0,u_1v) = q = \delta(q_0,u_2v)$.

As k-contextual languages have a lot in common with k-reversible languages (we show later in Theorem 7.20 that every k-contextual language is k-reversible), the following proof follows a similar proof of Angluin's (Angluin, 1982b) closely.

Theorem 7.16. For any k-contextual language L there exists a characteristic sample S^+ of L.

Proof. If $L = \varnothing$ then $S^+ = \varnothing$ is a characteristic sample of L so suppose $L \neq \varnothing$. Let $A = (Q, \Sigma, \delta, \{q_0\}, F)$ be the canonical acceptor of L. For each $q \in Q$ let L_q denote the set of k-leaders of q in A. For each pair $q \in Q$ and $x \in L_q$ let $u(q,x)$ be some string u such that $\delta(q_0,ux) = q$. The sample S^+ is defined as containing all strings u of length less than k which are in L, some string $u(q,x)xbv \in L$ for each $q \in Q$, $x \in L_q$, $b \in \Sigma$, and some string $u(q_f,x)x \in L$ for each $q_f \in F$, $x \in L_{q_f}$ No other strings are in S^+. Lemma 7.14 establishes that in the case $k = 0$, $L = \Sigma^*$ where $b \in \Sigma$ if and only if there is some $ubv \in L$. Thus S^+ is a characteristic sample of L for $k = 0$ if for every $b \in \Sigma$ there is some string of the form $ubv \in S^+$, which is so by the definition of S^+. Suppose $S \geq 1$.

Let L' be any k-contextual language containing S^+. We must show that L is contained in L'. Clearly any element of L of length less than k is in S^+ and therefore in L'. We show by induction that for every $w \in Pr(L)$ of length at least k, $T_{L'}(w) = T_{L'}(u(q,x),x)$, where x is the suffix of w of length k and $q = \delta(q_0,w)$. If w has length exactly k, then $w = x$ and $u(q,x) = \lambda$, so this condition is satisfied. Using the inductive hypothesis we suppose that for every $n \geq k$ this condition is satisfied for all strings $w \in Pr(L)$ of length at most n. Suppose w is any element of $Pr(L)$ of length $n + 1$. Let $w = w'axb$, where $|x| = k - 1$ and $a,b \in \Sigma$. By the inductive hypothesis $T_{L'}(w'ax) =$

$T_{L'}(u(q,ax)ax)$, where $q = \delta(q_0,w'ax)$. Thus $T_{L'}(w) = T_{L'}(u(q,ax)axb)$. Let $q' = \delta(q,b) = \delta(q_0,w)$. Then S^+ contains the strings $u(q,ax)axbv_1$ and $u(q',xb)xbv_2$, so L contains these strings. By Remark 7.15, this implies that $T_{L'}(u(q,ax)axb) = T_{L'}(u(q',xb)xb)$, so $T_{L'}(w) = T_{L'}(u(q',xb)xb)$, completing the induction step.

Now let w be any element of L of length at least k, and let x be the suffix of w of length k. Then $T_{L'}(w) = T_{L'}(u(q_f,x)x)$, where $q_f \in F$. The string $u(q_f,x)x$ is contained in S^+ by construction and therefore is in L'. Hence w is in L', which completes the proof that L is contained in L'. Thus L is the smallest k-contextual language containing S^+, and S^+ is a characteristic sample of L. QED.

Example 7.17. Consider the language 0^+1^+ whose canonical acceptor is shown in figure 7.3. Using the construction method of the above proof to construct a characteristic sample S^+ for this 1-contextual language, we obtain $L_A = \{0\}$, $L_B = \{1\}$ and $S^+ = \{0011\}$. Note that this is only one

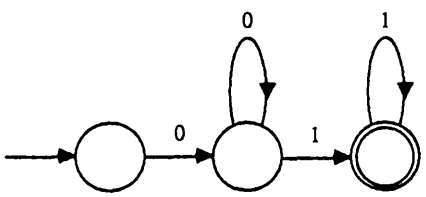


Figure 7.3 The canonical acceptor of the language 0^+1^+

possible solution for S^+. Among other characteristic samples are $S^+ = \{001, 011\}$ and $S^+ = \{0000111\}$.

Clearly a characteristic sample of a k-contextual language can consist of a single string. This is not true of any other similar language group in the literature (Angluin, 1982b; Biermann and Feldman, 1972; Levine, 1982; Miclet, 1980).

Lemma 7.18. If A is a k-contextual acceptor and A' is any subacceptor of A, then A' is a k-contextual acceptor.

Proof. Let k be a natural number, $A = (Q, \Sigma, \delta, I, F)$ be some k-contextual acceptor, and $A' = (Q', \Sigma', \delta', I', F')$ be a subacceptor of A. A' is a subacceptor of A only if $\delta'(q',b) \subseteq \delta(q',b)$ for all $q' \in Q'$ and $b \in \Sigma$. Let us assume that A' is not k-contextual. Thus by Remark 7.15 $\delta'(q_0, u_1v) \neq \delta'(q_0, u_2v)$ for some $u_1vw_1, u_1vw_1 \in L(A')$. We can show trivially by mathematical induction that since by the definition of subacceptors (see section 5.4.1) $\delta'(q',b) \subseteq \delta(q',b)$ for all $q' \in Q'$, $b \in \Sigma'$, it follows that $\delta'(q',w) \subseteq \delta(q',w)$ for all $w \in \Sigma'^*$. Remark 7.15 shows that $\delta(q_0, u_1v) = \delta(q_0, u_2v) = \{q\}$. Thus since $\delta'(q',w) \subseteq \delta(q',w)$ for all $w \in \Sigma'^*$, it follows that both $\delta'(q_0, u_1v) \subseteq \{q\}$ and $\delta'(q_0, u_2v) \subseteq \{q\}$. However $\delta'(q_0, u_1v) \neq \emptyset$ and $\delta'(q_0, u_2v) \neq \emptyset$ since $u_1vw_1, u_2vw_2 \in L(A')$. Thus $\delta'(q_0, u_1v) = \delta'(q_0, u_2v) = \{q\}$. This contradicts our assumption that A' is not k-contextual. Therefore A' is k-contextual. QED.

7.3.2. Relationship between k-reversibility and k-contextuality

The following definition of k-reversible languages is given by Angluin (Angluin, 1982b).

Definition 7.19. Let L be a regular language. Then L is k-reversible if and only if whenever u_1vw and u_2vw are in L and $|v| = k$, $T_L(u_1v) = T_L(u_2v)$.

Comparing this definition with that of k-contextuality (Definition 7.10), gives us the following theorem.

Theorem 7.20. Any k-contextual language L is k-reversible.

The proof of theorem 7.20 follows trivially from the fact that the definition for k-contextuality subsumes that of k-reversibility.

7.3.3. The KC algorithm

The following algorithm constructs a k-contextual acceptor given a positive sample and a k value.

Algorithm KC

Input: a nonempty positive sample S^+ and a k parameter.

Output: a k-contextual acceptor A.

* Initialisation
Let $A_0 = (Q_0, \Sigma_0, \delta_0, I_0, F_0)$ be $PT(S^+)$.
Let π_0 be $\{\{u\}: u \in Q_0, |u| < k\}$.
Let Q_0' be $Q_0 - \bigcup\pi_0$.

* Merging
For each state $u_1v \in Q_0'$ where $|v| = k$ do
begin
  If there exists some block B_1 such that $B_1 = B(u_2v,\pi_i)$ then
    Let B_2 be $B_1 \cup \{u_1v\}$.
  else
    Let B_2 be $\{u_1v\}$.

  Let π_{i+1} be π_i with B_2 replacing B_1.
  Increase i by 1.
end

*Termination
Let $f = i$
Output A_0/π_f.

Note that only π_f is a complete partition of Q_0.

7.3.4. The correctness of KC

The following Lemma describes the effect of *KC*.

Lemma 7.21. Let S^+ be a nonempty positive sample, k a natural number and let A_0/π_f be the acceptor output by KC on input S^+ and k. Then π_f is the finest partition of the states of A_0 such that A_0/π_f is k-contextual.

Proof. Let $A_0 = (Q_0, \Sigma_0, \delta_0, I_0, F_0)$. By inspection we note that the initialisation and merging sections of KC guarantee that every state of Q_0 will be placed into exactly one block of π_f. Thus π_f is a partition of Q_0, and A_0/π_f is a legal acceptor. Furthermore a trivial inductive argument can be employed to show that every block B of π_f contains either a single state $u \in Q_0$ for which $|u| < k$, or all states $uv \in Q_0$ for which uv has a particular suffix v of length k.

Let u_1vw_1 and u_2vw_2 be two strings in a language L, where $|v| = k$. By Definition 7.10, L is k-contextual if and only if $T_L(u_1v) = T_L(u_2v)$, i.e. u_1v and u_2v lead to the same state in $A(L)$. Since all states $uv \in Q_0$ for which uv has a particular suffix v of length k are contained within the same block of π_f it follows that u_1v and u_2v lead to the same state in A_0/π_f for any u_1vw_1, $u_2vw_2 \in S^+$. Thus A_0/π_f is k-contextual.

It remains to show that if π is any partition of Q_0 such that A_0/π is k-contextual, then π_f refines π. Let us assume the opposite, i.e. there exists some π which refines π_f where π is not equal to π_f and $L = L(A_0/\pi)$ is k-contextual. Thus at least one block of π_f is the union of more than one block of π. But as all blocks of π_f contain either singletons or contain all states $uv \in Q_0$ for which uv has a particular suffix v of length k there must exist at least two blocks of π containing states with the same k-leader. Let these two blocks B_1 and B_2 contain u_1v and u_2v respectively, where v is the common k-leader. This implies that $T_L(u_1v) \neq T_L(u_2v)$ and therefore L is not k-contextual; which contradicts the original assumption and shows that π_f refines all partitions π for which A_0/π is k-contextual. This completes the proof. QED.

The following theorem is analogous to a theorem proved by Angluin (1982b) for her k-RI algorithm.

Theorem 7.22. Let S^+ be a nonempty positive sample, and let A_f be the acceptor output by

algorithm *KC* on input S^+. Then $L(A_f)$ is the smallest k-contextual language containing S^+.

Proof. Lemma 7.21 shows that $L(A_f)$ is a k-contextual language containing S^+. Let L be any k-contextual language containing S^+, and let π be the restriction of the partition π_L to the elements of $Pr(S^+)$. If A_0 denotes the prefix tree acceptor for S^+, then Lemma 5.1 shows that A_0/π is isomorphic to a subacceptor of $A(L)$, and Corollary 5.2 shows that $L(A_0/\pi)$ is contained in L. From Lemma 7.18, $L(A_0/\pi)$ is k-contextual. Thus by lemma 7.21 π_f refines π, so $L(A_0/\pi_f)$ is contained in $L(A_0/\pi)$. Consequently, $L(A_f)$ is contained in L, and $L(A_f)$ is the smallest k-contextual language containing S^+. QED.

7.3.5. The running time of KC

Theorem 7.23. The algorithm KC may be implemented to run in time $O(n)$ where n is one more than the sum of the lengths of the input strings.

Proof. Let S^+ be the set of input strings and n be one more than the sum of the lengths of strings in S^+. The prefix tree acceptor $PT(S^+)$ can be constructed in time $O(n)$ and contains no more than n states. Both π_0 and Q_0' can be created in a single pass over all strings in S^+, and thus also take time $O(n)$. Since Q_0' contains at most n strings and each pass through the iteration can be completed in constant time given a hashing mechanism for finding the appropriate block B_1, merging also takes $O(n)$ time. The output automaton A_0/π_f can also be created in time $O(n)$. Since no operation takes more than time $O(n)$ it follows that the algorithm *KC* completes within time $O(n)$. QED.

7.3.6. Identification in the limit of k-contextual languages

In this section we show that *KC* is able to *identify in the limit* any language L (see section 5.2). We define an operator KC_∞ which given an infinite sequence of strings $w_1, w_2, w_3, \ldots$ and a parameter k produces an infinite sequence of acceptors $A_1, A_2, A_3, \ldots$ in which

$$A_n = KR(\{w_1, w_2, \ldots, w_n\},k) \text{ for all } n \geq 1.$$

An infinite sequence is called a *positive presentation* of a language L if and only if the range of the sequence is exactly L, that is, every element of the sequence is an element of L and vice versa. The following theorem shows that KC_∞ identifies k-contextual languages in the limit.

Theorem 7.24. Let L be a nonempty k-contextual language for some natural number k. Let w_1, w_2, w_3, ... be a positive presentation of L, and A_1, A_2, A_3, ... be the output of KC_∞ on this input. Then $L(A_1)$, $L(A_2)$, $L(A_3)$, ... converges to L after a finite number of steps.

Proof. By Theorem 7.16, L contains a characteristic sample. Let N be sufficiently large that w_1, w_2, ..., w_N contains a characteristic sample for L. For $n \geq N$, $L(A_n)$ is the smallest k-contextual language containing w_1, w_2, ..., w_N, by definition of KC_∞ and Theorem 7.22. Thus $L(A_n) = L$, by the definition of a characteristic sample (section 7.3.1). QED.

7.3.7. Incremental nature of KC

As stated in section 1.2 expert systems are generally built in an incremental fashion. For this reason it is desirable that any inductive tool used in the construction of expert systems produces a gradually changing output given progressive augmentation of the example set. Without this guarantee, the knowledge engineer (or expert) presenting the example material has no ability to predict the effect that any particular new example is likely to have on the system's knowledge structure. We therefore propose the following definition of incremental modification for grammatical induction algorithms.

Definition 7.25. Let A be the acceptor output by some grammatical induction algorithm I given the positive sample S^+ and let the acceptor A' be the output of I on input $S^+ \cup \{w\}$. We say that I is *incremental* if and only A is a subacceptor of A'.

Theorem 7.26. Given a fixed natural number k, the algorithm KC is incremental on input k and any positive presentation of some k-contextual language L.

Proof. Let k be a natural number, S^+ be a positive sample, w be some string, $A = PT(S^+)/\pi = (Q,$

Σ, δ, I, F) be the output of KC on input k and S^+, and $A' = PT(S^+ \cup w)/pi' = (Q', \Sigma', \delta', I', F')$ be the output of KC on input k and $S^+ \cup \{w\}$. We need to show that A is a subacceptor of A'.

By definition A is a subacceptor of A' if and only if $Q \subseteq Q'$, $I \subseteq I'$, $F \subseteq F'$ and $\delta(q, b) \subseteq \delta'(q, b)$ for all $q \in Q$ and $b \in \Sigma$. Following a similar argument to that of Lemma 7.21, we get that $\pi = \{\{u\}: uv \in S^+, |u| < k\} \cup \{B_y: xyz \in S^+, |y|=k, uv \in B_y\}$ and $\pi' = \{\{u\}: uv \in S^+ \cup \{w\}, |u| < k\} \cup \{B_y: xyz \in S^+ \cup \{w\}, |y|=k, uv \in B_y\}$. Thus for every block $B = \{u\}$ in π for which $uv \in S^+, |u| < k$ there exists one and only one corresponding block $B' = \{u\}$ in π'. Similarly, for every block B_y in π there is a corresponding block B_y' in π'. It follows from the definition of quotient that $Q \subseteq Q'$, $I \subseteq I'$, $F \subseteq F'$ and $\delta(q, b) \subseteq \delta'(q, b)$ for all $q \in Q$ and $b \in \Sigma$. Thus A is a subacceptor of A'. QED.

7.3.8. Using negative data

Negative data can be used in the same way as that described in section 7.2.6. That is, we are given a positive and negative example set (S^+,S^-), such that S^+ and S^- are disjoint finite sets of strings. We compute the k-contextual languages for $k = 0,1,2,...$ using the positive examples, S^+, until we find some k for which the inferred language does not contain any of the strings from the negative set S^-.

7.4. Use of semantic information

Gold has shown (Gold, 1967) that no algorithm can identify the entire set of regular languages from positive example sentences alone. Thus various approaches have been used which present a language identification algorithm with positive examples together with additional information. Generally, this additional information is sufficient to allow identification in the limit. Up until now the additional information has taken the following forms.

1) Negative examples. Angluin (Angluin, 1982a) shows how a combination of positive and negative examples can be used to infer any finite automata in polynomial time.

2) A limit on the total number of states. Moore (Moore, 1956) suggested this.

3) A value related to the compactness of the output automata. (Angluin, 1982b; Biermann and Feldman, 1972; Levine, 1982) have all suggested variants on this theme.

In this section we explain a new approach to example presentation, which requires neither an *ad hoc* numerical measure, nor the need for negative data. Instead we present *semantic information* in the positive examples, by way of situation action pairs. The output of the new technique is a finite state automaton (rather than a finite state acceptor), which is expressed in a similar manner to Mealy machines. It is described how variants of existing algorithms for inducing finite state acceptors can be used in this new framework.

7.4.1. Uniquely terminated Mealy machines

Let the automaton M be (Q, X, Y, δ, I, F). Q is the set of states contained in M. I is the set of initial states of M ($I \subseteq Q$). F is the set of final states of M ($F \subseteq Q$). X is the *situation* symbol set of M. Y is the *action* symbol set of M. δ is the *transition function* of M, which maps state/situation pairs of the form (q,x) to sets of action/next-state pairs of the form (y,q') where q and q' are members of Q, x is a member of the situation symbol set X, and y is a member of the action symbol set Y. We call M a *terminated* Mealy machine, in that it is similar to a form of finite state machine called a Mealy machine (see section 2.6.5). This similarity holds in all respects except that Mealy machines do not have accepting states. We call a *terminated* Mealy machine $M = (Q, X, Y, \delta, I, F)$ *deterministic* if and only if

a) I contains exactly one member, q_i and

b) if $\delta(q,x) = (y,q')$ (for some $q,q' \in Q$, $x \in X$ and $y \in Y$) there exists no other $y' \in Y$, $q'' \in Q$ such that $\delta(q,x) = (y',q'')$.

Let the *terminated* Mealy machine M be described by the *n–tuple* $M = (Q, X, Y, \delta, I, F)$. We extend the definition of *τ–termination* of finite state acceptors to *terminated* Mealy machines as follows. M is a *τ–terminated* Mealy machine (*TTM*) if and only if for any state $q \in Q$, $\delta(q,x_\tau) =$

(y_τ, q') *implies* $q' \in F$ (where $x_\tau \in X$, $y_\tau \in Y$).

We also extend the definition of *goal state acceptor* to that of a *goal state* Mealy machine (GSM) as follows. The automaton M is a *GSM* if and only if it has a single accepting state q_g and the set of states reached by a single transition from q_g, $\{q: x \in X, y \in Y, F = \{q_g\}, \delta(q_g,x) = (y,q)\}$ is empty. In other words, a GSM has a unique *goal* state which has no outgoing arcs.

We call any *terminated* Mealy machine that is both a *TTM* and a *GSM*, *uniquely terminated*. In the following sections we will discuss mainly the properties of *deterministic uniquely terminated* Mealy machines (*DUTMM*). As mentioned in 2.6.5, DUTMMs are the basis of control within modules of the Mugol language (chapter 3), and thus have special significance within this thesis.

Example 7.27. Figure 7.4 is a diagrammatic representation of the DUTMM $M = (Q, X, Y, \delta, I, F)$

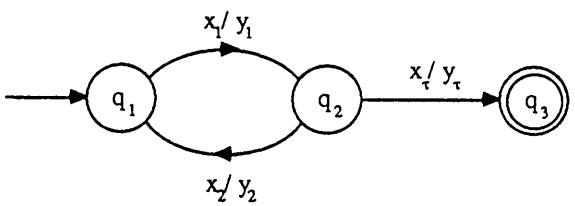


Figure 7.4 Example of a DUTMM

for which $Q = \{q_1,q_2,q_3\}$, $I = \{q_1\}$, $F = \{q_3\}$, $X = \{x_1,x_2,x_\tau\}$, $Y = \{y_1,y_2,y_\tau\}$, $\delta(q_1,x_1) = \{(y_1,q_2)\}$, $\delta(q_2,x_2) = \{(y_2,q_1)\}$, $\delta(q_2,x_\tau) = \{(y_\tau,q_3)\}$ otherwise $\delta(q,x) = \varnothing$ for all other $q \in Q$, $x \in X$, $y \in Y$.

7.4.2. Operational meaning of DUTMM's

Let $M = (Q, X, Y, \delta, \{q_i\}, \{q_g\})$ be a *DUTMM*. M can be viewed as having *semantic* properties which are akin to those of a subroutine of a programming language. M becomes *live* when called (the term *live* is used here to indicate that M is presently executing). M's executing state is initialised to the start state q_i. When executing some state q, M's present situation x is computed, and using the *transition function* the next state and next action $\delta(q,x) = (y,q')$, can be found. The next action y is executed, and on its termination, the presently executing state of M is changed to q'. If at any point M's present executing state is the goal state q_g then M returns to being *unlive* (the term *unlive* is used to indicate that M is no longer executing).

7.4.3. Situation/action sequences

Let X be the universe of situation symbols and Y be the universe of action symbols. $\Sigma_{sa} = (X \times Y)$, we call the universe of situation/action pairs. We call u a situation/action sequence if and only if $u \in \Sigma_{sa}^*$.

A *terminated* Mealy machine $M = (Q, X, Y, \delta, I, F)$ is said to *generate* the situation/action sequence $u = (x_1,y_1)(x_2,y_2) \ldots (x_n,y_n)$ if and only if there exists a sequence of not necessarily distinct states, $q_0, q_1, q_2, \ldots q_n$ such that $(y_{i+1},q_{i+1}) \in \delta(q_i,x_{i+1})$ for $0 \leq i \leq (n-1)$, $q_0 \in I$ and $q_n \in F$. Clearly the concept of "*generation* of sequences by terminated Mealy machines" is analogous to that of "*acceptance* of strings by finite state acceptors".

We call a set of situation/action sequences L_{sa} a situation/action language. The set of all situation/action sequences generated by some terminated Mealy machine M, $L_{sa}(M)$ is called the situation/action language of M. S_{sa}^+ is a positive sample of a situation/action language L_{sa} if and only if S_{sa}^+ is a subset of L_{sa}.

7.4.4. Mappings

Lemma 7.28. Given a bijection h_b which maps elements of Σ_{sa} to the universal alphabet Σ, there exists a bijection h_a which maps terminated Mealy machines to FSA's.

Proof. First we prove the existence of h_a by construction. Let $M = (Q_m, X, Y, \delta_m, I_m, F_m)$ be a terminated Mealy machine. h_a constructs an FSA $A_a = (Q_a, \Sigma, \delta_a, I_a, F_a)$. For every state $q \in Q_m$ there is exactly one state $q' \in Q_a$. For every initial state $q_i \in I_m$ there is exactly one state $q_i' \in I_a$. For every final state $q_f \in F_m$ there is exactly one state $q_f' \in F_a$. $\delta_a(q_a,b) = q_a'$ if and only if $\delta_m(q_m,x) = (y,q_m')$, the pairs of states (q_m,q_a) and (q_m',q_a') correspond in M and A_a, and $h_b((x,y)) = b$.

In order to show that h_a is a bijection, we need to prove the existence of the inverse mapping h_b^{-1}. This is also done by construction. Let $A_a = (Q_a, \Sigma, \delta_a, I_a, F_a)$ be a FSA. h_a constructs a terminated Mealy machine $M = (Q_m, X, Y, \delta_m, I_m, F_m)$. For every state $q \in Q_a$ there is exactly one state $q' \in Q_m$. For every initial state $q_i \in I_a$ there is exactly one state $q_i' \in I_m$. For every final state $q_f \in F_a$ there is exactly one state $q_f' \in F_m$. $\delta_m(q_m,x) = (y,q_m')$ if and only if $\delta_a(q_a,u) = q_a'$, the pairs of states (q_m,q_a) and (q_m',q_a') correspond in M and A_a, and $h_b^{-1}(b) = (x,y)$. QED.

Lemma 7.29. Given a bijection h_b which maps elements of Σ_{sa} to Σ, there exists a bijection h_u which maps situation/action sequences to strings.

Proof. h_u is very simply proved by construction. Let the situation action sequence u_{sa} be $(x_1,y_1)(x_2,y_2) \ldots (x_n,y_n)$. h_u constructs the string $u = b_1 b_2 \ldots b_n$ such that $h_b((x_i,y_i)) = u_i$, $1 \leq i \leq n$.

The existence of the inverse mapping, h_u^{-1} can be shown trivially and is thus omitted.

Lemma 7.30. Given the bijection h_u which maps situation/action sequences to strings, there exists a bijection h_{S^+} which maps sets of situation/action sequences into sets of strings.

Proof.

The existence of the this mapping and its inverse mapping, $h_{S^+}^{-1}$ can be shown trivially and is thus omitted.

7.4.5. The SKR algorithm

Let M be a terminated Mealy machine. We extend the usage of the term k-reversible to Mealy machines by saying that if $A = h_a(M)$ and A is k-reversible then M is also called k-reversible.

The following algorithm uses only a positive situation/action sequence sample in order to find the k-reversible Mealy machine with minimal value of k which produces the sequence.

Algorithm SKR

Input: a nonempty positive situation/action sample S_{sa}^+.

Output: the minimal-k reversible terminated Mealy machine M_f and
k's final value f.

* Initialisation
Let $k = 0$.
Let S^+ be $h_s(S_{sa}^+)$.

While $k \leq$ (the maximum length of a sequence in S_{sa}^+) + 1 do
until M_k is deterministic
begin
  Let A_k be $KR(S^+,k)$.
  Let M_k be $h_m(A_k)$.
  If M_k is not deterministic
    then increase k by 1.
end

*Termination
If M_k is not deterministic
  then
    fail.
  else
  begin
    Let $f = k$
    Output (M_f,f).
  end

Note that *SKR* can end in failure if the sample S_{sa}^+ inherently leads to a non-deterministic Mealy machine M_k for all settings of k.

7.4.6. Correctness of SKR

Theorem 7.30. Let S^+_{sa} be a positive sample of situation/action sequences. Given S^+_{sa} as input, the algorithm *SKR* will output, when it can, the pair (M_f,f) where f is the smallest value of k such that M_k is both k-reversible and deterministic. Otherwise, if no such pair exists, *SKR* will fail.

Proof. By inspection *SKR* will output the required pair (M_k,k) for the lowest value of k between 0 and the maximum length of sequence within S^+_{sa}, given that M_k is k-reversible and deterministic. According to Lemma 7.3 output of algorithm *KR* on any input S^+ and k is isomorphic to the prefix tree acceptor $PT(S^+)$ whenever k is greater than the length of the longest string within S^+. Thus if M_k is not deterministic when k is one greater than the maximum length of any string within S^+, then M_i is non-deterministic for all i greater than k, since all such M_i are isomorphic to the prefix tree acceptor of S^+. QED.

7.4.7. k-contextual sequence induction

The algorithm *SKC* which creates k-contextual Mealy machines given situation/action sequences is a trivial adaptation of *SKR* with a call to *KC* replacing that to *KR*. In fact, Biermann and Feldman's k-tail algorithm (1972) can be similarly adapted to work within a situation/action sequence environment with the accompanying advantage of eliminating the need for an arbitrary k parameter.

7.5. Conclusion

In this chapter we describe a new algorithm *KR* which is input/output equivalent to Angluin's k-RI algorithm (Lemma 7.6). However, whereas Angluin's algorithm runs in time $O(n^3)$, *KR* has been designed to run in time $O(n)$ (Theorem 7.9).

All algorithms described in chapter 5 have the common feature that no effective inductive inference is feasible with single example strings, no matter how long the given example. In section 7.3 we investigate the k-contextual language class for which effective induction *is* possible from

singleton example sets. We also show that the k-contextual languages are a restricted subset of the k-reversible languages (Theorem 7.20). In section 7.3.3 we give a simple algorithm, *KC*, which induces k-contextual languages. This is shown to run in time $O(n)$ (Theorem 7.23). Like the algorithm *KR*, *KC* has the property of being capable of identifying k-contextual languages in the limit (Theorem 7.24).

In section 7.4 we describe a method of automatically choosing the appropriate value of k for inductive construction of k-reversible and k-contextual Mealy machines. This is made possible by use of the semantic content of situation/action sequence examples. This method of inductive inference based on situation/action sequences is called *sequence induction* to distinguish it from grammatical induction. Sequence induction is demonstrated by application in chapters 6 and 8.

8

Inductive acquisition of chess strategies

Abstract. A variation of an algorithm for inducing "k-contextual" regular language grammars from sample sentences is applied to the construction of expert chess strategies. In a pilot study a small expert system for playing part of the King and two Bishops against King and Knight endgame (KBBKN) has been automatically constructed using this technique. The generated knowledge-base is directly executable in a Mugol environment (see chapter 3 and Appendices F and G).

8.1. Introduction

8.1.1. Computer chess research

In the study of expert system development, Michie (Michie, 1982a) has noted that use of chess expertise as a testbed domain is ideal in many respects. The domain is non-trivial though finitely bounded. It has a wealth of recorded expertise going back many centuries which has certainly not yet been fully exercised. Whereas chess specialists have developed a depth of understanding which is at least comparable with the expertise of more lucrative disciplines, expert-level chess players are generally more readily available for consultation.

Early work in programming computers to play chess was concentrated around efficiently implementing Shannon's chess playing strategy (Shannon, 1950). This employs extensive lookahead in order to compute approximations to the best next move. As this failed to produce results comparable with human expert play, recent research has focussed on more knowledge-rich approaches. Bratko and Michie (Bratko and Michie, 1980) described such a knowledge-based

system, AL1, based partly on earlier work by Huberman (Huberman, 1968). AL1's *advice module* generated a list of preference ordered pieces of advice. A separate *search module* used the board-state and advice list to produce a "forcing tree" which was applied as a strategy for play. As with all solutions in which knowledge must be hand-coded, the knowledge acquisition process becomes a developmental bottleneck.

Quinlan (Quinlan, 1979) suggested a method of bypassing this bottleneck by using inductive inference. Quinlan's algorithm, ID3, based on Hunt's CLS algorithm (Hunt, Marin and Stone, 1966), was used to build decision trees which classified end-game positions as won, drawn or lost. A vector of attribute values is used to describe any particular position. This vector together with a class value comprises an example classification. Although the solutions were exhaustively proved correct and ran five times faster than hand-crafted algorithms, they were also completely incomprehensible to chess experts.

In order to circumvent this understandability barrier Shapiro and Niblett (Shapiro and Niblett, 1982) introduced the notion of *structured induction*, in which a chess expert is required to hierarchically decompose the endgame classification rules; each sub-problem can then be solved inductively. While this approach avoids the problem of incomprehensibility, it unfortunately introduces a new bottleneck of problem structuring.

Paterson (Paterson, 1983) has described an attempt to automatically structure the KPK chess endgame domain from example material, using the statistical clustering algorithm CLUSTER (Michalski and Stepp, 1982). The results however have not been very promising, with the machine's suggested hierarchy not having any significance to experts. The primary reason for failure seems to lie in the fact that although the example set is a rich enough source of knowledge to be used for rule construction, additional information is necessary to indicate any higher level structure.

8.1.2. Sequence induction

In chapter 7 we described an efficient implementation of two sequence induction techniques, *k*-reversible induction and *k*-contextual induction. The *k*-contextual algorithm used for the experiments described here requires only positive examples. The necessary constraint on solutions is that the finite state acceptor produced be equivalent to the minimum sized *k*-contextual language containing the positive examples (see chapter 7). As described in section 7.4, when dealing with sequences of ID3-like examples, we can use the semantic content provided by the situational vector as an additional constraint mechanism, and thus circumvent the need for supplying the algorithm with the arbitrary measure required by all similar algorithms in the literature (Angluin, 1982b; Biermann and Feldman, 1972; Levine, 1982; Miclet, 1980). For this we employ the *SKC* algorithm described in section 7.4.7.

Situations in which sequence induction can be employed are many and varied (see chapter 6). If we understand well what the properties of the algorithm being used are, often we can take advantage of various presentation and solution constraints for different scenarios. Elsewhere (see chapter 7) several such properties are described and proved. For our purposes, the most important property of the *k*-contextual algorithm is that successive solutions are *incremental* (see section 7.3.7). Accordingly, as more examples were added the automaton output by the algorithm developed in a controlled and predictable fashion.

8.2. The problem - KBBKN

Programming strategies for chess endgames is a notoriously difficult task. Zuidema (Zuidema, 1974) commenting on two Algol 60 programs written for the King and Rook against King (KRK) endgame illustrates the difficulties by noting that "A small improvement entails a great deal of expense in programming effort and program length. The new rules will have their exceptions too."

In a project being carried out at the Turing Institute, the extremely complex chess endgame KBBKN is being studied with the aid of the world-class chess endgame specialist John Roycroft. Even this chess authority admits to being out of his depth. In the only definitive study of KBBKN,

written in 1851, Horwitz and Kling (Horwitz and Kling, 1851) claimed that with White-to-move (WTM), the game is drawn in all but trivial cases. For over a century this claim remained uncontested, until in 1983 Thompson revealed by exhaustive computation that almost all positions are forced wins for White, with a maximum length win of 66 moves being obtainable from 32 different positions (Roycroft, 1983; Thompson 1985). This surprising result adds to the pressure on the international chess community to revise the 50-move rule. According to this rule, if 50 full moves are made without a capture, castling or pawn move then a draw can be declared. However, clearly it may take up to 66 moves to force a win for a particular side. Thompson's computations have brought to light the existence of even longer minimax optimal paths in some other end-games.

The Turing Institute study involves two phases. In the first, Roycroft has studied the domain intensely with the aim of developing a sufficient set of primitive attributes. It is in this first phase that the author has carried out the evaluation of sequence induction as a knowledge acquisition tool. In the second phase it is intended that Roycroft's descriptions be matched against Thompson's exhaustive database for KBBKN.

Roycroft's first task was to select a sub-strategy within the KBBKN domain of an appropriate size and complexity for the application of sequence induction. The choice fell on the first section of one of the exceptional 66-move forced wins for White.

8.2.1. Initial position

Play commences from the position shown in figure 8.1.

Taking symmetry and slightly altered starting positions into account, this position is equivalent, in terms of the number of moves to a forced win, to several other similar positions. As this equivalence can be taken into account by the careful choice of terms when devising the expert system, we will ignore this extra dimension to the problem.

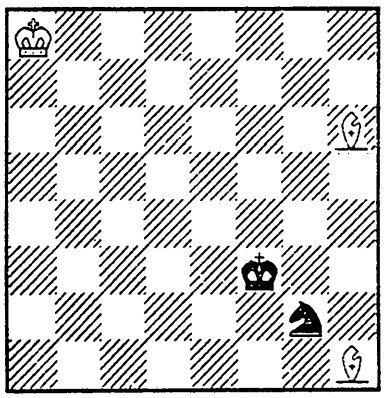


Figure 8.1 The initial position, WTM

8.2.2. Goal position

The aim of White in this sub-strategy is to liberate the dark-squares-White-bishop (wB(dark)) from the corner in no more than 12 moves. In order to achieve this it is necessary that

A) light-squares-White-bishop (wB(light)) prevents Black's king (bK) from attacking and capturing White's-bishop-on-square-h1 (wBh1). This is illustrated in figures 8.2, 8.3 and 8.4.

B) White's king (wK) moves to support the attack of wBh1 on Black's-knight-on-square-g2 (bNg2) (see figure 8.5).

Play achieving A) is trivially described and encoded. However, attaining B) is complicated considerably by White's choice of delaying tactics, employed to impede wK approaching h3. It was for this second goal that we used sequence induction to capture Roycroft's description.

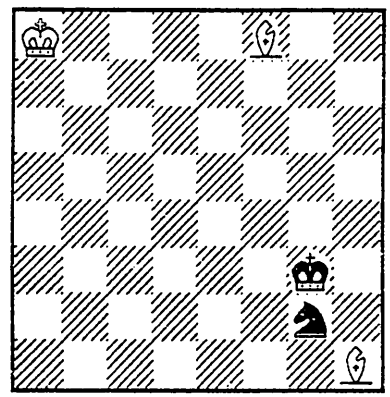


Figure 8.2 wB(light) prepares to prevent wK from moving to h2, WTM

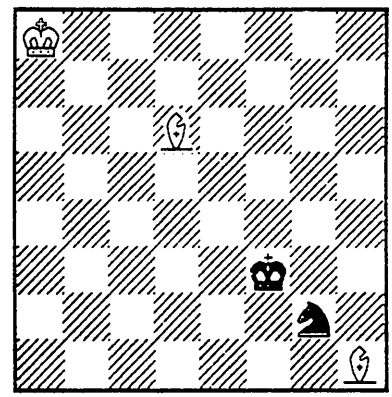


Figure 8.3 bK retreats after being checked by wB(light), WTM

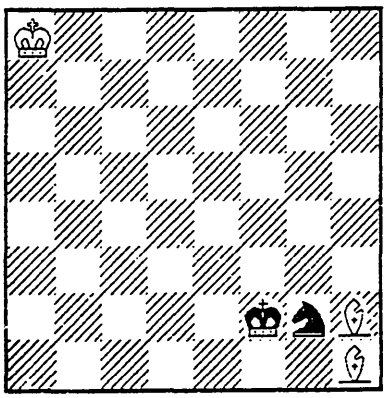


Figure 8.4 wB(light) takes up fortified position, WTM

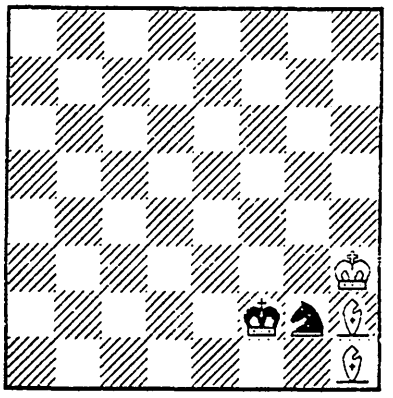


Figure 8.5 The goal of liberating wB(dark), bN forced to retreat, BTM

8.2.3. Attributes and actions

Roycroft was asked to give an exposition of play which included a set of sequences of moves together with a running commentary displaying points of interest. From this the author extracted four positional attributes (based on Roycroft's use of adjectival phrases), four action schemas taken by White (corresponding to verb phrases) and six sequences of play. The attributes were as follows

B1) Is White free to take bN? {y/n}

B2) Is wK on the same diagonal as the release position (h3)? {y/n}

B3) Can wBh1(dark) move? {y/n}

B4) Is the direct diagonal position closest to the release position covered? {y/n}

The actions were

Ba) wK approaches release position (h3) by moving along rank or file.

Bb) wK moves to non-check position closest to release position on direct diagonal.

Bc) wB(light) moves out of corner along its diagonal.

Bd) White takes bN.

Note that each action at this level represents a single move. However, the entire automaton to be derived represents a unit action involving several moves. Thus we might, if necessary, have a hierarchy of such actions and attributes, similar to that described by Shapiro and Niblett (Shapiro and Niblett, 1982) for classification (see conclusion).

8.2.4. The solution

The sequences used are reproduced in Appendix C. These sequences were added by stepwise-refinement, the result being tested after the addition of each sequence. Very early in this process, the *k*-value for the solution rose from 0 to 1, at which level it remained during the rest of development. Also, the number of states in the solution grew rapidly at first to reach a steady value of 5, at which it too stayed fixed. Altogether this process displayed a good incremental nature.

The first six sequences represent White's response to various well executed tactics played by Black. These were derived directly from Roycroft's description. Having by this stage generated a playing strategy that dealt adequately with more than Roycroft's described positions (the *k*-contextual algorithm successfully generalised solutions to a larger number of positions than those originally described) the automaton was presented and explained to Roycroft. Roycroft noted that the set of positions at which the White king can be delayed by Black was the most complex to describe. *Significantly, the state which described just these positions contained the most ID3-examples. The structure automatically imposed on the solution had a clear significance to the expert.*

As yet, with only six sequences, the solution was not able to cope with non-optimal play by Black. An additional seven sequences were added to deal with such play. The resulting *k*-contextual

automaton is given in Appendix D in a form which can be directly translated into a Mugmaker induction file. Appendix E demonstrates the transformation carried out by ID3-like induction to produce a runnable Mugol expert system. Note that all decision trees in the solution have the form of HSL (Michie, 1984) decision trees. Appendix F gives the Mugmaker file corresponding to the automaton solution of Appendix D. Using this Mugmaker file, the Mugol code of Appendix G was inductively generated.

8.3. Conclusion

We have demonstrated the feasibility of using sequence induction to construct expert-level chess strategies for endgame play. A great deal of further work is necessary to expand the work described here to completely cover the highly complex domain of KBBKN. However, the methodology used was found by the expert to be natural in terms of the example presentation requirements, as chess players are quite at home with describing play in terms of example move sequences. Furthermore, the bottleneck of structuring was eased, though not completely removed by the use of sequence induction. Whereas other attempts at automatic structuring have led to solutions which are not acceptable to experts, results produced by sequence induction were found to be intuitively correct by the endgame specialist John Roycroft.

In section 8.3.3 we noted that as the induced strategy represents a broadly defined action, it might be found necessary to form a hierarchy of successively more detailed action descriptions in order to create an extensive strategy. Therefore, it might be argued that our automatic structuring aid has gained us no ground, as it may still be necessary to do further manual structuring. We do not claim to have a complete answer to the structuring problem. However, Shapiro (Shapiro, 1983) when constructing his structured solution of KPa7KR found that the use of more than 7 examples within any particular context lead to unreadable machine induced solutions. We have used 13 example sequences each containing an average of 4 ID3-like sequences to produce a semi-structured solution in which each state's rule is derived from an average of only 3 examples. Thus despite the fact that the quantity of example material used to structure this level of problem is an

order of magnitude larger than that used by Shapiro, the generated solution contains a small number of easily understandable decision trees.

The *k*-contextual induction algorithm used has good incremental behaviour (see section 7.3.7). This algorithm has also been proved to *identify the correct solution in the limit.*

The use of two levels of induction, sequence induction and static induction, gives rise to very powerful generalisation, with solutions being output directly as runnable expert systems.

On the negative side, we have not developed a form of explanation which deals satisfactorily with sequence execution. It is hoped that by continued research, chess experts may be able to lead us to the most natural form of explanation required by chess players to describe sequences of play. Also, the *k*-contextual algorithm used for this research is written in Prolog. A more efficient implementation, with a better interface to the Mugol environment is needed.

9

Discussion

Abstract. Explanation of computer-based reasoning and the "bottleneck" (Feigenbaum, 1979) of knowledge acquisition are major issues in expert systems research. We have contributed to these areas in two ways. Firstly, we have implemented an expert system shell, the Mugol environment, which facilitates knowledge acquisition by inductive inference and provides automatic explanation of run-time reasoning on demand. RuleMaster, a commercial version of this environment, has been used in industry for the construction and testing of two large classification systems. Secondly, we have investigated a new technique called *sequence induction* which can be used in the construction of control systems. Sequence induction is based on theoretical work in grammatical learning. We have improved existing grammatical learning algorithms as well as suggesting and theoretically characterising new ones. These algorithms have been successfully applied to the acquisition of knowledge for a diverse set of control systems, including inductive construction of robot plans and chess end-game strategies. However, to date sequence induction has not been incorporated into the Mugol environment. We regard the automatic structuring of problem domains as the most important topic for further inductive inference research. Lastly we describe the author's present research project, Duce. Duce is a system for automatically structuring propositional calculus rules.

9.1. Summary

In chapter 1 we introduce the topic of expert system research, following Michie's definition. Expert system development involves continuous debugging of knowledge structures. We argue that the two most important tools in this debugging process are a) an *explanation facility* and b) an *inductive knowledge acquisition mechanism.* The major topic of interest within this thesis is that of inductive inference.

We describe two different forms of induction. *Static induction* algorithms take examples which represent descriptions of world situations to which labels are attached. These labels indicate a classification or an action to be taken. On the other hand, *sequence induction* relies on the presentation of example sequences to an inductive algorithm. Each element of the sequence is a situation/action pair similar in form to the static descriptions.

In chapter 2 we discuss the nature of inductive algorithms. Inductive algorithms use various types of example material to generate hypotheses in various rule formats. In their nature, inductive algorithms make conjectures concerning unknown facts. These conjectures must be shown to be sound according to some demonstrable criteria.

In sections 2.5 and 2.6 we use a parity problem to illustrate properties of various rule representations. In figure 2.8 we give a table of complexity results for the three chosen representations. This table shows that, for this problem at least, it is preferable to use a finite state machine representation rather than a decision-tree based one. We go on to show that finite state machine representations have more expressive power than those of propositional calculus and decision trees. However, there exist formalisms, such as Turing machines, which have even more expressive power than finite state machines. One might ask whether formal power is the ultimate criterion for deciding between representations. We argue that for expert system applications *expert comprehensibility* seems to be more pertinent to the choice of an appropriate representation than *formal power*.

In chapter 3 we describe the Mugol environment. This is an expert system building package intended to solve many of the problems involved in the construction of large knowledge based programs. This comprises an *induction engine* (Mugmaker), a *rule language* (Mugol) and an explanation facility. Mugmaker contains a variant of the ID3 static induction algorithm. Although we have investigated and tested sequence induction algorithms (chapters 5,6,7,8) the Mugol environment does not as yet contain such facilities.

Although in some respects Mugol has the characteristics of high-level languages such as Pascal or Ada, its explanation facility and rule format uniquely distinguish it for use in an expert system environment.

In comparison to other expert system approaches we believe that although our knowledge representation, in the form of decision trees, is no better than production rules, the fact that knowledge can be presented in the form of examples as well as rules means that the process of knowledge acquisition is greatly eased. It has been noted often during the construction of Mugol-based applications that whereas designers using dialogue acquisition methodologies report that construction of a prototype expert system takes two to three man years of effort (Shortliffe and Buchanan, 1975; Duda, Gashnig and Hart, 1979), similar sized Mugol applications (see Appendices A and B) have been consistently prototyped in around six person months.

Typical expert system applications contain aspects of both classification and control tasks. The Mugol environment provides a consistent knowledge representation for these disparate problem elements. Furthermore, an interface to external sources and sinks of information is provided. Such an interface, although not always recognised as a necessity for expert system development, has been found to be absolutely essential in all large Mugol applications (Appendices A, B and H).

In chapter 4 we describe a small robot planning system, ARCH, which was built by the author. The solution we describe works only for a small number of blocks, and only in simulation. However, Shepherd (Appendix H) has extended this solution within the Mugol environment, to build large recursively defined structures (around 30 blocks). Moreover, Shepherd's solution has been tested within a real-time robot environment using Puma robots and camera based sensory feedback.

As mentioned earlier in this section, the Mugol environment in its present form demands that the control structure of Mugol finite state machines be hand-coded. In chapters 5 and 7 we investigate techniques for automatically constructing finite state structures from traces of their intended execution (i.e. sequences of calls to predefined tests and actions). The techniques are

based on "grammatical induction", i.e. discovery of grammar from example sentences. In chapter 5 we present a survey of algorithms which infer a regular language from a given subset of that language. It is evident that an increasing number of heuristic approaches exist for inferring regular languages. We present a powerful new algorithm of low-order polynomial time complexity which generalises a number of those already in the literature, and provides a systematic framework for testing and comparing existing approaches without the burden of re-implementing many different algorithms.

In chapter 6 we describe six small but varied applications of the KR and SKR induction algorithms (see sections 7.2.2 and 7.4.5). The applications include automatic VLSI circuit synthesis, user modelling in a mathematical educational environment and generalisation of robot plans.

It is interesting to note that whereas each problem was inherently fairly difficult, the problem was automatically broken into a number of smaller problems, each of which would require very little decision making during execution of the automaton.

The method of constructing robot plans from example sequences has also been investigated by Dufay and Latombe (1984). However, they used a simpler inductive algorithm, essentially the same as that described by Miclet (5.4.5.3). Angluin (1982b) has shown Miclet's algorithm is merely a special case of k-reversible induction. We therefore believe our method to have a wider scope than that of Dufay and Latombe.

In chapter 7 we describe a new algorithm *KR* which is input/output equivalent to Angluin's k-RI algorithm (Lemma 7.6). However, whereas Angluin's algorithm runs in time $O(n^3)$, *KR* has been designed to run in time $O(n)$ (Theorem 7.9).

All algorithms described in chapter 5 have the common feature that no effective inductive inference is feasible with single example strings, no matter how long the given example. In section 7.3 we investigate the k-contextual language class for which effective induction *is* possible from singleton example sets. We also show that the k-contextual languages are a restricted subset of the

k-reversible languages (Theorem 7.20). In section 7.3.3 we give a simple algorithm, *KC*, which induces k-contextual languages. This is shown to run in time $O(n)$ (Theorem 7.23). Like the algorithm *KR*, *KC* has the property of being capable of identifying k-contextual languages in the limit (Theorem 7.24).

In section 7.4 we describe a method of automatically choosing the appropriate value of k for inductive construction of k-reversible and k-contextual Mealy machines. This is made possible by use of the semantic content of situation/action sequence examples. This method of inductive inference based on situation/action sequences is called *sequence induction* to distinguish it from grammatical induction. Sequence induction is demonstrated by application in chapters 6 and 8.

In chapter 8 the algorithm *SKC* (section 7.4.7) is applied to the construction of expert chess strategies. In a pilot study a small expert system for playing part of the King and two Bishops against King and Knight endgame (KBBKN) was automatically constructed using this technique.

A great deal of further work is necessary to expand the work described here to completely cover the highly complex domain of KBBKN. However, the methodology used was found by the expert to be natural in terms of the example presentation requirements, as chess players are quite at home with describing play in terms of example move sequences. Furthermore, the bottleneck of structuring was eased, though not completely removed by the use of sequence induction. Whereas other attempts at automatic structuring have led to solutions which are not acceptable to experts, results produced by sequence induction were found to be intuitively correct by the endgame specialist John Roycroft.

In accordance with theoretical results (see section 7.3.7), the k-contextual induction algorithm used was found to have good incremental behaviour. By using two levels of induction (sequence induction and static induction) very powerful generalisations are produced. The induced solution of chapter 8 has been executed in a Mugol environment (see Appendices F and G).

9.2. Directions for further work

Commercially available packages (McLaren, 1984; A-Razzak, Hassan and Pettipher, 1984; Michie, Muggleton, Riese and Zubrick, 1984) have already, during their short existence, proved the power of inductive inference in the construction of expert systems. These packages allow development-time savings in building large expert systems of at least an order of magnitude over the traditional "deductivist" rule extraction technique. However, it cannot be said that the "inductivists" have yet completely met their targets. Although present day inductive inference techniques go a long way towards easing what Feigenbaum (1979) called the *bottleneck* of knowledge acquisition, one might say that this bottleneck has merely shifted. The new bottleneck involves the hierarchical structuring of problem domains. This problem, related to what Michalski (1986) has termed *constructive induction* has had very little attention so far. Clearly however, investigation in this area is crucial to the further development of practical knowledge acquisition tools.

Along these lines, the author has recently been investigating a technique for interactive knowledge structuring and generalisation (Muggleton, 1986). The Prolog program which presently embodies this technique is called Duce. Duce uses a new transformational programming approach to automatically structure and generalise examples/rules described as horn clauses in propositional calculus. Six simple operators are used to progressively compress the rules by forming new concepts and generalisations. Duce is interactive, in that it both requests names for the various new concepts produced, as well as checking the validity of concepts against the user (or *oracle*).

Duce uses "illustrative examples" to describe new concepts to a human oracle in a graphical form. Thus in the chess world, the chess expert will decide on the comprehensibility of a new concept based on a number of chess board positions which exemplify the new concept. These positions are displayed graphically, allowing the expert to make decisions in a more natural setting than that of having to view a complex description based on low-level attribute descriptors.

Duce has been tested on two small tasks and one large task. The smaller tasks were those of finding a structure for the "even-parity" problem (see chapter 2) and developing a taxonomic structure for a small number of animal descriptions. Both of these tests were passed satisfactorily; Duce discovered a divide and conquer algorithm similar to that discussed in chapter 2 for the first problem and found the concepts of *bird* and *primate* in the second case.

The third, and most taxing test of Duce's capabilities depends on its ability to rediscover, and possibly improve on, a structure for deciding the predicate *won for white* for any chess position within the chess end-game domain of KPa7KR. A structure for this problem was originally created manually by Shapiro and Kopec, and described in Shapiro's PhD thesis (1983). The domain contains around 200,000 positions. This was reduced to around 3,000 examples by describing the positions in terms of 36 low-level descriptors suggested by the chess end-game specialist Danny Kopec. These examples form a complete enumeration and were automatically generated by use of a mini-max backup algorithm. Although experimentation within this domain is still in progress, the new high-level concepts of *delayed queening* and *mate threat* proposed by Duce have been accepted and named by another chess end-game specialist, Ivan Bratko.

As indicated in chapter 8, the problem of hierarchical knowledge structuring applies just as strongly to *sequence induction* as it does to *static induction.* to look at methods of inducing context free grammars from example sentences. This would require the construction of intermediary terms. According to Gold (1967) it is possible to identify context free grammars in the limit using a positive and negative example source. However, it may well make sense to also look for an oracle based transformational solution to this problem.

References

Angluin D. (1978), On the complexity of minimum inference of regular sets. *Information and Control* 39:337-350.

Angluin D. (1982a), A note on the number of queries needed to identify regular languages. *Information and Control* 51(1):76-87.

Angluin D. (1982b), Inference of reversible languages. *Journal of the ACM* 29:741-765.

Angluin D. and Smith C.H. (1982), *A survey of inductive inference: theory and methods.* New Haven, CT: Yale University.

Arbab B. and Michie D. (1985), Generating rules from examples. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence.* Los Altos, CA: Kaufmann, pp. 631-633.

A-Razzak M., Hassan T. and Pettipher R. (1984), EXTRAN-7: A Fortran-based software package for building expert systems. *Research and Development in Expert Systems.* ed. M.A. Bramer, pp 23-30. Cambridge: Cambridge University Press.

Biermann A.W. and Feldman J.A. (1972), On the synthesis of finite-state machines from samples of their behaviour. *IEEE Transactions on Computers* C-21:592-597.

Bobrow D.G. and Stefik M. (1983), *The LOOPS manual.* Palo Alto, CA: Xerox, 1983.

Bratko I. (1983), *Generating human-understandable decision rules.* E. Kardelj University Ljubljana (working paper).

Bratko I. and Michie D. (1980), A representation of pattern-knowledge in chess endgames. *Advances in Computer Chess 2*, Edinburgh: Edinburgh University Press, pp. 31-56.

Cohen P.R. and Grinberg M.R. (1983), A framework for heuristic reasoning about uncertainty. *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, Los Altos, CA: Kaufmann, pp. 355-357.

Dechter R., and Michie D. (1984), *Induction of plans.* Glasgow: The Turing Institute (TIRM-84-006).

Duda R.O., Gashnig J. and Hart P.E. (1979), Model design in the PROSPECTOR consultant program for mineral exploration. *Expert Systems in the Microelectronic Age*, ed. D. Michie. Edinburgh: Edinburgh University Press, pp. 153-167.

Duda R.O., Hart P.E., and Nilsson N. (1976), *Subjective Bayesian methods for rule-based inference systems.* Menlo Park, CA: Stanford Research Institute, Artificial Intelligence Center (Technical Note 124).

Dufay B. and Latombe J.C. (1984), An approach to automatic robot programming based on

inductive learning. *International Journal of Robotics Research*, 3(4): 3-20.

Fagan L.M., Kunz J.C., Feigenbaum E.A., and Osborne J.J. (1979), Representation of dynamic clinical knowledge; measurement interpretation in the intensive care unit. *Proceedings of the Sixth International Joint Conference on Artificial Intelligence, Tokyo*, Los Altos, CA: Morgan Kaufmann Publishers, Inc., pp. 260-262.

Feigenbaum E.A. (1979), Themes and case studies of knowledge engineering. *Expert Systems in the Micro-electronic Age*, Ed. D. Michie. Edinburgh: Edinburgh University Press, pp. 3-25.

Fu K.S. and Booth T.L. (1975), Grammatical inference: introduction and survey. *IEEE Transactions on Systems, Man, Cybernetics*. SMC-5:95-111,409-423.

Gold E.M. (1967), Language identification in the limit. *Information and Control* 10:447-474.

Gold E.M. (1978), Complexity of automaton identification from given data. *Information and Control* 37:302-320.

Hopcroft J.E. and Ullman J.D. (1979), *Introduction to Automata and Formal Languages*. Reading, MA: Addison-Wesley.

Horwitz and Kling (1851), *Chess Studies*. London: Skeet.

Huberman B.J. (1968), *A program to play chess end-games*. Stanford, CA: Stanford University, Computer Science Department (Technical report no. CS 106).

Hunt E.B., Marin J. and Stone P. (1966), *Experiments in induction*. New York: Academic Press.

Intellicorp (1984), *The Knowledge Engineering Environment*. Menlo Park, CA: Intellicorp.

Levine B. (1982), The use of tree derivatives and a sample support parameter for inferring tree systems. *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-4:25-34.

McLaren R. (1984), *Expert Ease User Manual*. Glasgow: Intelligent Terminals Ltd.

Michalski R.S. and Chilausky R.L. (1980), Learning by being told and learning from examples: an experimental comparison of the two methods of knowledge acquisition in the context of developing an expert system for soybean disease diagnosis. *International Journal of Policy Analysis and Information Systems*. 4(2): 125-161.

Michalski R.S. and Stepp R. (1982), Revealing conceptual structure in data by inductive inference. *Machine Intelligence 10*. Ed. J.E. Hayes, Donald Michie and Y-H Pao. Chichester: Horwood. pp173-196.

Michalski R.S. (1983), A theory and methodology of inductive learning. *Machine Learning: An Artificial Intelligence Approach*. Palo Alto, CA: Tioga. pp. 83-134.

Michalski R.S. (1986), Understanding the nature of learning: issues and research

directions. *Machine Learning: An Artificial Intelligence Approach. Vol. 2* Eds. Michalski R.S., Carbonell J.G. and Mitchell R.M. Los Altos, CA: Kaufmann, pp. 3-25.

Michie D., Muggleton S., Riese C. and Zubrick S. (1984), RuleMaster: a second-generation knowledge-engineering facility. *Proceedings of the First Conference on Artificial Intelligence Applications*, Denver. IEEE Computer Soc., pp. 591-597.

Michie D. (1982), Measuring the knowledge content of expert programs. *The Bulletin of the Institute of Mathematics and its Application*, 18 (Nov/Dec), pp. 216-220.

Michie D. (1982a), Computer chess and the humanisation of technology. In Nature, Vol. 299, September 1982, pp. 391-394.

Michie D. (1984), Quality control of induced rule-based programs. *The Fifth Generation*. London: CGS Institute.

Michie D. (1985), Expert systems and robotics. *Handbook of Industrial Robotics*. Ed. S. Nof. New York: Wiley, pp 419-436.

Miclet L. (1980), Regular inference with a tail clustering method. *IEEE Transactions on Systems, Man, Cybernetics* SMC-10:737-743.

Minsky M. (1975), A framework for representing knowledge. *The Psychology of Computer Vision*. Ed. P.Winston. NewYork: Mcgraw-Hill, 211-277.

Moore E.F. (1956), Gedanken-experiments on sequential machines. *Automata Studies* Eds. C.E. Shannon and J. McCarthy. Princeton, NJ: Princeton University Press, pp. 129-153.

Moses J. (1975), A MACSYMA primer. Mathlab Memo No. 2, Computer Science Laboratory, Massachusetts Institute of Technology.

Mozetic I., Bratko I. and Lavrac N. (1984), *The derivation of medical knowledge from a qualitative model of the heart*. Ljubljana: Josef Stefan Institute.

Muggleton S.H. (1983), A process description language. *M.I.News* No.3. Glasgow: The Turing Institute.

Muggleton S.H. (1984a), *An inductively acquired strategy for building an arch in a blocks world*, (unpublished working paper)

Muggleton S.H. (1984b), Induction of regular languages from positive examples. *M.I.News 5*. Glasgow: The Turing Institute.

Muggleton S.H. (1985a), *Some experiments with grammatical induction*. M.I.News 7. Glasgow: The Turing Institute.

Muggleton S.H. (1986), An efficient method of inductively acquiring chess strategies. To appear in Machine Intelligence 11, Oxford: Oxford University Press.

Muggleton S.H. (1986), *Duce - a program for automatically structuring propositional calculus*

rules, (unpublished working paper).

Niblett T. (1985), YAPES: Yet Another Prolog Expert System. *CC-AI: the journal for integrated study of artificial intelligence, cognitive science and applied epistemology*. Intelligent systems 2(2), 3-30.

NTIS (1969), Use of the Skew T, Log P Diagram in Analysis and Forecasting. Air Weather Service Manual AWSM 105-124, NTIS AD695603. Springfield, VA: National Technical Information Service.

Pao T.W. and Carr J.W. III (1978), A solution of the syntactical induction-inference problem for regular languages. *Computer Languages* 3:53-64.

Paterson A. (1983), *An attempt to use CLUSTER to synthesise humanly intelligible subproblems for the KPK chess endgame*. Urbana, IL: Univ. Illinois, (UIUCDCS-R-83-1156)

Paterson A. (1984), *Computer Induction in a Tutorial Context*. Edinburgh: University of Edinburgh, (M.Phil. Thesis).

Paterson A. and Niblett T. (1982), *ACLS Manual*. Glasgow: Intelligent Terminals Ltd.

Popper K. R. (1972), *Conjectures and Refutations: the Growth of Scientific Knowledge*. London: Routledge.

Quinlan J.R. (1979), Discovering rules from large collections of examples: a case study. *Expert Systems in the Micro-electronic Age*. Ed. D. Michie. Edinburgh: Edinburgh University Press.

Quinlan J.R. (1982a), *INFERNO: a cautious approach to uncertain inference*. Santa Monica, CA: RAND Corporation (RAND Note N-1898-RC).

Quinlan J.R. (1982b), Semi-autonomous acquisition of pattern-based knowledge. *Introductory readings in expert systems*. Ed. D. Michie. New York: Gordon & Breach. pp. 192-207.

Radian Corporation (1984), *RuleMaster System User Manual*. Austin, TX: Radian Corporation (Technical Report DCN 84-141-603-01).

Riese C. (1984), Transformer fault detection and diagnosis using RuleMaster by Radian. Austin, Tx: Radian, 1984.

Roycroft J. (1983), A prophesy fulfilled. *EG magazine* (November, 1983), Ed. J. Roycroft.

Shafer G. (1976), *A Mathematical Theory of Evidence*. Princeton, NJ: Princeton University Press.

Shannon C.E. (1950), Programming a computer for playing chess. *Phil. Mag.* 41 pp. 256-275.

Shapiro A.D. (1983), *The Role of Structured Induction in Expert Systems*. Edinburgh: Edinburgh University (Ph.D. thesis).

Shapiro A.D. and Michie D. (1986), A self-commenting facility for inductively synthesised

endgame expertise. *Advances in Computer Chess 4*. Ed. D.F. Beal. Oxford: Pergamon Press, pp. 147-165.

Shapiro A.D. and Niblett T.B. (1982), Automatic induction of classification rules for a chess endgame. *Advances in Computer Chess 3*. Oxford: Pergamon Press, pp. 73-92.

Shortliffe E.H. and Buchanan B.G. (1975), A model of inexact reasoning in medicine. *Mathematical Biosciences* 23:351-379.

Thompson K. (1986), Private letter to J. Roycroft. *EG magazine* (January 1986), Ed. J. Roycroft.

Van Melle W.J. (1980), *System Aids in Constructing Programs*. Ann Arbor, M.I.: UMI, (Computer Science. Artificial Intelligence 11).

Yung-Choa Pan J. (1984), Qualitative reasoning with deep-level mechanism models for diagnosis of mechanism failure, *Proceedings of the First Conference on Artificial Intelligence Applications*, Denver. IEEE Computer Soc., pp. 295-301.

Zadeh L.A. (1979), A theory of approximate reasoning. *Machine Intelligence 9*. Ed. Hayes, Michie, Mikulich. Chichester: Horwood, pp. 149-194.

Zubrick S. (1984), *Willard: a severe thunderstorm forecasting system using RuleMaster by Radian*. Austin, Tx: Radian.

Zubrick S. (1986), Validation of a weather forecasting expert system. To appear in *Machine Intelligence 11*. Oxford: Oxford University Press.

Zuidema C. (1974), Chess, how to program the exceptions? *Afdeling informatica* IW21/74. Amsterdam: Mathematisch Centrum.

Appendix A

WILLARD

Abstract. The Mugol environment has been tested in the construction of two large expert classification systems, WILLARD and EARL. In this section we describe Steve Zubrick's implementation of WILLARD and discuss the validation methods that have been used to assess WILLARD's performance. The thesis author aided in suggesting the structuring methodology used in WILLARD.

Introduction

WILLARD (Zubrick, 1984) is an expert system for predicting the likelihood of severe thunderstorms occurring in the central USA. The system was written by Steve Zubrick, a meteorologist at Radian Corporation. Extensive testing of the system (Zubrick, 1986) has shown that it is capable of producing predictions which can usefully complement those of the US National Weather Service. The author gave help and advice in the structuring and example acquisition of WILLARD.

On average, over 1000 severe thunderstorms are reported each year in the central United States, causing the loss of many lives and billions of dollars of property damage. The National Weather Service defines severe thunderstorms as the occurrence of one or more of the following conditions:

wind gusts greater than 50 knots,

tornados, and/or

hailstones greater than 3/4 inch in diameter.

Severe thunderstorm forecasting for the entire U.S. is currently done by highly skilled meteorologists at the National Severe Storms Forecast Center (NSSFC).

This time-consuming task entails continuous analysis of vast amounts of raw data and numeric modelling results, much of which turns out to be irrelevant. An expert system might automatically screen the data, providing the meteorologists with suggested forecasts together with their justifications.

A large number of specific case studies of occurrences of severe thunderstorms have been documented and analysed in the meteorological literature. However, no coherent system of rules covering all possible cases has yet been synthesised. For this reason, an inductive rule generator would appear to be a powerful tool for generalising this accumulated knowledge.

The system

For the purposes of rapid development, an initial set of examples provided by the expert were used to build the prototype expert system. Additional cases of real weather data have subsequently been applied in the ongoing refinement of WILLARD. An illustration of the use of inductive inference in the development of WILLARD is given in figure 3.1 (section 3.3).

The WILLARD expert system is composed of a hierarchy of thirty modules, each of which contains a single decision rule (see figure A.1). This hierarchy is on average four levels deep. All modules' rules were developed using inductive generalisation. A total of around 140 examples were used in building WILLARD. WILLARD has a domain size of approximately nine million measurably different situations.

For the top level module, the inductive algorithm was able to order the critical meteorological factors in a manner consistent with the way forecasters perform their analysis. For example, if the key factors are unfavourable, then a decision can be made rapidly, otherwise, more parameters are investigated until a decision can be reached.

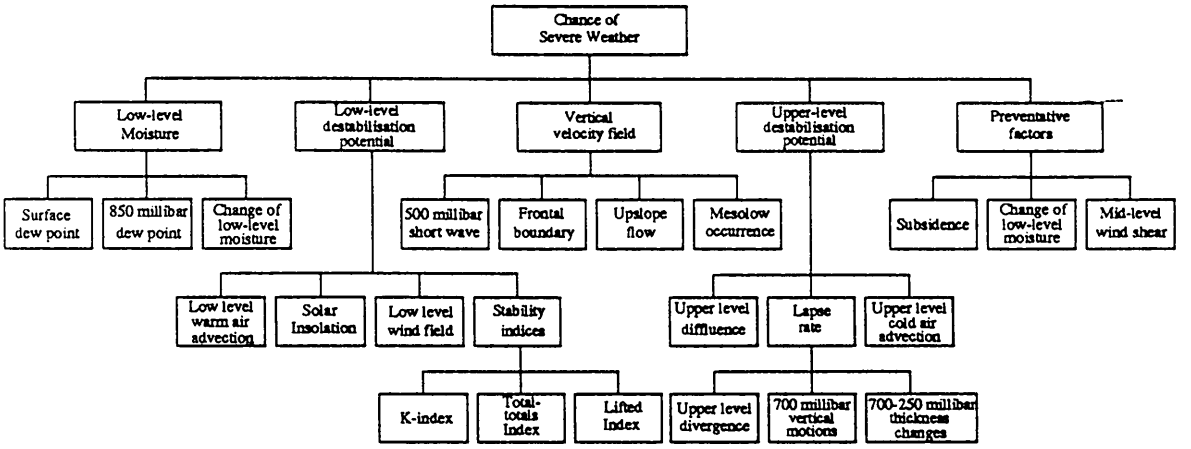


Figure A.1 WILLARD structure

Although WILLARD is essentially an expert classification system, the Mugol environment facilitates the use of control loops required for top level control and the monitoring of incoming data.

WILLARD can operate in interactive or automatic forecast mode. In the manual mode, the system asks questions of the meteorologist about pertinent weather conditions for the forecast area and produces a complete, reasoned forecast.

In the automatic mode, WILLARD obtains all necessary information from National Meteorological Center data files. External FORTRAN functions were interfaced to WILLARD to access and operate on these data files. The user may specify an area, in which WILLARD will generate a grid of nodal values for the chance of severe thunderstorms for that area. A sample explanation of a forecast is shown in figure A.2.

Validation

FULL EXPLANATION OF THE FORECAST:

Since upper level cold air advection causing increased
upwards vertical velocities is present
it follows that the upper-level destabilisation
potential is sufficient (1)
Since the K Index is strong
when the Lifted Index is strong
it follows that the stability indices condition
is favourable (2)
Since daytime heating acting as a possible trigger mechanism
for potential instability release is strong
when (2) the stability indices condition is favourable
it follows that low-level destabilisation potential
is favourable (3)
Since an approaching 500 millibar short wave trough is present
it follows that the vertical velocity field
is favourable (4)
Since a high 850 mb dew point is present
when surface dew point classification is moderate
it follows that the low-level moisture field
is marginal (5)
Since (1) the upper-level destabilisation potential is sufficient
when (3) low-level destabilisation potential is favourable
and (4) the vertical velocity field is favourable
and (5) the low-level moisture field is marginal
it was necessary to advise:
'There's a MODERATE CHANCE that thunderstorms occurring
12 hours from now will be severe at this location.'
in order to actually forecast the chance of severe thunderstorms

Figure A.2 Sample WILLARD forecast explanation

Steve Zubrick, with help from the US National Weather Service (NWS), has carried out comparisons of WILLARD's forecasting ability versus that of the standard collective outlook issued by forecasters of the Severe Local Storms Unit (SELS) (Zubrick, 1986).

The validation data used spanned a 24 day period in the spring of 1984. Severe thunderstorm outlooks were given by both WILLARD and SELS in terms of the following three aerial density/risk categories.

a) **Slight Risk** - 2 to 5 percent aerial coverage.

b) **Moderate Risk** - 6 to 10 percent aerial coverage.

c) **High Risk** - greater than 10 percent aerial coverage.

Three statistics were quantified over all of the WILLARD and SELS predictions. These were

x - severe storm reports correctly predicted (i.e. those reports found within a severe risk outlook area);

y - severe storm reports not predicted (i.e. those lying outside the severe risk outlook area);

z - non-severe weather predicted to be severe.

In terms of these values WILLARD and the SELS predictions were compared according to three criteria which are believed by meteorologists to give a good indication of predictive skill. These criteria were

1) **Probability of Detection (PoD).** This is defined as

$$PoD = \frac{x}{(x+y)}$$

2) **False Alarm Ratio (FAR).** This is defined as

$$FAR = \frac{z}{(z+x)}$$

3) **Critical Skill Index (CSI).** This is defined as

$$CSI = \frac{x}{(x+y+z)}$$

Figure A.3 shows a comparison of WILLARD and SELS in terms of these 3 criteria over a representative selection of days during the test period.

From the table we see that although WILLARD's probability of detection is generally lower than that of the SELS predictions, it has a generally better false alarm rate. The critical skill index gives us the clearest overall view of skill, and shows WILLARD to have skill which although generally lower than SELS, is still comparable.

Date	PoD (%) SELS/WILLARD	FAR SELS/WILLARD	CSI SELS/WILLARD
25 04 84	82/57	.46/.22	.49/.49
26 04 84	88/20	.44/.41	.52/.81
29 04 84	85/54	.51/.34	.45/.42
25 05 84	100/68	.58/.48	.43/.42
26 05 84	33/91	.74/.90	.17/.10
27 05 84	89/26	.50/.21	.47/.24
04 06 84	70/50	.54/.89	.39/.10
06 07 84	92/71	.44/.31	.54/.54

Figure A.3 Comparison of WILLARD and SELS forecasts

References

Zubrick S. (1984), Willard: a severe thunderstorm forecasting system using RuleMaster by Radian. Austin, Tx: Radian, 1984.

Zubrick S. (1986), Validation of a weather forecasting expert system. To appear in Machine Intelligence 11. Oxford University Press.

Appendix B

EARL

Abstract. In this section we describe Charles Riese's implementation of EARL and discuss the validation testing that has been used to assess EARL's performance. EARL is presently in routine industrial use. The author aided in suggesting the structuring methodology used in EARL.

Introduction

EARL (Riese, 1984) is a system for diagnosing imminent break-down in large oil-cooled electrical transformers. The system was constructed by Charles Riese who is a software engineer working for the Hartford Steam Boiler Company. When EARL was tested against 859 test-cases, its diagnosis was correct in 99.5% of the cases studied.

Large oil-filled transformers are used by utilities for power distribution. These transformers sometimes fail due to insulation deterioration, overheating due to overload, failure of bolted or compression joints, corona, arcing, and overheating from inadvertent grounded core. All of these failure modes involve some form of heating of the oil and/or insulation. These materials decompose when heated and some of the decomposition products are hydrogen and hydrocarbon gases which dissolve in the oil. The concentrations of these gases can be measured with conventional gas chromatographs. Over the past 20 years, techniques have been developed to diagnose transformers' conditions from dissolved gas analyses.

When large transformers (in excess of 10 MW) fail, the service interruption and repair or replacement costs may run into millions of dollars. This provides financial incentive to detect the onset of transformer failures before catastrophic damage occurs. Hartford Steam Boiler Inspection and Insurance Company (HSB) insures industrial equipment and has sponsored the development of a Mugol based expert system which utilises oil sample analyses to prepare transformer condition

reports and to make recommendations on repair action.

The classificatory portion of the expert system contains 27 modules, each having one or more induced rules. Since this is a developing field, the theory relating gas concentrations to faults is not well worked out or documented. It was necessary to rebuild the expert system structure several times, as better organisations of the knowledge became apparent. The induction of rules from examples proved valuable in this rule construction and testing process.

The rules can be divided into several categories. First there are rules to check the validity of data, to determine if there was a leak during sample transport or a chemical analysis error. Other rules determine the presence of failure symptoms: is there low or high temperature heating?, is heating near insulation?, etc. A third set of rules diagnoses particular faults from the symptoms and gas concentrations, and the final set of rules decides which corrective actions to recommend.

The primary system is used for screening the gas analysis results at the chemistry laboratory. Experts seem to make better use of their time, and to be able to check more transformers.

Validation

Owing to the high cost involved in incorrect diagnoses (in the order of millions of dollars), the accepted rate of human diagnostic failure in this domain is below 0.1%.

EARL was tested using 859 test cases for which gas concentration data was available. In 208 of these cases EARL and the expert concurred that a problem existed and that the transformer needed to be overhauled, while in the other 651 cases they both agreed that no problem existed. Out of the 208 cases in which they decided that a problem existed, in 204 cases the expert's explanation was the same as EARL while in the remaining 4 cases the expert's explanation differed from EARL. This highlights the importance of explanation in the "debugging" stage of expert system development. Without explanation, these 4 cases would have been taken as EARL delivering a correct decision, however it would not be realised that it was based on the wrong reasoning; the danger being that later erroneous reasoning could be used to reach the wrong

conclusion, with potentially serious consequences.

In 10 of the 204 cases in which both EARL and the expert agreed on the diagnosis, engineers overhauling the transformers checked to find what the real problem was (this is done rarely as it is very expensive), and in all 10 cases found that EARL and the expert's joint opinion had in fact been correct.

According to these statistics EARL gave the the same advice as the expert for the sam reason in 99.5% of cases tested. In the remaining 0.5% of cases, EARL actually gave the the same advice as the expert, but for different reasons.

It is not known to the author exactly what the estimated cost advantage of using EARL is, nor what the typical rate of inter-expert diagreement is.

Conclusion

EARL is now in full-time field use and automatically drafts textual reports for HSB clients. Both the expert and the knowledge engineer involved in building EARL were satisfied with the Mugol expert system environment. Inductive knowledge acquisition allowed the expert system to be constructed to field test standard in an order of magnitude less time than that expected using dialogue acquisition techniques.

References

Riese C. (1984), Transformer fault detection and diagnosis using RuleMaster by Radian. Austin, Tx: Radian, 1984.

Appendix C

Example move sequences (see ch.8)

Actions

Ba) wK approaches release position (eg. h3) by moving along rank or file

Bb) wK moves to non-check position on direct diagonal which is closest to release position

Bc) wB(light) moves out of corner along its diagonal

Bd) white takes bN

Attributes

B1) white free to take bN

B2) wK on the same diagonal as release position

B3) wBh1 can move

B4) (wK on direct diagonal) and (direct diagonal position closest to release position is covered)

Example move sequences (see ch.8)

Sequence 1 Starts from wKa8 and bN does delaying check

B1	B2	B3	B4	Action	Position	Move
n	n	n	n	Ba	wKa8 wBh1 wBh2 bKf3 bNg2	wKb8
n	n	n	n	Ba	wKb8 wBh1 wBh2 bKf2 bNg2	wKc8
n	y	n	n	Bb	wKc8 wBh1 wBh2 bKf3 bNg2	wKd7
n	y	n	n	Bb	wKd7 wBh1 wBh2 bKf2 bNg2	wKe6
n	y	n	n	Bb	wKe6 wBh1 wBh2 bKf1 bNg2	wKf5
n	y	n	n	Ba	wKf5 wBh1 wBh2 bKf1 bNe3	wKg5
n	n	n	n	Bb	wKg5 wBh1 wBh2 bKf1 bNg2	wKg4
n	y	n	n	Bb	wKg4 wBh1 wBh2 bKf2 bNg2	wKh3
n	-	y	n	Bc	wKh3 wBh1 wBh2 bKf1 bNf3	wBa8

The '-' in the last line allows the algorithm to generalise to the case in which bN releases wB(light).

Sequence 2 Starts from wKb7 and bN does delaying check

B1	B2	B3	B4	Action	Position	Move
n	n	n	n	Ba	wKb7 wBh1 wBh2 bKf2 bNg2	wKc7
n	n	n	n	Ba	wKc7 wBh1 wBh2 bKf3 bNg2	wKd7
n	y	n	n	Bb	wKd7 wBh1 wBh2 bKf2 bNg2	wKe6
n	y	n	n	Bb	wKe6 wBh1 wBh2 bKf1 bNg2	wKf5
n	y	n	n	Ba	wKf5 wBh1 wBh2 bKf1 bNe3	wKg5
n	n	n	n	Bb	wKg5 wBh1 wBh2 bKf1 bNg2	wKg4
n	y	n	n	Bb	wKg4 wBh1 wBh2 bKf2 bNg2	wKh3
n	-	y	n	Bc	wKh3 wBh1 wBh2 bKf1 bNf3	wBa8

Example move sequences (see ch.8)

Sequence 3 Starts from wKb8 and bN does delaying check

B1	B2	B3	B4	Action	Position	Move
n	n	n	n	Ba	wKb8 wBh1 wBh2 bKf2 bNg2	wKc8
n	y	n	n	Bb	wKc8 wBh1 wBh2 bKf3 bNg2	wKd7
n	y	n	n	Bb	wKd7 wBh1 wBh2 bKf2 bNg2	wKe6
n	y	n	n	Bb	wKe6 wBh1 wBh2 bKf1 bNg2	wKf5
n	y	n	n	Ba	wKf5 wBh1 wBh2 bKf1 bNe3	wKg5
n	n	n	n	Bb	wKg5 wBh1 wBh2 bKf1 bNg2	wKg4
n	y	n	n	Bb	wKg4 wBh1 wBh2 bKf2 bNg2	wKh3
n	-	y	n	Bc	wKh3 wBh1 wBh2 bKf1 bNf3	wBa8

Sequence 4 Starts with wKa8 and bN does not do delaying check

B1	B2	B3	B4	Action	Position	Move
n	n	n	n	Ba	wKa8 wBh1 wBh2 bKf3 bNg2	wKb8
n	n	n	n	Ba	wKb8 wBh1 wBh2 bKf2 bNg2	wKc8
n	y	n	n	Bb	wKc8 wBh1 wBh2 bKf3 bNg2	wKd7
n	y	n	n	Bb	wKd7 wBh1 wBh2 bKf2 bNg2	wKe6
n	y	n	n	Bb	wKe6 wBh1 wBh2 bKf1 bNg2	wKf5
n	y	n	n	Bb	wKf5 wBh1 wBh2 bKf2 bNg2	wKg4
n	y	n	n	Bb	wKg4 wBh1 wBh2 bKf1 bNg2	wKh3
n	-	y	n	Bc	wKh3 wBh1 wBh2 bKf2 bNf3	wBa8

Sequence 5 Starts with wKg4

B1	B2	B3	B4	Action	Position	Move
n	y	n	n	Bb	wKg4 wBh1 wBh2 bKf1 bNg2	wKh3
n	-	y	n	Bc	wKh3 wBh1 wBh2 bKf2 bNf3	wBa8

Example move sequences (see ch.8)

Sequence 6 Starts with wKh3

B1	B2	B3	B4	Action	Position	Move
n	-	y	n	Bc	wKh3 wBh1 wBh2 bKf2 bNf3	wBa8

Black plays badly

Sequence 7 Starts with wKa8 after bK has left bN undefended (en prise)

B1	B2	B3	B4	Action	Position	Move
y	-	n	n	Bd	wKa8 wBh1 wBh2 bKe2 bNg2	wB x N!

Sequence 8 Starts with wKa8 and bK leaves bN as first move

B1	B2	B3	B4	Action	Position	Move
n	n	n	n	Ba	wKa8 wBh1 wBh2 bKf3 bNg2	wKb8
y	-	n	n	Bd	wKb8 wBh1 wBh2 bKe3 bNg2	wB x N!

Example move sequences (see ch.8)

Sequence 9 Starts with wKg4 and bK leaves bN as first move

B1	B2	B3	B4	Action	Position	Move
n	y	n	n	Bb	wKg4 wBh1 wBh2 bKf1 bNg2	wKh3
y	-	n	n	Bd	wKh3 wBh1 wBh2 bKe1 bNg2	wB x N!

Sequence 10 Starts with wKb8, bN does not do delaying check but allows the release of wB

B1	B2	B3	B4	Action	Position	Move
n	n	n	n	Ba	wKb8 wBh1 wBh2 bKf2 bNg2	wKc8
n	-	y	n	Bc	wKc8 wBh1 wBh2 bKf2 bNd1	wBc6

Sequence 11 Starts with wKe6, bN does delaying check and then allows the release of wB

B1	B2	B3	B4	Action	Position	Move
n	y	n	n	Bb	wKe6 wBh1 wBh2 bKf1 bNg2	wKf5
n	y	n	n	Ba	wKf5 wBh1 wBh2 bKf1 bNe3	wKg5
n	-	y	n	Bc	wKg5 wBh1 wBh2 bKf1 bNd1	wBc6

Example move sequences (see ch.8)

Sequence 12 Starts with wKe6, bN does delaying check and then allows itself to be taken (by moving to g4).

B1	B2	B3	B4	Action	Position	Move
n	y	n	n	Bb	wKe6 wBh1 wBh2 bKf1 bNg2	wKf5
n	y	n	n	Ba	wKf5 wBh1 wBh2 bKf1 bNe3	wKg5
y	-	n	n	Bd	wKg5 wBh1 wBh2 bKf1 bNg4	wKg4!

Sequence 13 Starts with wKd7, bN checks allowing itself to be taken by wB(dark)

B1	B2	B3	B4	Action	Position	Move
n	y	n	n	Bb	wKd7 wBh1 wBh2 bKf2 bNg2	wKe6
y	y	y	n	Bd	wKe6 wBh1 wBh2 bKf2 bNf4	B x N

Appendix D

Result of sequence induction (see ch.8)

Actions

Ba) wK approaches release position (eg. h3) by moving along rank or file

Bb) wK moves to non-check position on direct diagonal which is closest to release position

Bc) wB(light) moves out of corner along its diagonal

Bd) white takes bN

Attributes

B1) white free to take bN

B2) wK on the same diagonal as release position

B3) wBh1 can move

B4) (wK on direct diagonal) and (direct diagonal position closest to release position is covered)

B1	B2	B3	B4		(Action,NextState)
STATE 0					
n	-	y	n	=>	(Bc,GOAL)
n	n	n	n	=>	(Ba,1)
n	y	n	n	=>	(Bb,2)
y	-	n	n	=>	(Bd,GOAL)
STATE 1					
n	-	y	n	=>	(Bc,GOAL)
n	n	n	n	=>	(Ba,1)
n	y	n	n	=>	(Bb,2)
y	-	n	n	=>	(Bd,GOAL)
STATE 2					
n	-	y	n	=>	(Bc,GOAL)
n	y	n	n	=>	(Bb,2)
n	y	n	y	=>	(Ba,3)
y	-	n	n	=>	(Bd,GOAL)
y	y	y	n	=>	(Bd,GOAL)
STATE 3					
n	-	y	n	=>	(Bc,GOAL)
n	n	n	n	=>	(Bb,4)
y	-	n	n	=>	(Bd,GOAL)
STATE 4					
n	y	n	n	=>	(Bb,2)

Appendix E

Automata after ID3-like induction (see ch.8)

Actions

Ba) wK approaches release position (eg. h3) by moving along rank or file

Bb) wK moves to non-check position on direct diagonal which is closest to release position

Bc) wB(light) moves out of corner along its diagonal

Bd) white takes bN

Attributes

B1) white free to take bN

B2) wK on the same diagonal as release position

B3) wBh1 can move

B4) (wK on direct diagonal) and (direct diagonal position closest to release position is covered)

```
STATE 0
[B1]
        y : => ( Bd, GOAL )
        n : [B3]
            y : => ( Bc, GOAL )
            n : [B2]
                y : => ( Bb, 2 )
                n : => ( Ba, 1 )
STATE 1
[B1]
        y : => ( Bd, GOAL )
        n : [B3]
            y : => ( Bc, GOAL )
            n : [B2]
                y : => ( Bb, 2 )
                n : => ( Ba, 1 )
STATE 2
[B1]
        y : => ( Bd, GOAL )
        n : [B3]
            y : => ( Bc, GOAL )
            n : [B4]
                y : => ( Ba, 3 )
                n : => ( Bb, 2 )
STATE 3
[B1]
        y : => ( Bd, GOAL )
        n : [B4]
            y : => ( Bc, GOAL )
            n : => ( Bb, 4 )
STATE 4
( Bb, 2 )

GOAL
```

Appendix F

KBBKN Mugmaker induction file (see ch.8)

```
MODULE: wKsupp
        DECLARATIONS:
[       INTENT: "move wK to support the attack of wBh1 on bNg2"]

        ACTIONS:
                wKapp       [prints "wK approaches release position (eg. h3)\\n" ;
                             advise "       by moving along rank or file"]
                wKnonc      [prints "wK moves to non-check position on direct\\n";
                             prints "       diagonal which is closest to\\n";
                             advise "       the release position (eg. h3)"]
                wBout       [print "wB(light) moves out of corner along its\\n
                             advise "       diagonal"]
                wtakes      [advise "white takes bN"]
        CONDITIONS:
                wcantake    [ask "Is white free to take bN? " "y,n"]
                                                {y n}
                wKinck                  [ask "Is the wK in check? " "y,n"]
                                                {y n}
                wKond       [ask "Is wK on same diagonal as release position? "
                                "y,n"] {y n}
                wBcanmv     [ask "Can the cornered wB now move? " "y,n"]
                                                {y n}
                diagcvr     [prints "Is the wK on the direct diagonal and\\n";
                             prints "  the direct diagonal position closest\\n";
                             ask "  to the release position is covered\\n" "y,n"]
                                                {y n}
        STATE: zero
                EXAMPLES:

                n       n       -       y       n       =>      (wBout,GOAL)
                n       n       n       n       n       =>      (wKapp,1)
                n       n       y       n       n       =>      (wKnonc,2)
                y       n       -       n       n       =>      (wtakes,GOAL)

        STATE: one
                EXAMPLES:

                n       n       n       n       n       =>      (wKapp,1)
                n       n       y       n       n       =>      (wKnonc,2)
                y       n       -       n       n       =>      (wtakes,GOAL)

        STATE: two
                EXAMPLES:

                n       -       y       n       y       =>      (wKapp,3)
                n       n       -       y       n       =>      (wBout,GOAL)
                n       n       y       n       n       =>      (wKnonc,2)
                y       n       -       n       n       =>      (wtakes,GOAL)

        STATE: three
                EXAMPLES:

                n       n       -       y       n       =>      (wBout,GOAL)
                n       n       n       n       n       =>      (wKnonc,4)
                y       n       -       n       n       =>      (wtakes,GOAL)

        STATE: four
                EXAMPLES:

                n       n       y       n       n       =>      (wKnonc,2)
```

Appendix G

KBBKN Mugol code generated from Appendix F (see ch.8)

The following Mugol module was generated from the Mugmaker file given in Appendix F. The module was partially verified against the KBBKN data-base (see section 8.2) using a sample set of sequences. However, the test sequences used were not exhaustive.

```
MODULE wKsupp IS
        INTENT: "move wK to support the attack of wBh1 on bNg2"

  STATE: zero
   IF (ask "Is white free to take bN? " "y,n") IS
     "y" : ( advise "White takes bN", GOAL )
     ELSE IF (ask "Can the cornered wB now move? " "y,n") IS
        "y" : ( advise "wB(light) moves out of corner along its diagonal", GOAL )
        ELSE IF (ask "Is wK on same diagonal as release position? "
                                              "y,n") IS
           "y" : ( advise "wK moves to non-check position on direct diagonal which is closest to the release position (eg. h3)", two )
           ELSE ( advise "wK approaches release position (eg. h3) by moving along rank or file", one )

  STATE: one
   IF (ask "Is white free to take bN? " "y,n") IS
     "y" : ( advise "White takes bN", GOAL )
     ELSE IF (ask "Can the cornered wB now move? " "y,n") IS
        "y" : ( advise "wB(light) moves out of corner along its diagonal", GOAL )
        ELSE IF (ask "Is wK on same diagonal as release position? "
                                              "y,n") IS
           "y" : ( advise "wK moves to non-check position on direct diagonal which is closest to the release position (eg. h3)", two )
           ELSE ( advise "wK approaches release position (eg. h3) by moving along rank or file", one )

  STATE: two
   IF (ask "Is white free to take bN? " "y,n") IS
     "y" : ( advise "White takes bN", GOAL )
     ELSE IF (ask "Can the cornered wB now move? " "y,n") IS
        "y" : ( advise "wB(light) moves out of corner along its diagonal", GOAL )
        ELSE IF (ask "Is the wK on the direct diagonal and the direct diagonal position closest to the release position is covered? " "y,n") IS
           "y" : ( advise "wK approaches release position (eg. h3) by moving along rank or file", three )
           ELSE ( advise "wK moves to non-check position on direct diagonal which is closest to the release position (eg. h3)", two )

  STATE: three
   IF (ask "Is white free to take bN? " "y,n") IS
     "y" : ( advise "White takes bN", GOAL )
     ELSE IF (ask "Is the wK on the direct diagonal and the direct diagonal position closest to the release position is covered? " "y,n") IS
        "y" : ( advise "wB(light) moves out of corner along its diagonal", GOAL )
        ELSE ( advise "wK moves to non-check position on direct diagonal which is closest to the release position (eg. h3)", four )

  STATE: four
   ( advise "wK moves to non-check position on direct diagonal which is closest to the release position (eg. h3)", two )

GOAL OF wKsupp
```

Appendix H

GENARCH: a practical solution to general arch building

by Barry Shepherd

Abstract. The problem solved by GENARCH is an extension of the simple arch problem of ARCH (chapter 4). Again the arch consists of two piles and a beam, but now the piles can contain any number of blocks which can be of any size, although the final heights of the piles must be equal. In addition each block within a pile can itself be an arch, and the beam can also be an arch. This nesting can be to any level.

Introduction.

A strategy for building a simple arch in a blocks world has already been generated using the Mugol environment (Michie, Muggleton, Riese and Zubrick, 1984). This Mugol solution is called ARCH (see chapter 4 of main thesis) and is based on a strategy first proposed by Dechter and Michie (1984). The ARCH problem consists of assembling an arch using four equal height blocks and a single beam. The blocks are assembled into two piles of two blocks in a specific order, and the beam is placed to bridge these piles. Initially the blocks are stacked in any order on two work piles, and the beam is placed in a beam-store. Blocks must be cleared (if necessary) before they can be moved. Blocks cleared from the top of other blocks can only be placed on one of the work piles.

ARCH cannot be considered a practical solution since it solves the problem only in simulation and then only in a symbolic manner. Although the planning aspects of the problem are solved the (mainly numerical) complexities of actually directing a robot to pick up and move the various blocks are not tackled. GENARCH is a practical solution to a more general arch building problem and has also been generated using the Mugol environment. GENARCH can be run either in

simulation or in a real environment using a Puma 200 robot.

The problem solved by GENARCH is an extension of the simple arch problem considered by ARCH. Again the arch consists of two piles and a beam, but now the piles can contain any number of blocks which can be of any size, although the final heights of the piles must be equal. In addition each block within a pile can itself be an arch, and the beam can also be an arch. This nesting can be to any level. Initially the blocks and beams are stacked in any order in any number of work_piles.

An example of a structure which can be categorised as a general arch is shown in figure H.1.

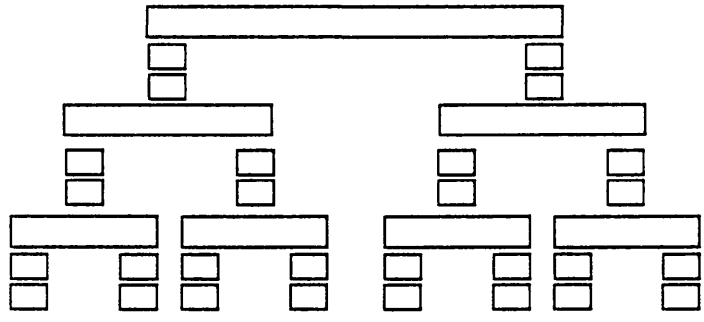


Figure H.1 Example of a general arch

Structure of the solution.

The problem is divided into a hierarchy of sub-problems for Mugol, this hierarchy is shown in figure H.2. The Mugol modules "build_arch", "onto", "pick_up" and "place_at" perform high-level actions. The lowest level actions which are specified in Mugol are the set of robot primitives:

move_to(x y z o a t)

grasp

release

home

set_speed

These robot primitives are coded in "C" and provide a basic device-independent interface between Mugol and the robot chosen to perform the task. At present they have been created for a Puma

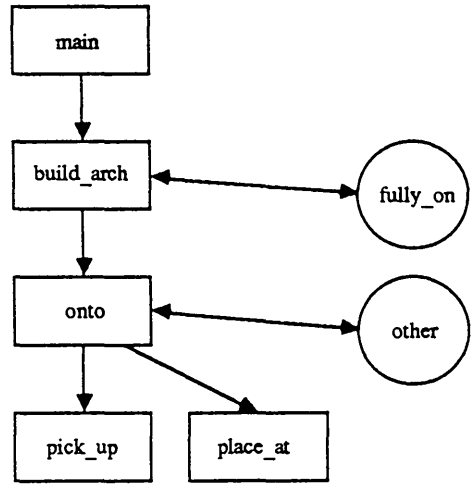


Figure H.2 Solution structure

200, and also a Rhino XR-1.

The Mugol modules "fully_on" and "other" return high-level, problem-specific information about the current state of the assembly task. However GENARCH is a "blind" solution and thus requires detailed knowledge of the state of the world at all times. This knowledge must be both relational (eg "what's on red_block1?") and also numerical (eg. the position, orientation and size of red_block1). A "C" coded world model has been created in order to store this information. This is described in more detail in a later section. The following primitives are an example of the interface between Mugol and this world-model:

```
above(object_name)
top_of(pile_name)
property(object_name,property_name)
```

Note: all objects (whether blocks, beams, piles, arches or places) are stored in the same manner in the world model and are referenced by Mugol using only their names.

The Mugmaker induction modules.

Listings of all of the Mugol modules are given later.

The action module build_arch

A major part of the solution is the module "build_arch". This takes as its input the name of a place where an arch is to be built and the names of all of the components of the arch.

```
build_arch(location, left_list, right_list, beam)
```

Where left_list and right_list are lists of the names of those objects (in the correct order) which are to be used to make the arch left_pile and right_pile (call these component lists), beam is the name of the object to be used as the arch beam.

Eg:

build_arch("tabletop" "[redblk,greenblk]" "[blueblk,blueblk]" "yellowbeam")

In summary build_arch examines each constituent object in turn and checks to see if it is in position (using the module "fully_on"), if not it moves the object to the correct position (using the module "onto"), this may entail clearing the top of the object if it is not clear.

The query module fully_on.

In the module "build_arch" an object in the left or right pile is assumed to be in its final position if it is on top of the object which occurred before it in the object list for that pile. The beam is in position if it is on top of both the left pile and the right pile. The module:

object fully_on place

checks to see if the named object (whether a primitive object or a structure) is fully located on the stated place. If the object is a primitive object than the world_model primitive "above" can be used on its own to answer this question, if the object is itself an arch then its status (ie fully_assembled or partially assembled) must also be checked.

The action module onto.

The module:

object_name onto place/object_name

moves an object (a primitive or a structure) onto another object (a primitive object, structure or place). If the object to be moved is a primitive object and if it is clear then it can be picked up by the robot (module "pickup") and placed on the correct location (module "place_at"). If the object is a structure (ie an arch) then it cannot be moved as a single entity by the robot but instead must be assembled (using "build_arch") at the specified location.

Mugol modules.

The modules *pick_up*, *place_at* and *other* were not derived inductively but were written directly in Mugol. *Pick_up* takes as an argument an object name extracts its position and orientation from the world model and and generates a series of "move_to" instructions in order to move the robot gripper horizontally from its home position to a point directly above the object and then down onto the object, close the gripper and return back the way it came. *Place_at* takes as its argument the name of a place and operates similarly to *Pick_up* in order to put the held object on the place so that the centroid of the object is directly above that of the place and their orientations (the orientation of a block is the direction of its principal axis) align. *Other* takes the name of an object as argument and returns the name of the smallest work_pile other than the one in which it is currently located.

The world model.

The world model consists of a hierarchy of lists. Each object in the problem, whether a primitive object (eg a block) or a structure (eg an arch) or a place (eg a work_pile) is represented in the model by a list. This list contains the properties of that object and also sub_lists representing the objects contained within that object (none if its a primitive object). An object can have any number of properties but most objects will possess:-

name

type (eg: cuboid,sphere,disc,pile,arch,surface etc)

position(x,y,z,) - usually the position of its centroid.

orientation(o,a,t) - the orientation of its principle axis.

dimensions (length,width,height)

Examples of other properties are colour, material, weight etc.

Any property of an object can be extracted from the world model using the primitive :

property(object_name, property_name)

Frequently used properties can also be extracted using individual primitives eg: X(object_name),Len(object_name).

All objects are referenced using their name. Object names do not have to be unique, eg. many identical block's can all be called "redblock", however distinct objects do require individual entries. When an object is referenced the first object found when doing a search of the world-model will be the one which is used. If a particular object is required then its "path" in the world-model (which is unique) can be used instead of the object name.

Relational information can be obtained from the position of the object in the world model. For example a structure consisting of a pile of blocks will be represented in the world model by a list which will contain (in addition to its own properties) a sub-list for all of those objects contained in the pile. The order in which these occur is the physical order in which they are stacked on top of each other. This ordering has different meanings for different types of structure/place etc.

The following primitives are used by GENARCH to extract relational information from the model:

above(object_name)

bottom_of(pile_name) - bottom of a pile or stack.

top_of(pile_name) - top of a pile or stack.

In order to initialise the model the following primitive is used:

new_object(place_name, object_name, property_list)

This will create a new object with the name and properties specified and insert into the world model at the specified place.

The movement of objects within the model resulting from real actions performed by the robot is

GENARCH: a practical solution to general arch building

achieved using:

model_to_hand(object_name)

This extracts the stated object, it can now be referenced using the name "held_object".

hand_to_model(place_name)

insert the held object into the stated place.

Specifying a particular task.

The modules described so far contain a strategy for building a general arch given the names of the components of that arch. Hence a specific arch could be built with the following "main" module:

```
MODULE: GENARCH

 STATE: start
(       initialise_model;
        build_arch("table_top" "[redblk,redblk]" "[blueblk,blkblk]" "grnbeam"),
        goal
)

GOAL OF GENARCH
```

where "initialise_model" creates entries in the world model for all of the primitive objects (ie the blocks and beams), the work-piles and the place the arch is to be built on (ie "table_top"). If one component of this arch was another arch then the above "main" module would be inadequate. This is solved by creating what can be called a "task description module" which gives the names and

composition of all of the structures in the complete arch. The task description module for the arch shown in the introduction (tower1) is given below.

```
MODULE: build

        INTENT:"build $1 at $2"
        IN:     string    {object,place}

        STATE: start

        if (object) is

        "tower": (build_arch(object place "[tow_L]" "[tow_R] "towbm"),goal)
        "tow_L": (build_arch(object place "[towLL]" "[towLR] "towLbm"),goal)
        "tow_R": (build_arch(object place "[towRL]" "[towRR] "towRbm"),goal)
        "towLL": (build_arch(object place "[Y1,Y2]" "[Y3,Y4] "Ybm1"),goal)
        "towLR": (build_arch(object place "[B1,B2]" "[B3,B4] "Bbm1"),goal)
        "towRL": (build_arch(object place "[Y5,Y6]" "[Y7,Y8] "Ybm2"),goal)
        "towRR": (build_arch(object place "[B5,B6]" "[B7,B8] "Bbm2"),goal)
        "towbm": (build_arch(object place "[G1,G2]" "[G3,G4] "Gbm1"),goal)
        "towLbm": (build_arch(object place "[R1,R2]" "[R3,R4] "Rbm1"),goal)
        ELSE    (build_arch(object place "[R5,R6]" "[R6,R7] "Rbm2"),goal)

GOAL OF build
```

The "main" module for GENARCH can now be

GENARCH: a practical solution to general arch building

MODULE: GENARCH

```
STATE: start
(       initialise_model;
        build("tower" "table_top"),
        goal
)
```

GOAL OF GENARCH

(Note: the module "onto" is changed to call the "build" module instead of the "build_arch" module when a structure is to be moved).

This mechanism for defining arches can easily be extended to other structures, eg: walls, steps, pyramids etc. For example in the arch defined above the component: towL can be defined as a wall by replacing its definition line with:

```
"towL": (build_wall(place name "redblock" 6 5));
```

where build_wall is a strategy for building a wall using redblocks which is 6 blocks long and 5 blocks high.

Example structures.

The structure shown in the introduction (tower1) contains 28 blocks and 7 beams, one particular goal state is shown in figure H.3 and this has been built using the Puma from an initial state also shown below.

Another structure which has been build using GENARCH (simulation only so far) is shown in figure H.5 (tower2). One set of initial positions is also shown, this initial state required 83 moves in order to reach the goal.

The task specification module for "tower2" is shown below, note that this structure can be built by simply replacing the task specification module for "tower1" with that for "tower2".

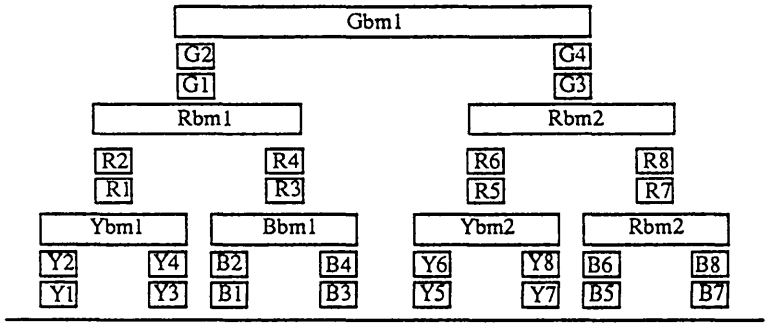


Figure H.3 A goal state for tower1

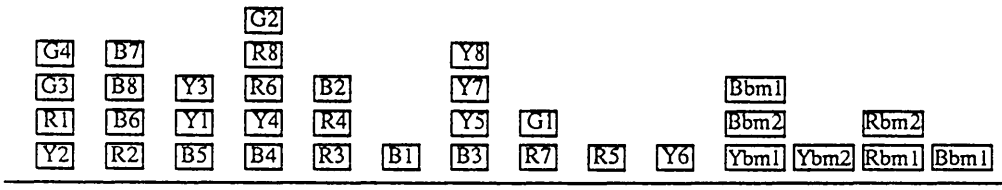


Figure H.4 An initial state for tower1

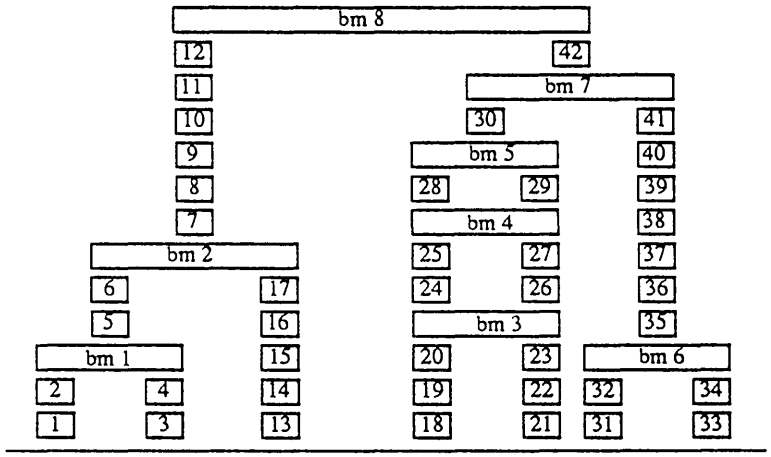


Figure H.5 Goal state for tower 2

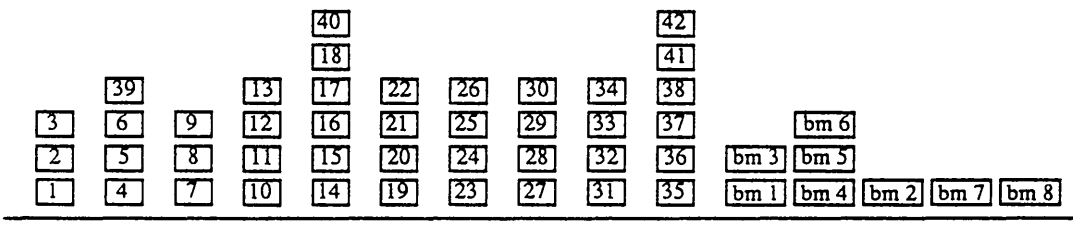


Figure H.6 Initial state for tower2

Task specification module for tower2.

```
MODULE build IS

    INTENT:        "construct $1 at $2"
    IN:    string  {name,place}

  STATE: do_it
   IF (name) IS
   "tower" : ( build_arch(name place "[tow_L]" "[tow_R]" "bmTm" ) , goal )
   "tow_L" : ( build_arch(name place "[tow_LL]" "[13,14,15]" "bmTL") , goal )
   "tow_R" : ( build_arch(name place "[tow_RL1,tow_RL2,tow_RL3]"
                                  "[tow_RR,35,36,37,38,39,40]" "bmTR" ), goal )
   "bmTm"   : ( build_arch(name place "[07,08,09,10,11,12]" "[42]" "bm08" ),goal)
   "tow_LL"  : ( build_arch(name place "[01,02]" "[03,04]" "bm01" ) , goal )
   "bmTL"   : ( build_arch(name place "[05,06]" "[16,17]" "bm02" ) , goal )
   "tow_RL1" : ( build_arch(name place "[18,19,20]" "[21,22,23]" "bm03" ) , goal )
   "tow_RL2" : ( build_arch(name place "[24,25]" "[26,27]" "bm04" ) , goal )
   "tow_RL3" : ( build_arch(name place "[28]" "[29]" "bm05" ) , goal )
   "tow_RR"  : ( build_arch(name place "[31,32]" "[33,34]" "bm06" ) , goal )
   ELSE ( build_arch(name place "[30]" "[41]" "bm07" ) , goal )

GOAL OF build
```

References

Dechter R., and Michie D. (1984), Induction of plans. Glasgow: The Turing Institute (TIRM-84-006).

Mugol modules.

```
MODULE pick_up IS
        IN:    string      object
               LOCAL:      float  ht

        STATE:       start
        (
               above (ht + "15") object ;
               release ;
               above (ht + "5") object ;
               above (ht + "2.5") object ;
               above (ht + "0.5") object ;
               robot_speed("slow"),
               grabit
        )

        STATE: grabit

               if (ht > "1") is
               "T":  (above (ht - "1") object , backof)
```

```
        ELSE  (above (ht/"2") object , backof)

    STATE: backof
    (
        grasp ;
        above (ht + "0.5") object ;
        robot_speed("medium");
        above (ht + "2.5") object ;
        above (ht + "5") object ;
        above (ht + "15") object;
        Model_to_hand object,
        goal
    )

GOAL OF pick_up

MODULE place_at IS

    IN:   string     place
    LOCAL:     float  ht

    STATE: start
    (
        Ht "held_object" -> ht ;
        (Ht place) + ht  -> ht ;
        above (ht + "15") place ;
        above (ht + "5") place ;
        above (ht + "2.5") place ;
```

```
      above (ht + "0.5") place ;
      robot_speed("slow") ,
      releaseit
)

STATE: releaseit

      if (ht > "1") is
      "T":  (above (ht - "1") place , backof)
      ELSE  (above (ht/"2") place , backof)


STATE: backof
(
      release ;
      above (ht + "0.5") place ;
      robot_speed("medium");
      above (ht + "2.5") place ;
      above (ht + "5") place ;
      above (ht + "15") place;
      Data_update("held_object" "X" ftos(X place)) ;
      Data_update("held_object" "Y" ftos(Y place)) ;
      Data_update("held_object" "Z" ftos(Z place + (Ht place)))
      Data_update("held_object" "Hd" ftos( Hd place)) ;
      Data_update("held_object" "Az" ftos( Az place)) ;
      Data_update("held_object" "Ro" ftos( Ro place)) ,
      Data_update(place "Ht" ftos((Ht place) + ht)) ;
      Hand_to_model place,
```

```
            goal
        )

GOAL OF place_at

MODULE other IS
        IN:   string place
        OUT:  string best_place
        LOCAL: string toplace
        LOCAL: list plist

      STATE: get_top_place
      ( "[]" -> plist ;
              place !< plist -> plist;
              headof plist -> toplace,
              decide
      )

      STATE: decide
       IF (toplace) IS
          "p1" : ( smallest("[p2,p3,p4,p5,p6,p7,p8,p9,p10]") -> best_pl
"p2" : ( smallest("[p1,p3,p4,p5,p6,p7,p8,p9,p10]") -> best_place, GOAL )
          "p3" : ( smallest("[p1,p2,p4,p5,p6,p7,p8,p9,p10]") -> best_pl
"p4" : ( smallest("[p1,p2,p3,p5,p6,p7,p8,p9,p10]") -> best_place, GOAL )
          "p5" : ( smallest("[p1,p2,p3,p4,p6,p7,p8,p9,p10]") -> best_pl
"p6" : ( smallest("[p1,p2,p3,p4,p5,p7,p8,p9,p10]") -> best_place, GOAL )
          "p7" : ( smallest("[p1,p2,p3,p4,p5,p6,p8,p9,p10]") -> best_pl
"p8" : ( smallest("[p1,p2,p3,p4,p5,p6,p7,p9,p10]") -> best_place, GOAL )
```

```
          "p9" : ( smallest("[p1,p2,p3,p4,p5,p6,p7,p8,p10]") -> best_pl
"p10" : ( smallest("[p1,p2,p3,p4,p5,p6,p7,p8,p9]") -> best_place, GOAL )
          "b1" : (  "b2" -> best_place, GOAL )
          ELSE (  "b1" -> best_place, GOAL )

  GOAL OF other
```

Mugol robot primitives.

```
  /* This provides the Mugol interface to the robot primitives
  /* All code below this level is device dependent & coded in C */

  PRIMITIVE MODULE init_robot is
        INTENT:     "initialise the robot arm"
        OUT: string status
  GOAL of init_robot

  PRIMITIVE MODULE robot_home is
        INTENT:     "put robot arm in its home position"
        OUT: string      status
  GOAL of robot_home

  PRIMITIVE MODULE move_to is
        IN:   float {x,y,z,h,a,r}
        OUT: string      status
  GOAL of move_to

  PRIMITIVE MODULE grasp is
        INTENT:     "close the robot gripper onto an object"
  GOAL of grasp
```

```
PRIMITIVE MODULE release is
      INTENT: "fully open the robot gripper"
GOAL of release

PRIMITIVE MODULE robot_speed is
      INTENT: "change the speed of the robot"
      IN:   string      speed
GOAL of robot_speed

PRIMITIVE MODULE robot_wait is
      INTENT: "wait for the robot to stop moving"
GOAL of robot_wait
```

Mugol world-model primitives.

```
/* This provides the Mugol interface to the model primitives. */

/*-------------------------------------*/
/* Model update and access routines */

PRIMITIVE MODULE Init_model is
      INTENT:     "initialise the world model"
GOAL of Init_model

PRIMITIVE MODULE Pr_model is
      INTENT:     "print the world model"
GOAL of Pr_model

PRIMITIVE MODULE Draw_model is
      INTENT:     "draw the world model"
GOAL of Draw_model
```

```
PRIMITIVE MODULE New_object is
      INTENT:     "create a new object called $2 in $1"
      IN:   string      {where,name,data}
GOAL of New_place

PRIMITIVE MODULE Insert is
      INTENT:     "insert $1 into $2"
      IN:   string      {object,place}
GOAL of Insert

PRIMITIVE MODULE Extract is
      INTENT:     "extract $1"
      IN:   string      object
GOAL of Extract

PRIMITIVE MODULE Path is
      INTENT:     "find the path name of the object $1"
      IN:   string      objname
      OUT:  string      answer
GOAL of Path

PRIMITIVE MODULE Is_in is
      INTENT:     "determine if $1 is in $2"
      IN:   string      {object,place}
      OUT:  boolean     answer
GOAL of Is_in

PRIMITIVE MODULE Model_to_hand is
      INTENT:     "move $1 from the model to the hand-store"
```

```
      IN:   string      object
GOAL of Model_to_hand

PRIMITIVE MODULE Hand_to_model is
      INTENT:     "move $1 from the hand-store to the model"
      IN:   string      place
GOAL of Hand_to_model

/* ------------------------------------------------------------*/
 * these are not  C coded routines but are an integral part
 * of accessing the world model.
 */

MODULE build_arch.In is
      INTENT:     "add $1 to the end of $2"
      IN:   string      {name,path}
      OUT:  string      newpath

      STATE:      begin
            if name is
            "":   (path -> newpath,goal)
            ELSE  (path main.# "," main.# name -> newpath,goal)
GOAL of In

MODULE Locn is
      INTENT:     "the location of $1"
      IN:   string      path
      OUT:  string      locn
      LOCAL:      list  plist
```

```
STATE:      begin
        if path is
        "":   ("" -> locn,goal)
        ELSE  ("[]" -> plist;
               path !< plist -> plist;
               headof plist -> locn;
               tailof plist -> plist, loop)

STATE:      loop
        if plist is
        "[]": (null,goal)
        ELSE  (locn main.# "," main.# (headof plist) -> locn;
               tailof plist -> plist, loop)
GOAL of Locn

MODULE Name is
      INTENT:     "the name of $1"
      IN:   string      path
      OUT:  string      name
      LOCAL:      list  plist

      STATE:      begin
            if path is
            "":   ("" -> name,goal)
            ELSE  ("[]" -> plist;
                   path !< plist -> plist;
                   headof (reverse plist) -> name, goal)
GOAL of Name
```

```
/* ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Model query routines... Probably problem dependent */

PRIMITIVE MODULE Above is
      INTENT:     "find the name of the object on top of $1"
      IN:   string      objname
      OUT:  string      answer
GOAL of Above

PRIMITIVE MODULE Top_of is
      INTENT:     "find the name of the top member of the stack $1"
      IN:   string      name
      OUT:  string      answer
GOAL of Top_of

PRIMITIVE MODULE Bottom_of is
      INTENT:     "find the name of the bottom member of the stack $1"
      IN:   string      name
      OUT:  string      answer
GOAL of Bottom_of

PRIMITIVE MODULE Place_len is
      INTENT:     "find the num of items in $1"
      IN:   string      name
      OUT:  integer answer
GOAL of Place_len

PRIMITIVE MODULE Data_len is
      INTENT:     "find the number of data items in $1"
```

```
      IN:   string      name
      OUT:  integer answer
GOAL of Data_len

PRIMITIVE MODULE Type_of is
      INTENT:     "find the type $1"
      IN:   string      object
      OUT:  string      answer
GOAL of Type_of


PRIMITIVE MODULE X is
      INTENT:     "find the X co-ord of $1"
      IN:   string      object
      OUT:  float  answer
GOAL of X

PRIMITIVE MODULE Y is
      INTENT:     "find the Y co-ord of $1"
      IN:   string      object
      OUT:  float  answer
GOAL of Y

PRIMITIVE MODULE Z is
      INTENT:     "find the Z co-ord of $1"
      IN:   string      object
      OUT:  float  answer
GOAL of Z

PRIMITIVE MODULE Hd is
```

```
        INTENT:     "find the heading of $1"
        IN:   string      object
        OUT:  float  answer
GOAL of Hd

PRIMITIVE MODULE Az is
        INTENT:     "find the azimuth of $1"
        IN:   string      object
        OUT:  float  answer
GOAL of Az

PRIMITIVE MODULE Ro is
        INTENT:     "find the rotation of $1"
        IN:   string      object
        OUT:  float  answer
GOAL of Ro

PRIMITIVE MODULE Len is
        INTENT:     "find the length of $1"
        IN:   string      object
        OUT:  float  answer
GOAL of Len

PRIMITIVE MODULE Wid is
        INTENT:     "find the width of $1"
        IN:   string      object
        OUT:  float  answer
GOAL of Wid
```

```
PRIMITIVE MODULE Ht is
        INTENT:     "find the height of $1"
        IN:   string      object
        OUT:  float  answer
GOAL of Ht

PRIMITIVE MODULE Status_of is
        INTENT:     "determine the status of $1"
        IN:   string      object
        OUT:  string      answer
GOAL of Status_of

PRIMITIVE MODULE Data_item is
        INTENT:     "determine the $2 th data item in $1"
        IN:   string      object
        IN:   integer     indx
        OUT:  string      answer
GOAL of Data_item

PRIMITIVE MODULE Data_update is
        INTENT:     "update the $2 th data item in $1 with $3"
        IN:   string      object
        IN:   string      name
        IN:   string      value
GOAL of Data_update

PRIMITIVE MODULE Set_global is
        INTENT:     "create/update the global $1 with the value $2"
        IN:   string      name
```

```
      IN:   string      value
GOAL of Set_global

PRIMITIVE MODULE Global is
      INTENT:     "retrieve the value of the global $1"
      IN:   string      name
      OUT:  string      value
GOAL of Global
```