ORIGINAL

KNOWLEDGE ACQUISITION AND REFINEMENT TOOLS FOR THE ADVISE META-EXPERT SYSTEM

BY

ROBERT EUGENE REINKE

July 1984

File No. UIUCDCS-F-84-921

ISG 84-4

File No. UIUCDCS-F-84-921

KNOWLEDGE ACQUISITION AND REFINEMENT TOOLS FOR THE ADVISE META-EXPERT SYSTEM

BY

ROBERT EUGENE REINKE

B.S., University of Illinois, 1982
B.S., University of Illinois, 1982

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1984

ISG 84-4

Urbana, Illinois

July 1984

This research was supported in part by the Office of Naval Research under grant N00014-82-K-0186 and in part by the National Science Foundation under grant MCS 82-05166.

1. INTRODUCTION

Computer systems which can perform a task that normally requires a highly-trained human expert are understandably in great demand. Unfortunately, the development of such systems has proven to be difficult. Building an expert system typically requires months (or years) of work by both the human expert who is providing knowledge and the computer scientist (knowledge engineer) attempting to put that knowledge into a computer program. The primary problem lies in the area of *knowledge acquisition and refinement.* Computer programs that aid a domain expert in expressing and modifying knowledge would decrease the amount of time necessary to build useful expert systems.

This thesis describes software tools that are the first steps along the path to an integrated knowledge acquisition and refinement system. The research described here is based on the belief that a teacher (i.e., the domain expert) should be provided with several ways to present and modify knowledge. The goal of this research is to develop a domain independent inference system that can be taught in several different ways.

Two specific computer programs are described. The first, GEM (Generalization of Examples by Machine), is the newest in a series of inductive inference programs developed by the Artificial Intelligence Laboraotory at the University of Illinois [Michalski 77, Michalski and Larson 78, Stepp 79, Hoff, Michalski and Stepp 83]. The second program, ATEST, allows rapid, batch-type testing of a knowledge base. ATEST was developed specifically to aid in rule base refinement. Both of these programs are intended, in their final implementation, to be part of the ADVISE general purpose inference system [Michalski and Baskin 83, Michalski et. al. 84].

This chapter presents an overview of previous knowledge base development work and a discussion of specific problems in knowledge acquisition. This will establish the context for the ideas in Chapter 2 on a paradigm for rule base development. Chapter 3 describes the rule formalism used throughout the remainder of the thesis, and Chapter 4 contains a detailed description of the GEM and ATEST programs. The last two chapters present results from the application of these programs to three domains.

ACKNOWLEDGEMENTS

I would first like to thank my thesis advisor, Dr. R.S. Michalski, for many useful ideas, comments and suggestions. He provided the motivation and the initial impetus for this thesis. Thanks also go to Dr. A.B. Baskin for much help and many insights into the problems involved in building expert systems. The members of the ADVISE working group, Carl Uhrik, Lance Rodewald, Kent Spackman and Mark Seyler, contributed intellectually and materially to this thesis and to the software described here. Bob Stepp, Jeff Becker and J.R. Hong provided useful information and ideas about the learning problem and the AQ algorithm. Thanks go to Tom Channic, Dave Hammerslag, Bob Stepp and Carl Uhrik for proof-reading.

This thesis would not have been possible without the tremendous support, both emotional and financial, provided by my parents. They were always there when I needed them.

This research was supported in part by the Office of Naval Research under grant NOOO 14-82-K-0186 and in part by the National Science Foundation under grant MCS 82-05166.

TABLE OF CONTENTS

1.1. An Overview of Expert Systems and Knowledge Acquisition

An expert system contains both procedural and declarative knowledge. Declarative knowledge is stored in a knowledge base. Two methods have been widely used in building knowledge bases: production rules and semantic networks. Systems such as MYCIN [Davis, Buchanan and Shortliffe 77], R1 [McDermott 82] and PLANT/ds [Michalski et. al. 83] use production rules to represent a domain expert's decision making rules. Expert systems such as Prospector [Gaschnig 82] and BABY [Rodewald 84] use semantic networks to represent real-world situations and phenomena. Procedural knowledge is built into the control scheme, a deductive inference mechanism which uses the knowledge base. The expert system (directed by the control scheme) asks questions of the user and applies information to the knowledge base in order to derive conclusions. This separation of procedural and declarative knowledge is a major feature of expert system architecture. Bramer [82] presents an overview of expert systems and related issues. Buchanan [82] has developed a good partial bibliography of work in this area.

Knowledge acquisition is the bottleneck in expert systems development, so several research efforts have been directed towards providing the domain expert with tools to aid him in refining and correcting a knowledge base. The Teiresias system [Davis 76] provided an interactive, English-language front end to the MYCIN rule base. Teiresias contained meta-level knowledge about diagnostic and therapuetic rules in the form of rule models. These models were used to generate expectations about the form and content of rules. Teiresias' expectations helped it to guide the debugging process. The ONCONCIN system [Shortliffe et. al. 77] was equipped with a tool that aided the expert in identifying problem areas in the knowledge base [Suwa, Scott and Shortliffe 82]. Both this tool and Teiresias worked in the context of a single expert system. They also assumed that the domain expert had already developed at least a partial knowledge base.

The standard method of knowledge base development is a generate-and-test process [Feigenbaum 77]. The domain expert and the knowledge engineer construct a knowledge base and control scheme. The domain expert then tests the system on examples. When problems are revealed during testing, the

expert modifies the knowledge base. The system is tested again. This cycle is repeated until the domain expert is satisfied with the system's performance.

Several research projects have resulted in domain independent expert systems [van Melle 80, Forgy and McDermott 77, Hayes-Roth et. al. 81]. All of these systems provide a single knowledge representation and inference mechanism. They present the domain expert with an "empty" expert system; this simplifies the knowledge engineer's job, since he no longer needs to go through the system development and selection of a knowledge representation. Though these systems speed the expert system development process, they do so by providing a method that follows the standard generate and test procedure. The domain expert is given a language in which to express his knowledge, but he is not given tools which will aid him in doing so.

Promising results in the area of rule acquisition have been obtained through the use of computer programs which induce rules from examples of expert decisions. In some cases, rules formed by such programs have outperformed rules written by human experts [Michalski and Chilausky 80, Quinlan 83]. However, this method has limitations. Inductively derived rules are sometimes too complex to be used or understood by humans. Since one of the principal features of expert systems is the ability to explain the reasoning behind decisions, such complex rules are not appropriate. The comprehensibility of induced rules may be increased by breaking the problem into subproblems which can be solved individually [Shapiro and Niblett 82, Reinke 82]. Unfortunately, the problem breakdown must be done by the human domain expert. Some research has been done on automating this process by applying a "concept formation program" [Michalski and Stepp 83a] for dividing examples into a hierarchy of subclasses [Paterson 83].

Another weakness with the inductive inference tools developed so far is the lack of any kind of knowledge to guide the search for appropriate generalizations. Quinlan's ID3 algorithm, for example, uses an information-theoretic measure to select the next attribute in its decision tree [Quinlan 79]. To compound the problem, the decision tree format used by ID3 is difficult for humans to understand,

although it is easily executable by machine. The GEM program (see chapter four) presents rules in an if-then format that is easier to understand. Also, the methodology used by GEM permits the user to specify *background knowledge* about the domain for which rules are being formed. However, the language GEM uses is somewhat limited (see Chapter 2).

Despite some weaknesses, inductive inference is a promising tool. A special advantage to this method of knowledge acquisition is that the expert is often better at generating examples than he is at generating an explicit declaration of his knowledge [Michie 82]. Since human apprentices are almost always taught by example, it would seem worthwhile to provide a knowledge base builder with inductive inference tools.

Programs that learn from examples will aid the domain expert in rule base acquisition. The problem of rule base debugging remains. The expert must try to find and deal with cases where his knowledge base might produce an ambiguous result. For example, in a medical expert system such as ONCONCIN it is essential that conflicting therapy recommendations not be given. Similarly, expert systems in, say, fault diagnosis, should be able to deal with every fault that may arise. In domains where rule base consistency and completeness are valid concerns, the weight of the problem is again left on the domain expert's shoulders.

1.2. Knowledge Acquisition and the ADVISE system

ADVISE is a set of software tools under development at the Artificial Intelligence Laboratory at the University of Illinois [Michalski and Baskin 83]. These tools, taken together, form a "meta-expert system" – a system for building expert systems. ADVISE is unique in that it does not use a single knowledge representation and inference mechanism. EMYCIN [van Melle 80], for example, allows only rule based knowledge and provides only a backward chaining control scheme. ADVISE uses a single low-level format for knowledge representation. This format is powerful enough to represent rules, networks and relational tables. ADVISE also allows a host of different inference mechanisms modules. Eventually the system will provide an inference mechanism language that will allow the user to define

his control scheme in terms of already existing tools.

Figure 1 shows a conceptual block diagram of the ADVISE system. The Control Block provides the user with an integrated access to the system's knowledge acquisition and inference tools. The Control Block accesses the Knowledge Acquisition Block or the Query Block, depending on the mode the user has selected. The Knowledge Acquisition Block provides direct access to the knowledge base through a relational database system, a rule editor and a network editor. Machine learning tools are attached to this block and provide a means for learning rules from examples (i.e. the inductive inference

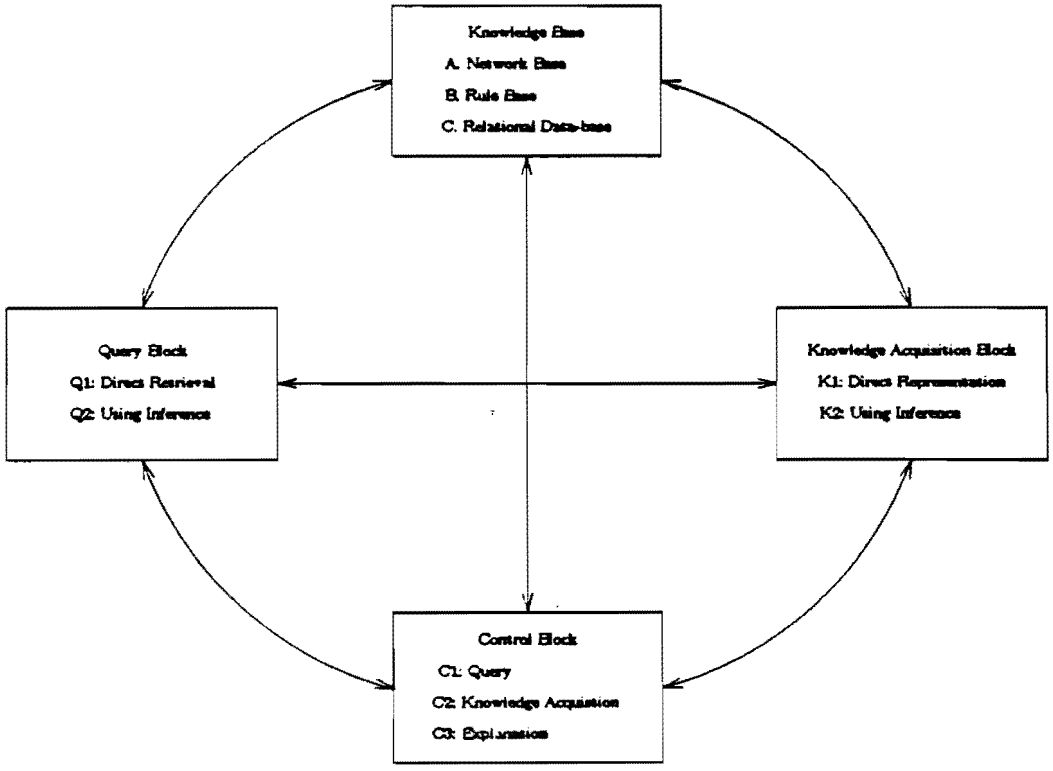


Figure 1. A conceptual level block diagram of the ADVISE system.

programs modify and update the knowledge base rather than the user doing so using more primitive tools). The Query Block provides deductive inference mechanisms (control schemes) for expert systems consultations and a relational database system for direct retrieval of knowledge stored in relational form.

Several expert systems have been implemented under ADVISE [Boulanger 83, Reinke 83, Rodewald 84]. These systems were used to drive the initial development of the meta-expert system tools.

Currently, the low-level knowledge representation language in ADVISE has been completed. The Knowledge Acquisition Block is partially completed [Spackman 83], as is the Query Block. The research described in this thesis is aimed at upgrading the capabilities of the knowledge acquisition block.

2. A PARADIGM FOR RULE BASE DEVELOPMENT

As stated in Chapter 1, the standard method for forming an expert system's knowledge base is a generate-and-test process. There are difficulties, however, in both the generation and testing of knowledge bases. The source of these difficulties is twofold. First, the expert is trained to *make* decisions, not to explicitly state his knowledge. Second, the expert is provided with virtually no aids in either stage of the process. He must generate his knowledge base from scratch with only the knowledge engineer's guidance to help him. He must then produce test examples which show faults in the knowledge base he himself just constructed.

Some relief is provided by expert sytem development systems, which establish a framework for expressing knowledge. Such systems give the expert a pre-defined knowledge representation method, and therefore make the knowledge acquisition process somewhat easier. However, they may also force the expert to channel his knowledge into a format which does not fit it. The knowledge representation problem will not be dealt with here. Instead, we will carefully delineate an area of applicability, and describe new tools for knowledge acquisition within that area.

2.1. Assumptions and Domain of Applicability

Many different knowledge representation formalisms, each applicable to a range of domains, have been developed in the last twenty years. Unfortunately, some of these formalisms have been used in areas for which they are not really acceptable. In order to avoid this trap, the knowledge representation to be used will be exactly defined. Such a presentation will naturally suggest certain problem types.

The methodology described in this thesis deals only with rules. In a variety of application areas, an expert's knowledge can best be expressed in the form of if-then rules. With some extensions to the if-then format, a rule formalism can deal with uncertainty in information, with weighted conditions and with multiple decisions and associated confidences. A detailed discussion of the syntax and semantics for rules is presented in Chapter 3.

Given that we are dealing with rule based knowledge, we need to define restrictions on the rule language. The major assumption made here is that we are dealing with domains where the expert system's job is to select an appropriate decision from a pre-specified list of possible decisions. As mentioned above, we will allow the system to associate a strength, or certainty value, with any decision it makes. The important thing is that the number of decisions that could be made is finite.

We will also place restrictions on the attributes used to describe problems. Rules will be written in terms of discrete, finite variables. We will assume that the domain expert is able to specify the important variables in his area of expertise, and that these variables are known and available. This does not assume, however, that all the attributes given by the domain expert are applicable, or that all information will be available during consultations conducted by the expert system developed. We are simply restricting the language which the rules use.

A final set of restrictions apply to the type of data available to inductive inference programs (see the next two sections, and Chapter 4). The learning and testing tools described here all act on the assumption that examples will be presented to them in terms of defined attributes, as above. That is, examples of expert decisions will be given in terms of discrete, finite attributes, and each example (event) will be associated with a single decision. We also assume that events will be given in their entirety when presented to a learning program, i.e. events will not be given in piecemeal form.

In summary, we are restricting ourselves to rule based knowledge. Our rules will be written in terms of discrete, finite attribute values. If a rule specifies a decision, that decision will be one of a known set of decision classes. If the expert is to present examples of his decisions, the examples will be presented in terms of the same attributes, and each example will have a defined decision associated with it. We hope to develop a method which will provide, within this restricted framework, useful tools for building and debugging rule bases. Discussions throughout the remainder of this thesis will assume we are dealing with knowledge of the type described in this section.

2.2. The Standard Rule Acquisition Paradigm

Figure 2 shows a flow chart of the standard knowledge engineering process. In the figure, circles represent processes and blocks represent objects (both humans and computer programs) which participate in the processes. The rule base specification process shown consists of two major subparts. First, the knowledge engineer must obtain from the domain expert a list of the variables that are relevant to the problem area. In medical diagnosis systems, for example, this would be a list of relevant symptoms, patient data and laboratory data. Once the attributes are defined, the expert may write the rules for the initial knowledge base. At this stage, in consultation with the domain expert, the knowledge engineer must decide exactly what needs to be represented and how to represent it in the form of rules. He must consider, for example, how to deal with uncertainty, with weights on conditions, and with how the rules should be evaluated.

Once this process is completed, the knowledge engineer must proceed on his own to encode the rules and the inference mechanism which will use them. Due to the complexity of the next stages of knowledge acquisition, the engineer must be certain that his system is easy to modify and that he has provided sufficient explanatory facilities so that the expert, when debugging the knowledge base, can find the causes of problems.

This leads to the third, and most difficult, stage of the expert system development process. During the rule base refinement stage, the domain expert must test his "pupil" on pre-classified examples. This process often involves several domain experts using the systems over a period of months. Once enough difficulties have been noted, the domain expert must go back to the rule base and make additions and changes to it and possibly to the list of relevant attributes.

2.3. A New Paradigm for Rule Acquisition

The problem with the standard paradigm is that the process relies very heavily on the time and effort of the very expert whose job should be eased by the system. The entire process also depends on

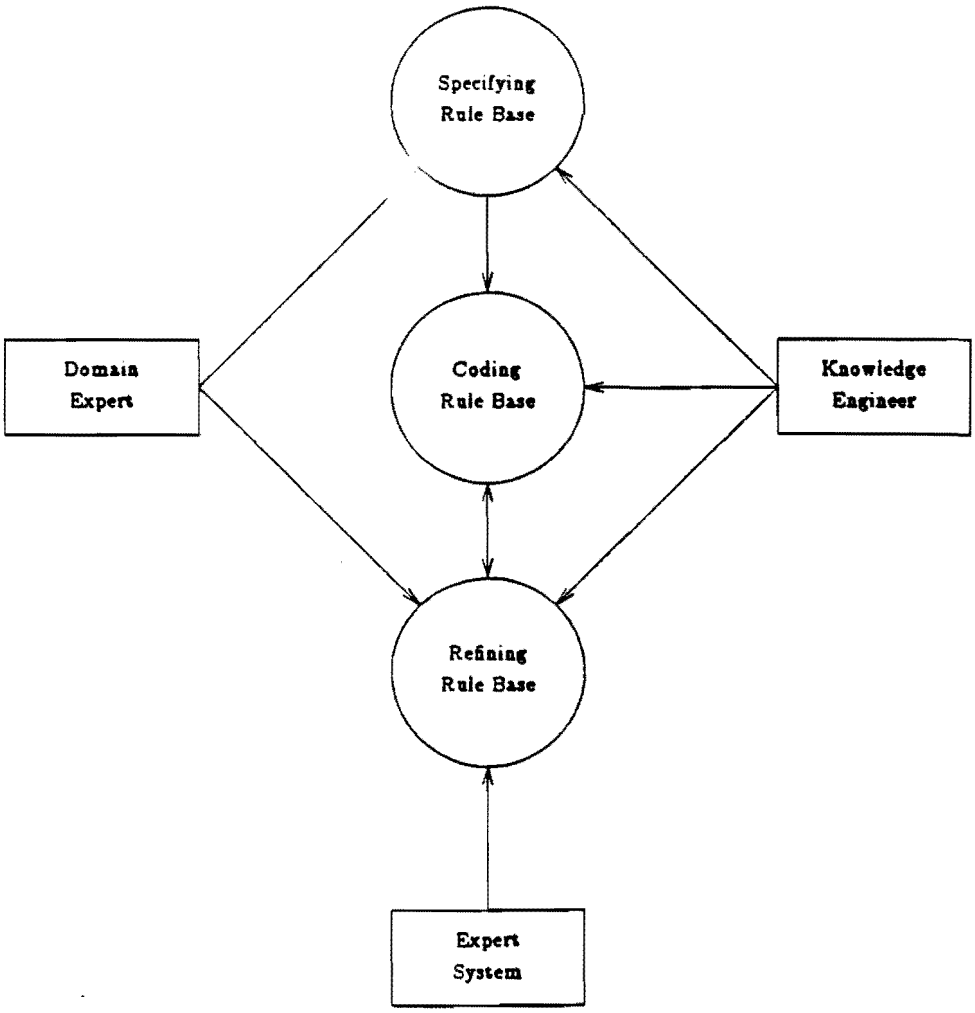


Figure 2. The standard paradigm for rule base development.

the domain expert's ability to elucidate and explain his knowledge. All of this suggests that the expert needs help in building and refining a rule base. Figure 3 shows a new paradigm for knowledge base construction, aimed at giving the expert help in those areas in which he is weak. The tools aligned with the

expert are intended to work with *examples* of expert decisions, as well as with explicit declarations of an expert's knowledge. These tools should also aid the expert in producing examples that will be of importance.

Under the new methodology, the development of a rule base begins with the expert specifying the attributes relevant to the problem. Some work has been done in aiding the expert here through a program that picks important attributes out of an exhaustive list [Baim 82]. At this point, the expert has two options. He may proceed in the standard way, aided only by a rule editor, or he may choose to

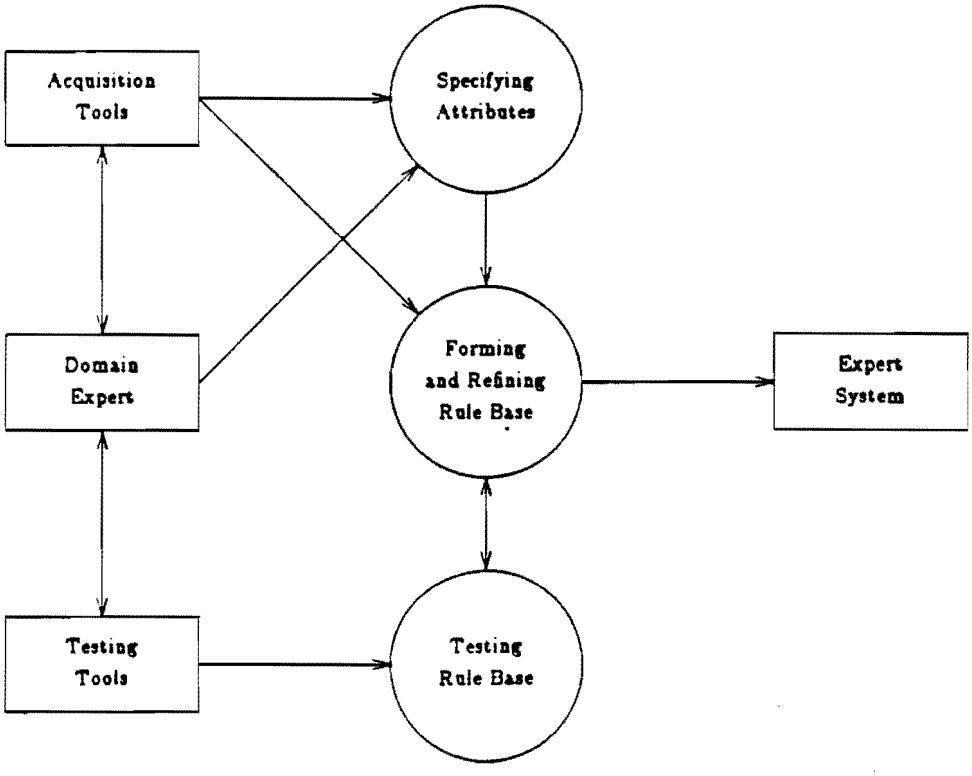


Figure 3. Paradigm for rule base development using automated refinement and testing

present the induction tools with a set of tutorial examples. The tools will produce a rule base which is guaranteed to work correctly for the examples given. In either case, the initial knowledge base is constructed, and the expert enters the knowledge refinement stage.

Here, the expert needs to produce examples that will demonstrate problems in the rule base. The testing tools shown in Figure 3 really consist of two parts: a mechanism to suggest areas where the rule base may not work correctly (i.e. it should suggest testing examples) and a mechanism that rapidly tests examples provided on the knowledge base and presents the results in a usable format to the domain expert.

If problems have been revealed in the knowledge base, it must be refined to deal with those cases which it handled incorrectly. Again, the expert is given the option of doing the work himself. However, he may present the examples which caused problems to the induction tool, which will refine the knowledge base so that it deals with these new examples correctly.

Note here that the new paradigm completely subsumes the old one. Within the context of the new method, the expert may still, if he chooses, do all the work himself, aided by the testing and editing tools. The most desirable course is probably a hybrid, wherein the expert may define some knowledge which is used to guide the induction process.

Given this paradigm, we can create a description of the software tools that should be available to the expert system builder. First, we need an efficient, correct method for generating and refining a rule base using examples. Next, we need tools that will help the expert generate testing examples and run those testing examples on the knowledge base. Additional tools to aid the expert in attribute definition would also be desirable. All these tools should work in the context of a powerful rule language which will be of use in a wide variety of domains.

The next chapter presents a rule language that is a subset of the multi-valued logic language supported by the ADVISE system. Chapter 4 describes programs which partially fill the induction and testing tool slots in the new paradigm.

3. RULE BASE SYNTAX AND SEMANTICS

The ADVISE knowledge representation language can be used to express rules. This language (essentially a tuple or list formalism – see [Boulanger 83]) is powerful, but too detailed to be presented to a domain expert. ADVISE does provide, in the context of a rule-based system, a method for dealing with uncertainty in evidence and in rules, so these problems need not be considerations in our rule language. What is needed is a specification of exactly what constitutes a rule base (i.e. how it is structured) and the syntax and semantics for the rules themselves.

3.1. Rule Base Syntax

A typical rule base has no structure. It consists of a group of rules of the form

condition implies action.

Here, **condition** is a logical statement which evaluates to some numeric value expressing the degree of truth of the left hand side of the rule. The **action** associated with a rule is the assignment of a value to some variable. The rule language described here will follow this general pattern, with one important exception – the knowledge base may be structured. That is, the action of a rule may be the selection of another set of rules. Such a scheme allows a modicum of control information to be incorporated into the rule base. There are two motivations for this. The first is the incomprehensibility of large, induced rule bases. As discussed in Chapter 1, this problem may be relieved somewhat by imposing a structure on the rule base. The second motivation is the fact that such a structure is natural in some domains. The standard taxonomic key, for example, could be easily represented with a structured knowledge base. Figure 4 shows a simple example of such a rule base.

An EBF grammar for rules is shown in Figure 5. Under this syntax, rules consist of a collection of (optionally weighted) modules. Each module contributes a numeric value between zero and one to the truth weight of the left hand side of the rule. A module consists of a disjunctive normal form expression. Each conjunct (complex) in the module is a list of primitive conditionals called "selectors"; where a selector is a statement of the form *attribute relation value*. The terminology for this rule base is derived from

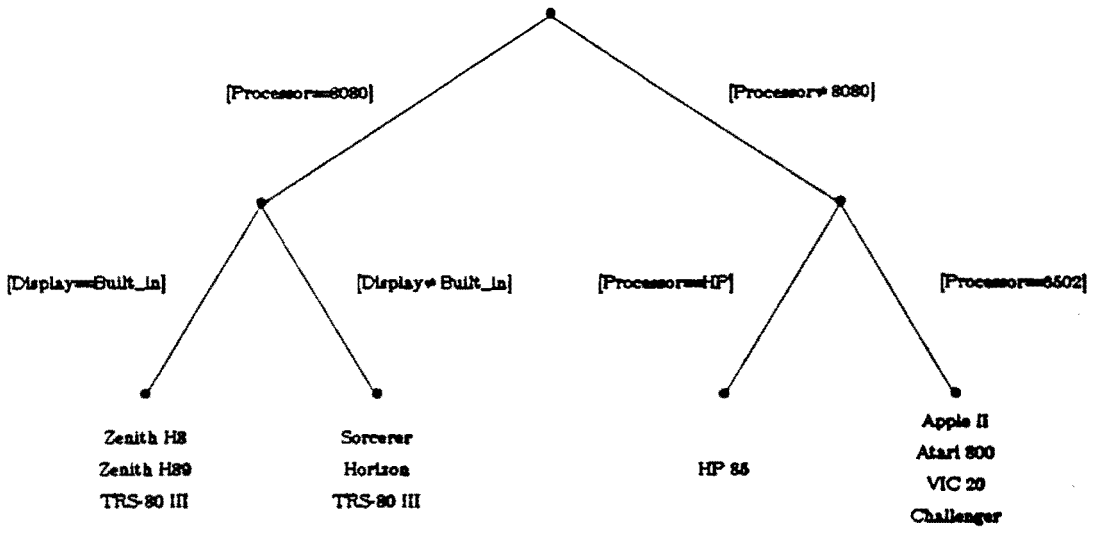


Figure 4. A simple structured rule base for classifying microcomputers (from [Michalski and Stepp 83b]).

the variable-valued logic language VL_1 [Michalski 73].

3.2. Rule Base Semantics

To define the semantics for a rule base of this type, we need to specify how the various operators in it (conjunction, disjunction, and logical operators in selectors) are to be evaluated. However, it seems likely that in different domains, different definitions of each operator might work best (see Chapter 5). Accordingly, we define no restrictive semantics on the modules, but rather provide a set of software switches which allow the user to select from a number of different options for each operator. The ADVISE rule evaluator has such a set of switches [Michalski et. al. 84], as do the ATEST evaluation routines (Chapter 4).

Rulegroup	::=	Rule^{+}
Rule	::=	Module:α_1:α_2 [+ Module:α_1:α_2]* ::> Decision
Module	::=	Sel Op Module | Sel Op Sel | Sel
Sel	::=	"[" variable "=" Values "]"
Op	::=	Λ | V
Values	::=	value[,value]*
Decision	::=	Sel | Rulegroup

Figure 5. An EBF grammar for rule bases.

The modules and the α-weights have a slightly more restrictive definition. The α_1 weight is intended to capture the notion of *independent evidence*, i.e. this number should express how important the corresponding module is in reaching the decision on the right hand side if the values of the other modules are unknown. The α_2 weight is intended to express the notion of *cumulative evidence*. This number should express the strength of the decision if this module and all previous modules are known to be true. If any module is known to be false, then the α_2 weight is not used.

Given these definitions of the weights, the semantics of "addition" of modules may be defined. Obviously, the α_2 weights contain their own semantics – there is no need to fold the weights together. However, a threshold must be defined at which the α_2 weights are no longer useful. Research remains to define multiple evaluation schemes for the sum of different α_1 weights. Currently, two such weights are folded together by taking their probabilistic sum.[1]

[1] The probabilistic sum of two numbers a and b is $a + b - (a \times b)$.

Figure 6 shows two rules from an imaginary rule base for identifying animals in a zoo. The first rule says that if an animal is striped and if its height at the should is between four and seven feet, then we can be fifty percent certain that the animal is a zebra. If we know that the animal has hooves and no antlers, then we have only ten percent confidence that the animal is a zebra. However, if both conditions are known to be true, then we are one hundred percent certain. The second rule is interpreted in a similar way, except that its action is not the identification of an animal, but the selection of another set of rules. If all the conditions in this rule are met, then the rule group "tiger_rules" is called, presumably to identify the particular type of tiger.

The rule schema described here has considerably more power than the available induction routines can derive. Some of the features (e.g. the α weights) are there to provide extra expressive power for the domain expert who is writing his own rules.

In an ADVISE rule base, several other tools are available. Each node in the rule base structure is called a rule group. A rule group may have contextual information associated with it, so rule groups may provide another level of "knowledge chunking". For example, different rule groups may have different rule evaluation settings. These settings are stored in the rule group structure and accessed by the rule evaluator. Such chunking provides conceptual simplicity for normal humans and a modular

---

```
[fur_pattern = striped][shoulder_height = 4..7] : 0.50 : 0.50
                        +
    [hooves = present][antlers = absent] : 0.10 : 1.00
                    ::> [animal = zebra]

[fur_pattern = striped][shoulder_height = 3..5] : 0.50 : 0.50
                        +
              [carnivorous = true] : 0.05 : 1.00
                ::> [rule_group = tiger_rules]
```

Figure 6. Two rules illustrating the use of α - weights.

---

construction for programmers. Eventually, rule group structures may be used to contain many bits of information about the properties of rules within the group. For example, Teiresias-like rule models could be stored with each rule group. Future versions of the ATEST program will leave, within the rule group structure, "footprints" of its evaluation. These footprints can be used to guide the next round of induction or rule group refinement.

4. TOOLS FOR INFERENCE AND TESTING

Under the paradigm developed in Chapter Two, a domain expert building a rule-based system is provided with tools that aid in the construction and refinement of the rule base. This chapter presents a detailed description of programs developed to fufill these roles. The programs described are somewhat limited in their scope, but it is hoped that they are the first step on the path towards an automated knowledge extraction system. The first section of this chapter sketches the AQ algorithm, as it is the heart of the GEM program. Section 4.1.2 describes the modifications to AQ that are necessary to make it work incrementally. Section 4.2 describes the ATEST program and presents the algorithms it uses to check consistency and completeness in the knowledge base.

4.1. The GEM Tool for Rule Base Refinement

4.1.1. The AQ Algorithm

The AQ algorithm is a method for producing minimal or quasi-minimal descriptions of classes of events. Events are given as vectors of values of discrete attributes with finite domains. The type of problem dealt with by the AQ method can be illustrated using generalized logic diagrams. These decision diagrams [Michalski 78] are planar representations of the multi-dimensional problem spaces used to represent events. The common Karnaugh map is a variation of the logic diagram applicable only to two-valued logics.

Figure 7 shows a decision diagram. Each cell in the diagram represents a single vector in the attribute space of the problem. Each such vector is called an *event*. Letters in event cells represent the assignment of a class to that event. So, the cell labelled a_1 is the first event of class A. The decision diagram is therefore a partial function mapping attribute vectors to classes. The function $f(x_1,x_2,x_3,x_4)$ shown in Figure 7 is defined as f(1,0,0,1)=A, f(0,0,2,1)=B, and so on. Each circled area (labelled with upper case letters) is a disjunct in the *cover* for a class. A cover is a description of the class based on the observed events of the class. A correct description is one that is satisfied by every event in the class and

none of the events of other classes. Obviously, there may be several covers for any set of events. The goal of the AQ algorithm is to produce the best possible cover within the constraints of its search space.

AQ works by selecting a single event, called the *seed*, in the class for which it is producing a cover. The seed is generalized as much as possible without covering any negative events (i.e. events of other classes). The generalization of a seed e^+ against a list of negative events $E^- = \{e^-_1, e^-_2, ..., e^-_n\}$ is called the *star* of e^+ against E^-, and written $G(e^+ \mid E^-)$. A star is formed by producing increasingly specialized covers of the seed event through logical intersections. Initially, the star is equivalent to the entire event space. The partially completed star is specialized by intersecting it with extensions of the seed against individual negative events. The extension against operator (" —| ") is illustrated in Figure 8. Generalization by extension is done by taking the negation of a negative event and intersecting it with the value of the seed along each attribute. In our example, the seed a_1 is initially extended against b_1. This result is intersected with the star so far (in this case, the entire event space), producing the partial star $G(a_1 \mid$

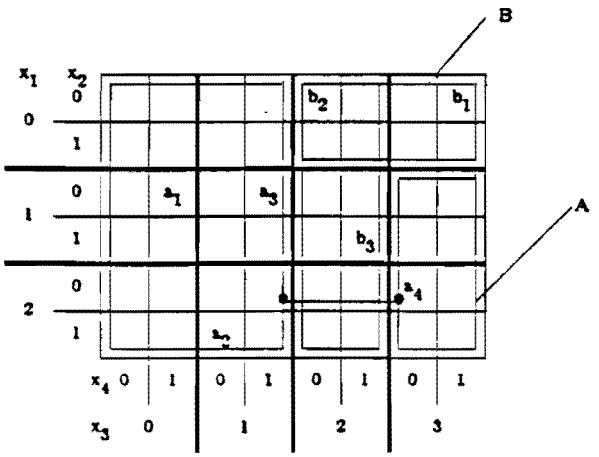
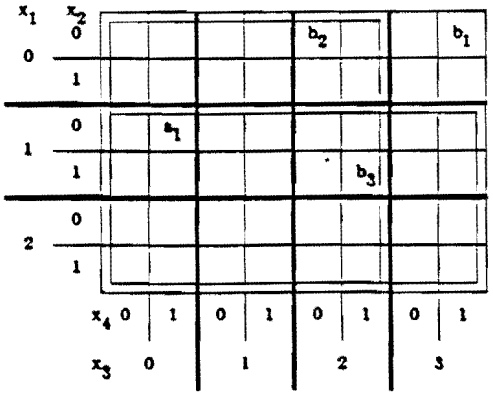


Figure 7. An example of a decision diagram.

$\{b_1\}$) shown in Figure 8. Since this partial star still covers the negative events b_2 and b_3, a_1 is extended against b_2 and the result intersected with the result shown in Figure 8. Figure 9 shows the partial star $G(a_1 \mid \{b_1,b_2\})$ formed after extension against b_2. The process is repeated for b_3, producing the star $G(a_1 \mid \{b_1,b_2,b_3\})$, which is shown as the cover for class A in Figure 7.

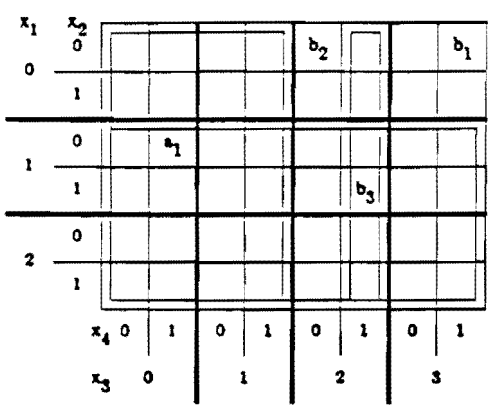
Naturally, if the star generation process were to work exhaustively, the search for covers could be exponential in the number of negative events. To deal with this, a parameter (*mazstar*) controls how many disjuncts may be kept in the partial star. If an intersection produces a partial star that is too large, the star is trimmed by applying a "lexicographic functional" (LEF) which selects the best conjuncts in the partial star and disposes of the remainder.

The idea of the LEF is essential here. The LEF specifies criteria for guiding the search through the space of possible covers. A LEF consists of a list of pairs of the form (criterion,tolerance). Typical cri-



$$(a_1 \text{ —| } b_1) = [x_3 = 0..2] \text{ V } [x_1 = 1..2]$$

Figure 8. An example of generalization by extension against – a_1 extended against b_1.



$$G(a_1 \mid \{b_1, b_2\}) = [x_1 = 1..2] \vee [x_3 = 0..1] \vee [x_3 = 2][x_4 = 1]$$

Figure 9. The partial star formed after extending a_1 against b_1 and b_2.

teria are "maximize the number of positive events covered", or "minimize the size of the complexes". The first criterion is applied to a list of conjunctive expressions (complexes), producing a cost for each. The first tolerance is used to establish equivalence classes among the complexes based on these costs. Ties (i.e. complexes in the same equivalence class) are broken by applying the next (criterion,tolerance) pair in the LEF. Criterion are applied until the desired number of complexes have been selected.

When star generation is completed, we are left with a list of complexes, each of which covers the seed event and none of the negative events. The lexicographic functional is applied to select the single best complex. This is added to the cover for this class (the cover starts out as empty). If there are any positive events remaining which are not covered by the description formed so far, a new seed is selected and the process iterates.

The AQ method can be used to produce *disjoint* or *intersecting* covers. Intersecting covers logically intersect over "don't care" areas of the event space. Disjoint covers, obviously, do not intersect. If

the list of negative events used during star generations includes events of classes for which covers have already been formed, intersecting covers will result. If, however, the list contains the covers of those classes, then disjoint covers will be formed.

Applying AQ will produce, for each class, a cover that is satisfied by all the events of the class and none of the events of the other classes (see [Michalski 75] for a more detailed theoretical discussion of the AQ method and the covering problem). The re, however, two essential problems with using AQ as a methodology for rule acquisition in expert tems. The first is that the algorithm, though obviously powerful, does not use much knowledge about the domain in forming its rules. The LEF is one method for specifying information (costs can be associated with variables, causing the LEF to select complexes containing "cheap" variables). The lack of guidance in rule formation often produces rules that, while correct, are somewhat misleading. Typically, inductively derived rules contain conditions that seem irrelevant to the domain expert. One solution to this problem is to allow the domain expert to specify background knowledge about the domain. What is needed here, in other words, is a combination of *learning by being told* and *learnin from examples*. In the next chapter we present a different approach. We can cause an induced rule to e press more information about a class by making the rule as long as possible. In other words, we seek to produce a description of a class that characterizes that class, rather than discriminating it from other classes. This gives the human reader more details about what the system has learned. Some interesting results in this area are presented in Chapter 5.

The second major problem with the AQ method is that it learns everything at once. AQ takes as input a set of examples and produces as output a set of rules, but it cannot modify those rules. It would be useful if AQ could use rules it had learned previously to derive new rules. It turns out that some fairly mple modifications to the algorithm will allow it to learn in an incremental fashion. These mo ons are discussed in the next section.

4.1.2. An Incremental Version of the AQ Algorithm

There are several different ways to learn incrementally. One method is learning with imperfect memory. In the case of inductive inference, this would mean forming rules from examples then throwing away some or all of the examples. When new examples which contradict the rules are observed, the rules would be modified using the old rules and only those examples which were retained from the previous steps. This seems to be how people learn, as evidenced by the fact that they sometimes forget examples they have already seen. There are two related problems with learning using imperfect memory. The first, as mentioned, is that it can lead to errors. The second is that it is necessary, in order to minimize errors, to find a method of selecting important events for retention. An earlier implementation of AQ [Michalski and Larson 78] was designed to perform incremental learning with imperfect memory. However, this method did not select important events for retention, but simply used initial hypotheses to aid its search for covers. This sometimes resulted in rules which did not cover events that had been observed at earlier stages of the learning process. In order to avoid these problems, the GEM program performs a different type of incremental learning – learning with perfect memory. In this case, the system remembers every example it has seen, as well as the rules it formed, and so can be guaranteed to produce new rules that are completely correct. The essential problem with this method is storage. Also, there is the danger that such a method will not provide any real speed-up over just scrapping the initial rules and starting over. Some experiments to confirm that the algorithm developed here avoids this problem are presented in Chapter 5.

As mentioned above, AQ can be made to learn incrementally in a fairly simple manner. The essential modification (originally suggested by Jeff Becker) involves changing the star generation procedure. Recall that during star generation, the result of extending a seed against a negative event is intersected with the partial star. If the extension was the first for this seed, however, the result *becomes* the partial star. In other words, the initial extension is intersected with the entire event space. If the initial intersection is done with some subset of the event space, the resulting star is guaranteed to fall within that

subset.

We can use this modification in producing specializations of a cover. Suppose that we start out with the situation shown in Figure 10, and that we are attempting to form a new cover for class "A". The problem is that the old cover is covering the newly classified events b_4 and b_5. The first step in modifying this cover, under the incremental algorithm, is to specialize it so that it covers no negative events. This is done by applying the star procedure to all positive events covered by the old rule (events a_1, a_2, a_3 and a_4 in our example) with the intial partial star equal to the old cover. The negative events are all events of other classes covered by the old cover (b_4 and b_5 in Figure 10). Figure 11 shows the result of specializing the cover for class A against the events b_4 and b_5.

Once the rule is specialized, we can apply the regular AQ algorithm, using the specialized covers (and any uncovered positive events) as seeds.

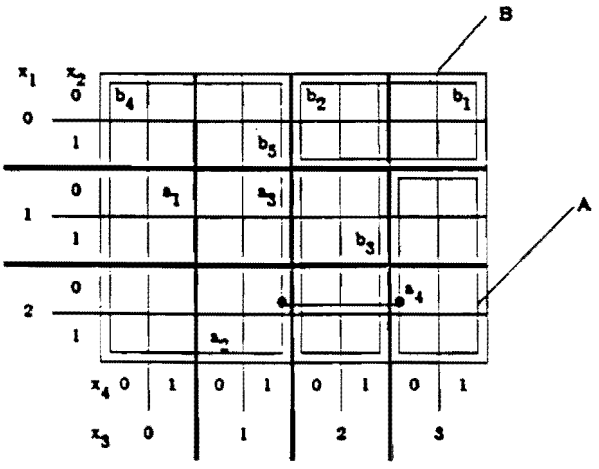


Figure 10. An incremental learning problem.

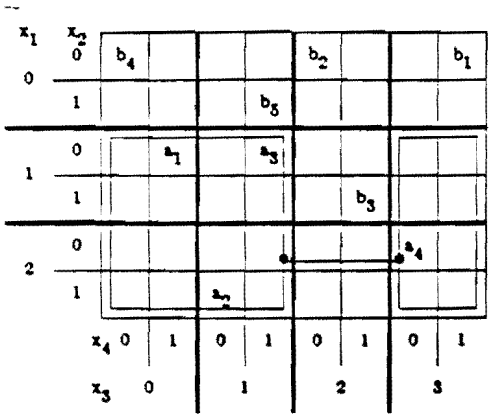


Figure 11. Specialized cover for class A generated by applying the modified star procedure.

Obviously, given perfect memory, this method will produce rules that take into account all the events seen so far. The major concern is the time consumed by the bookkeeping necessary for maintaining all the events ever observed. These concerns are addressed in the next chapter.

4.2. The ATEST Tool for Rule Base Debugging

For the new rule base acquisition paradigm to be effective, the domain expert must be able to produce testing examples for his knowledge base and apply those examples in order to assess rule base performance. ATEST is a tool developed specifically for that purpose. It provides the domain expert with two new capabilities. First, ATEST allows the expert to rapidly test a rule base on numerous examples under a variety of evaluation schemes. These evaluation facilities provide information about the overall performance of the rule base and about the performance of specific rules on specific examples. Second, ATEST provides routines that check a rule base for consistency and completeness. These routines can be used to point out problem areas in the rule base and to help the expert generate new examples.

Section 4.2.1 presents an introduction to the rule testing terminology used throughout the rest of this thesis. Section 4.2.2 presents the evaluation parameters available in ATEST and describes the program's evaluation and trace-abilities in detail. Section 4.2.3 presents a discussion of the consistency and completeness problems and describes the algorithms used by ATEST to test consistency and completeness in a rule base.

4.2.1. Terminology

ATEST views rules as expressions which, when applied to a vector of attribute values, will evaluate to a real number. This number is termed the *degree of consonance* between (the left hand side of) the rule and the event. The method for arriving at the degree of consonance, given a syntactically correct rule and an event, varies with the settings of the various ATEST parameters (see next section). When ATEST is run on a set of pre-classified testing examples, it simply applies each rule to each example and reports the degree of consonance. However, with a large number of testing examples, and a large number of rules, output of this sort is likely to get unwieldy. Therefore, ATEST has the ability to summarize the results.

Rule testing is summarized by lumping together the results of testing all the events of a single class. This is done by establishing equivalence classes among the rules that were tested on those events. Each equivalence class (called a *rank*) contains rules whose degrees of consonance were within a specified tolerance (called *tau*) of the highest degree of consonance for that rank. When ATEST summarizes the results, it reports, for each rule, the number of testing events for which that rule was a first rank decision.

The only remaining term to be defined is *satisfication*. Satisfaction applies to disjunctive normal form (DNF) expresssions. A DNF expression is a disjunction of conjunctive statements, i.e. a module as defined in Chapter 3. A DNF expression is said to be satisfied if some complex in it is satisfied. A complex is satisfied by an event if every selector in the complex is true for the event. In other words, satisfaction is a boolean logic conditional, and therefore applies to selectors and DNF expressions, but not to

modules or rule groups (which may have weights associated with their conditions).

4.2.2. The ATEST Evaluation Routines

ATEST takes as input a set of attribute definitions, a set of rules (and an optional structuring on the rules), a set of testing events, and a set of parameter values. The parameters control what ATEST does with the rules and how it evaluates the rules on the testing events. There are nine different parameters involved with rule testing. Six of these determine how rules are evaluated. The remaining three control which of ATEST's capabilities will be used during a given run. This section presents a discussion of the six evaluation parameters. The User's Guide in the appendix provides definitions for all the parameters.

Three evaluation parameters provide definitions for the logical operators in Figure 5. The operator "and" ("$\wedge$") may be evaluated as *minimum* or as *average*. The operator "or" ("$\vee$") may be evaluated as *maximum* or as *probabilistic sum*. The final evaluation parameter controls the the definition of the elementary conditions, called selectors. A selector may be treated as a boolean conditional (i.e. it may evaluate to 0 or 1), or as a function which when applied to an event evaluates to a normalized real number between 0 and 1. Given a selector in some attribute x whose domain is the ordered list $(a_1,a_2,...,a_n)$, and an event where $x = a_k$, the normalized value for the selector $[x = a_j]$ is

$$1 - (| a_j - a_k | / n).$$

If the selector has several values on its right hand side, the value closest to a_k is used.

The *tau* parameter mentioned in the previous section controls the assignment of rules to equivalence classes when testing on a single event. This parameter allows the user to determine what kind of range in degree of consonance he may expect when actually using the rule base for consultation. Increasing tau will increase the number of first rank decisions, and therefore increase the number of (possibly conflicting) actions associated with a given testing event. By varying the tau parameter, the expert can determine how robust his rules are in discriminatory terms.

The *dropa2* parameter (mentioned in Chapter 2) controls the use of the α_2 weight on rules. It specifies the truth threshold a module must exceed before that module can be included in the weight of cumulative evidence.

The remaining parameter, *threshold*, controls the degree of consonance threshold for a rule. ATEST reports, for every class, how many testing events caused the correct rule to have a degree of consonance greater than *threshold*. Figure 12 shows a sample problem input to ATEST and the resulting output if all of ATEST's evaluation capabilities (see Appendix) are being utilized. The output shown consists of two parts. The table is a confusion matrix showing the performance of the rules on class B events. The numbers in the matrix are the degrees of consonance; numbers surrounded by asterisks indicate correct first rank decisions. If ATEST is told to summarize the results, only the first and last rows of this table will be output. The second portion of the output is a trace of evaluation for those cases where the rule base did not perform correctly. The selectors surrounded with question marks are those which were not satisfied. Selectors in double brackets are those which were satisfied. In a structured rule base, this trace is considerably more complex, as it details the paths taken to reach the final degree of consonance.

4.2.3. Consistency and Completeness

In some domains, it is essential that no two rules in the rule base conflict, i.e. that the rule base is *consistent*. Inconsistency occurs if there is a situation (event) in which two rules would indicate different, mutually exclusive actions. In the terminology of Section 4.2.1, an inconsistency exists if there is an event which causes two rules of different class to evaluate to first rank decisions.

There are also cases in which it is necessary for some conclusion to be reached for every possible input. We say a rule base is *incomplete* if there is an event for which no rule has a degree of consonance greater than *threshold*. The threshold used in ATEST is defined by the user, but has a default value of 0.50.

Rule A : $|x_1 = 0,3|$

Rule B : $|x_1=2||x_3 = 1,2|$ V $|x_1 = 0||x_3 = 1|$

Testing Events for class B:

	x_1	x_2	x_3	x_4
B_1	1	1	1	0
B_2	0	0	1	1

Parameters:

Operator	Interpretation
AND	average
OR	maximum

ATEST OUTPUT :

TEST RESULTS FOR CLASS B

EVENT	#TIES	A	B
B-1		0.00	0.50
B-2		0.00	*1.00*
#1st rank events		0	1

Number of events satisfying rule for correct class : 1

The rule for class B was evaluated as follows for testing event B-1:

$??|x_1 = 2|??|[x_3 = 1,2||$ V $??|x_1 = 0|??|[x_3 = 1||$

Figure 12. Sample input and ATEST output for a toy problem.

Testing consistency and completeness in a rule base are relatively easy if we are dealing with unweighted, non-structured rules and applying a boolean logic scheme for rule evaluation. However, the

rule base we defined in Chapter 3 allows weighted, structured rules which may be evaluated in multiple ways. Therefore, ATEST does consistency and completeness checking under more general conditions. The routines in ATEST use a generate and test method for recognizing consistency and completeness problems. This methodology takes advantage of the speed and flexibility of the evaluation procedures already present for testing examples.

Consistency and completeness are handled in essentially the same manner. First, ATEST calls routines that apply logical and set theoretic operators to the rules to produce "test complexes". The test complexes are fed through the evaluation routines and the results are examined to determine if there is indeed a problem.

- If rules R_1 and R_2 are being tested for consistency:

$$R_1 : [x_1 = 3][x_2 = 4..6][x_3 = 4][x_5 = 7][x_8 = 9] : 0.8$$
$$+$$
$$[x_6 = 0..3] : 0.4$$
$$::> [d_1 = 0]$$

$$R_2 : [x_1 = 0][x_2 = 6..8][x_3 = 4][x_5 = 4][x_8 = 9] : 0.9$$
$$+$$
$$[x_6 = 4..6] : 0.05$$
$$::> [d_1 = 1]$$

- Then ATEST will generate the test complexes:

$$[x_1 = \text{FALSE}][x_2 = 6][x_3 = 4][x_5 = 7][x_7 = 4][x_8 = 9]$$
$$[x_6 = 4..6]$$
$$[x_6 = 0..3]$$

- These complexes, if "and" is evaluated as average, will cause ATEST to report:

The complex : $[x_2 = 6][x_3 = 4][x_5 = 7][x_7 = 4][x_8 = 9]$
produces a dc of 0.88 with rule R_1 and a dc of 0.91 for rule R_2.

Figure 13. An example of consistency testing.

The generating routines for consistency operate by forming the intersection of the left hand sides of the rules that are to be tested. A standard logical intersection will not work for two reasons. First, there are cases where such an intersection will be empty even though, under certain evaluation schemes, the rules will produce conflicting decisions. Second, the number of intersections to be performed grows exponentially with the number of complexes in the rules.

The first problem is dealt with by changing the definition of intersection. The consistency testing routine multiplies rules together in the standard fashion except that the existence of non-intersecting selectors in a conjunct does not reduce the intersection to the empty set. Instead, a special selector, which always evaluates to zero, is inserted. In this way, events that may satisfy two rules to a high degree of consonance may be generated.

The second problem is handled in two ways. First, the consistency checking routines accept a parameter (*dweight*) which specifies a minimum weight for modules. If a module has an α_1 weight below

---

- Given four boolean variables x_1, x_2, x_3, x_4 and the rules:

$$R_1 : [x_1 = f][x_2 = t][x_3 = t] : 0.80$$
$$+$$
$$[x_3 = f] : 0.60$$
$$::> [d_1 = 0]$$

$$R_2 : [x_1 = t][x_2 = t]$$
$$::> [d_1 = 1]$$

- The union of all complexes is subtracted from the entire event space yielding the test complexes:

$$[x_1 = f][x_2 = f][x_3 = t]$$

$$[x_1 = t][x_2 = f][x_3 = t]$$

- Causing ATEST to report that neither complex satisfies any rules.

Figure 14. An example of completeness testing.

dweight, the module is simply not used when forming the intersection of two rules. Second, the fact that the knowledge base is structured should tend to decrease the number of test complexes produced. Since consistency checking is only done between children of the same parent in the rule base structure, the number of rules that are involved in consistency checking is reduced. Figure 13 shows an example of how the consistency and completeness routines work.

Completeness checking is done by taking the union of the left hand sides of all rules that have the same parent in the rule base. Again, the dweight parameter is used to exclude modules whose weights may be too low. Once the union is formed, it is subtracted from that portion of the event space which should be covered. If the rules being tested are at the top of the knowledge base structure, then the union is subtracted from the entire event space. Otherwise, the union is subtracted from that portion of the event space covered by the parent node. Figure 14 shows an example of the steps involved in completeness testing.

This process again generates test complexes. These complexes are applied to every rule used in the union. If none of the rules have a degree of consonance greater than the defined threshold, then the test complex is reported as an area of the event space that the rules should cover but do not.

5. AN EVALUATION OF LEARNING MODES

The new paradigm for rule base refinement requires an efficient method for learning incrementally from examples. Chapter four presented a revised version of the AQ algorithm which, it is hoped, will satisfy this criterion. Also, it was mentioned that rapid testing of a knowledge base under different evaluation schemes, as allowed by ATEST, may aid a domain expert in selecting the best method by which to evaluate his rule base. This chapter presents experiments designed to test whether the incremental learning algorithm presented in Chapter 4 satisfies the criteria of Chapter 2, and whether the testing facilities provided by ATEST are worthwhile in knowledge base development.

There are two major concerns related to the incremental learning algorithm. The first is whether the method provides a worthwhile way to learn incrementally from examples. The second is that there may be several ways to form rules incrementally using this method. The LEF in GEM provides a means to vary the rule formation process; there may be certain criterion which work best in a given domain or a given domain type.

Under the rule base formation paradigm, rules are not put into the expert system until the domain expert is satisfied with their performance. ATEST provides a means to test the performance of a rule base under several different evaluation schemes. If, however, rules in different domains perform at the same level regardless of evaluation schemes, then the extra tools ATEST provides may not be necessary. On the other hand, it may be that rules formed by different means in the same domain should be evaluated differently. For example, "AND" may not mean the same thing to a domain expert as to the GEM program. In such a case, the function should be evaluated differently for different types of rules.

In order to assess the performance of both ATEST and GEM, the programs were applied to three separate problems of increasing complexity. Hopefully, the range in problems is sufficient to suggest the differences in performance that will occur in other real-world applications.

The first domain was the classification of different species of *Stenonema* mayfly nymphs [Lewis 74]. Seven species of Interpunctatum group nymphs were described in terms of 7 attributes -- the size of the

event space was on the order of 10^6. Five different examples of each species were used for learning, and another five were used for testing the induced rules. The description in Figure 15 is a typical event in this domain.

The second application area was the King-Pawn-King black-to-move chess endgame, where the pawn's side is white. Here, examples were described in terms of 31 boolean attributes [Shapiro and Niblett 82]. This domain differed somewhat from the other two in that each example actually covered several legal KPK positions. That is, the input examples are somewhat generalized representations of the board positions. The examples were correctly classified into *Won* for the pawn's side or *Drawn*. All legal combinations of attribute vectors (a total of 1901) were used. Half of the events were used for learning, with the remainder set aside for testing of the induced rules. A typical event for this domain is shown in Figure 16.

The largest application area was the soybean disease diagnosis domain [Michalski and Chilausky 80]. Diseased soybean plants were described in terms of 50 attributes. Attribute domains ranged in size from two to eleven values, meaning that approximately 10^{30} attribute vectors were possible. The event set consisted of examples of 17 different soybean diseases common in Illinois; there were 17 different examples of each disease. Figure 17 shows a typical example of one disease, alternaria leaf spot. This

---

1) maxilla_crown_spines = 10
2) maxilla_lateral_setae = 21
3) inner_canine_teeth = 2
4) outer_canine_teeth = 7
5) terga_mid_dorsal_pale_streaks = absent
6) terga_dark_posterior_margins = absent
7) dark_marks_sterna_9 = absent

Figure 15. A single event in the mayfly domain -- a nymph of the species *Stenonema carolina*.

1) cimmt = false	(Can the black king immediately capture the pawn?)
2) cplu2 = false	(Is the distance of the pawn from the queening square greater than the black king's effective distance plus two?)
3) cplu1 = false	(Is the distance of the pawn from the queening square greater than the black king's effective distance plust one?)
4) cahea = false	(Can the black king get ahead of the pawn on the pawn's file?)
5) cwksa = false	(Is the white king ahead of the pawn?)
6) ccnt = false	(Does the white king control the seventh rank square covering the queening square on the black king's side of the pawn?)
7) cpat1 = false	(Is the black king constrained to retreat?)
8) cpat2 = false	(Is the black king in stalemate or will advancing pawn force stalemate?)
9) rneac = false	(Can the black king reach c8 before the white king?)
10) rnea8 = false	(Can the black king reach a8 before the white king?)
11) rneap = false	(Is the black king nearer to the pawn than the white king?)
12) rpsq = false	(Can the black king move inside the pawn's square?)
13) rrp1 = false	(Special pattern; see [Shapiro and Niblett 82])
14) rrp2 = false	(Can the black king trap the white king near the edge of the board?)
15) rnear = false	(Special pattern; see [Shapiro and Niblett 82])
16) rstal = false	(Does this position lead to stalemate?)
17) mdiro = false	(Is the white king one rank ahead of the pawn and does he have the opposition?)
18) mmp1 = false	(Can white, by moving the king alone, get to mainpatt 1 rank ahead of the pawn?)
19) mmp2 = false	(Can white, by moving the king alone, get to mainpatt 2 ranks ahead of the pawn?)
20) mpmov = false	(Can white get to mainpatt by first moving the pawn?)
21) diro5 = false	(Does the 6th rank pattern hold or can it be achieved?)
22) btop5 = false	(Is the black king directly in front of the pawn or can he get there?)
23) mp5 = false	(Can white get to mainpatt by moving the king alone?)
24) r5p6 = false	(Special pattern; see [Shapiro and Niblett 82])
25) spra7 = false	(Is pawn on rank 7 and not on rook's file?)
26) spran = true	(Is pawn on rank 5 or 6 and not on rook's file?)
27) smain = false	(Is pawn on rank 1-5 and not on rook's file?)
28) srfil = false	(Is pawn on rook's file?)
29) sint = false	(Can black prevent pawn from running or mainpatt but not both?)
30) nxto7 = true	(Can white king force its way next to pawn?)
31) stlm7 = false	(Is the initial position a stalemate?)

Figure 16. A single event in the KPK domain of the *Won* class. Parenthesized expressions are definitions of the corresponding attribute.

data set was divided into a set of 170 examples (10 of each disease) to be used by GEM for learning and 119 (7 of each disease) to be used in testing the rules formed. The data used differed from that described in [Michalski and Chilausky 80]. For these experiments, fifteen more attributes were used and two new diseases were added to the data. In this domain, rules written by human experts were also available, so these too were tested.

5.1. Single Step Learning

This section presents experiments designed to answer the questions raised above about different rule formation methods and the utility of different evaluation schemes. These experiments have nothing to do with incremental learning methods. Instead, we wish to determine how different goals for rule

---

1) time_of_occurrence = october
2) precipitation = above_normal
3) temperature = above_normal
4) cropping_history = three_or_more
5) damaged_area = plants_in_upland_areas
6) severity = minor
7) plant_height = normal
8) condition_of_leaves = abnormal
9) leaf_spots = present
10) leaf_spot_color = brown
11) color_of_spot_on_reverse_side = none
12) yellow_leaf_spot_halos = absent
13) leaf_spot_margins = water_soaked
14) raised_leaf_spots = absent
15) leaf_spot_growth = scattered_with_concentric_rings
16) leaf_spot_size = greater_than_eighth_inch
17) shot_holing = present
18) shredding = absent
19) leaf_malformation = absent
20) premature_defoliation = present
21) leaf_mildew_growth = absent
22) leaf_discoloration = none
23) position_of_affected_leaves = scattered_on_plant
24) condition_of_leaves_below_affected_leaves = unaffected
25) leaf_withering_and_wilting = absent
26) condition_of_stem = normal
27) stem_lodging = does_not_apply
28) stem_cankers = does_not_apply
29) canker_lesion_color = does_not_apply
30) reddish_canker_margin = does_not_apply
31) fruiting_bodies_on_stem = does_not_apply
32) external_decay_of_stem = does_not_apply
33) mycelium_on_stem = does_not_apply
34) external_stem_discoloration = does_not_apply
35) location_of_stem_discoloration = does_not_apply
36) internal_discoloration_of_stem = does_not_apply
37) sclerotia_internal_or_external = does_not_apply
38) condition_of_fruit_pods = abnormal
39) fruit_pods = diseased
40) fruit_spots = colored_spots
41) condition_of_seed = normal
42) seed_mold_growth = does_not_apply
43) seed_discoloration = does_not_apply
44) seed_discoloration_color = does_not_apply
45) seed_size = does_not_apply
46) seed_shriveling = does_not_apply
47) condition_of_roots = normal
48) root_rot = does_not_apply
49) root_galls_or_cysts = does_not_apply
50) root_sclerotia = does_not_apply

Figure 17. A single event in the soybean disease diagnosis domain -- a plant with alternaria leaf spot.

---

formation, as expressed in the LEF, can affect the quality of induced rules and the type of evaluation scheme that is best for them.

The AQ algorithm produces a quasi-optimal description of a set of classes in terms of discrete finite attributes. Normally, AQ is applied to a problem with the express goal of producing a small, correct and discriminatory description of each class. However, rules produced in this manner sometimes do not seem sensible to the domain expert because the attributes chosen by GEM for use in the rules are sometimes not the attributes he chose. One obvious solution to this problem is to allow the domain expert to tell GEM which attributes are important and useful and allow this knowledge to guide the search. However, this places the weight of rule formation right back on the expert's shoulders. In order to specify the varying importance of each attribute to the induction program, the expert must essentially write rules, which is what we are trying to avoid.

Another possibility, which does not depend on the expert's ability to explain his methods, is to cause GEM to generate *characteristic descriptions* [Michalski 83]. A characteristic description does not define a class as an entity distinct from other classes, but presents a description of the class in as much detail as possible. In other words, a characteristic description is the type of explanation an expert might give if asked to describe a class in detail, apart from considerations of other classes. If an induction program produces good characteristic descriptions, the expert can view, in detail, how the learning process is proceeding. Causing GEM to generate a lengthy, complete description of the events in a class is not difficult. By inserting a criterion in the LEF that causes AQ to select the *longest* complexes in a partial star, the algorithm will produce a detailed description of each class. Ideally, such a description will be conjunctive.

If a discriminant description is necessary, there are two ways to build it. Naturally, GEM could be applied in the usual way. However, GEM can induce over rules in the same way as it induces over examples. It may be possible to produce good discriminant rules by inducing over characteristic descriptions (this idea was suggested by R.S. Michalski).

It is therefore possible to produce three kinds of rules from a set of examples -- characteristic descriptions, discriminant descriptions induced from the examples, and discriminant descriptions induced from characteristic descriptions. The performance of these different types of rules is compared to give an idea of the usefulness of each method. Also, rules produced in different ways may work best under different evaluation schemes. GEM was applied to each of the three domains above. In each domain, a set of examples was chosen at random to be the learning events. These events were used to produce the three different types of descriptions. Then, the induced rules were tested on the remaining examples under four different evaluation schemes. The evaluation schemes varied in their evaluation of the "AND" operator, and in whether or not selectors in linear variables were normalized. Performance of the rules on the testing examples was measured by two means -- what percentage of the events caused the correct rule to have a degree of consonance in the first rank (with tau = 0.02) and what percentage of the events caused the correct rule to have the only first rank degree of consonance. The experiments were repeated twice in each application area in order to observe how changing the learning events affected the performance of the rules. All rules were formed using the "disjoint cover" mode of the GEM program (see Chapter 4). It is possible that different results would be obtained if intersecting covers were created.

The results for the rules induced to identify the *Stenonema* mayfly nymphs are shown in Table 1. The induction time shown is the total CPU time used by the GEM program while forming rules. The complexity measure used here is a rather simple scheme that characterizes rules by size. The *complexity* of a single rule is defined as the *sum of the number of complexes (conjuncts) in the rule, the number of attributes used in the rule, and the number of selectors in the rule.* The complexity of a set of rules is the average complexity of the rules in the set.

All induction times shown in the table are for a Pascal implementation of GEM runnning under UNIX on a VAX 11/780. In this application area, the induction times for the characteristic and discriminant rules were about the same.

Rule Type	Induction Time	Complexity	"OR"	"AND"	Normalized	%1st Rank	%Only Choice
Characteristic	17.70 secs	8.28	maximum	minimum	no	34.29	34.29
Description			maximum	average	no	82.86	77.14
			maximum	minimum	yes	100.00	0.00
			maximum	average	yes	100.00	0.00
Discriminant	20.13 secs	8.00	maximum	minimum	no	34.29	34.29
From			maximum	average	no	74.29	71.43
Examples			maximum	minimum	yes	100.00	0.00
			maximum	average	yes	100.00	0.00
Discriminant	0.85 secs	7.00	maximum	minimum	no	65.71	65.71
From			maximum	average	no	91.43	85.71
Characteristic			maximum	minimum	yes	100.00	0.00
			maximum	average	yes	100.00	0.00

Table 1. Comparison of three different rule types for identification of *Stenonema* mayfly nymphs.

None of the rules produced were very complex. The relative complexity of different rule types remained the same over two runs. Figure 18 shows an example of the three rule types induced for one of the seven species of nymphs.

The interesting point here is the sensitivity of the rules to evaluation scheme. Preliminary results showed that the only useful evaluation scheme for "or" was "maximum." In the table, the "%1st Rank" column shows the percentage of all learning events for which the correct rule evaluated to a first rank decision (see section 4.2.1 for a definition of "rank"). The "%Only Choice" column shows the percentage of all events for which the correct rule evaluated to the *only* first rank decision. For mayfly identification, normalization of linear selectors produces disastrous results. Because most of the attributes are linear, counting "nearness" of a selector to an event causes virtually every rule to be satisfied by every example. On the other hand, averaging had a positive effect on all the rules. These two effects are related because one attribute is usually enough for identification and one completely satisified selec-

Characteristic description :

[maxilla_crown_spines = 10][maxilla_lateral_setae = 21,26,28,30][inner_canine_teeth = 2]
[outer_canine_teeth = 7..8][terga_dark_posterior_margins = absent]

Discriminant description induced from characteristic description :

[maxilla_crown_spines = 10][inner_canine_teeth = 2][terga_dark_posterior_margins = absent]

Discriminant description induced from examples :

[terga_mid_dorsal_pale_streaks = absent]

Figure 18. Three different rule types for identifying nymphs of the species *Stenonema carolina*.

tor may cause a rule to be satisified (causing normalization to fail). Similarly, having one selector not satisfied should not cause a rule to be rejected (hence the failure of minimum as an evaluation scheme in this case).

The results for the KPK endgame data are shown in Table 2. Since none of the attributes used in this domain were linear, the normalization parameter in ATEST will not have any effect. In this domain, attempting to induce discriminant rules from characteristic ones fails – GEM simply returns the rules given as input. Therefore, there are no entries in the table for such rules. Since there are two mutually exclusive classes, a decision that is first rank, but not the only first rank, is irrelevant; for this reason the "%1st Rank" column is also excluded from Table 2.

Again, the results shown are for one set of learning examples. A second run, using a different set of learning examples, produced results that were similar in terms of the relations between the different types of rules. However, the performance in absolute terms differed considerably – the rules induced during the second run were more than 95% accurate.

Rule Type	Induction Time	Complexity	"OR"	"AND"	%Only Choice
Characteristic Description	453.58 secs	60	maximum maximum	minimum average	40.32 40.32
Discriminant From Examples	436.95 secs	48.5	maximum maximum	minimum average	40.21 40.21

Table 2. Comparison of two different rule types for the KPK chess endgame.

In this domain, there was little difference between the characteristic and discriminant descriptions. This is probably due to the nature of the attributes used to describe events. Since each input vector is really a generalization of several actual chess positions, one event may not generalize easily to cover another. This hypothesis is partly borne out by the fact that the rules produced were very disjunctive, containing an average of twenty complexes each. This also explains GEM's failure to induce discriminant covers from the characteristic ones. There may not be many cases where a longer complex will serve as well as a shorter (more general) one. In this problem area, when such a long complex does exist, it will be disjoint from other complexes in the cover. It then becomes impossible to generalize several complexes together to form a discriminant description. Typical rules of each type, induced for the "won for white" class, are shown in Figures 19 and 20.

Another important point here is that choice of evaluation scheme made no difference in rule performance. This is also related to the nature of the problem space. Since the effect of treating conjunction as average is to generalize the complexes, averaging will only improve performance if a complex can be easily expanded to cover more events. Since each complex in this problem area seems to be highly specialized, generalization through averaging does not increase rule correctness.

[cimmt = f][cplu2 = f][cplu1 = f][cahea = f][cwksa = f][rrp2 = f][rstal = f] V
[cimmt = f][mmp2 = t][btop5 = f][spra7 = f][srfil = f] V
[cimmt = f][cplu2 = f][cplu1 = f][cahea = f][rneac = f][rrp2 = f][rstal = f] V
[cimmt = f][rrp2 = f][mp5 = t][spra7 = f][spran = t][srfil = f] V
[cimmt = f][cplu1 = f][rneac = f][rrp2 = f][rnear = f][rstal = f][mpmov = t][spra7 = f][srfil = t] V
[cimmt = f][cplu1 = f][cahea = f][rneac = f][rrp2 = f][rstal = f]
[mmp2 = t][srfil = t][nxto7 = t] V
[cimmt = f][cahea = f][rnear = f][btop5 = f][spra7 = t][srfil = f][nxto7 = t] V
[cimmt = f][ccrit = f][mmp1 = t][mp5 = t][spra7 = f][smain = t][srfil = f] V
[cimmt = f][rrp2 = f][mpmov = t][mp5 = f][spra7 = f][smain = t][srfil = f] V
[cimmt = f][ccrit = f][btop5 = f][spra7 = t][srfil = f][nxto7 = t] V
[cimmt = f][btop5 = f][spra7 = f][spran = t][srfil = f] V
[cimmt = f][ccrit = f][mdiro = t][btop5 = f][spra7 = f][srfil = f] V
[cimmt = f][ccrit = f][spra7 = f][smain = t][srfil = f][sint = t] V
[cimmt = f][ccrit = f][diro5 = t][spra7 = f][spran = t][srfil = f] V
[cimmt = f][cplu1 = f][cahea = f][cwksa = f][ccrit = t][rrp2 = f][rstal = f][mp5 = f][srfil = t][nxto7 = t] V
[cimmt = f][cplu1 = f][ccrit = f][rneac = f][rstal = f][mmp1 = t][mp5 = t][spra7 = f] V
[cimmt = f][cplu1 = f][rneac = f][rrp2 = f][rstal = f][mmp2 = t][mp5 = t][spra7 = f] V
[cimmt = f][cplu1 = f][ccrit = f][rneac = f][rrp1 = t][rstal = f][spra7 = f][srfil = t] V
[cimmt = f][cplu1 = f][cwksa = f][rrp2 = f][rstal = f][mpmov = t][mp5 = f][spra7 = f][srfil = t] V
[cimmt = f][rrp2 = f][r5p6 = t][spran = t][srfil = f][nxto7 = t] V
[cimmt = f][ccrit = f][mpmov = t][spra7 = f][smain = t][srfil = f] V
[cimmt = f][ccrit = t][rneac = f][rrp2 = f][rstal = f][mmp2 = t][mpmov = t][spra7 = f][srfil = t] V
[cimmt = f][cahea = f][ccrit = t][rrp2 = f][rnear = f][mp5 = t][srfil = f][nxto7 = t] V
[cimmt = f][rneac = f][rstal = f][mmp2 = t][btop5 = f][spra7 = f]

Figure 19. Discriminant description for the KPK class *Won*.

[cimmt = f][cplu2 = f][cplu1 = f][cahea = f][cwksa = f][rpsq = f][rrp2 = f] V
[cimmt = f][cplu1 = f][cahea = f][rneac = f][rrp2 = f][mmp2 = t][nxto7 = t] V
[cimmt = f][cplu1 = f][ccrit = f][rneac = f][mpmov = t] V
[cimmt = f][cplu2 = f][cplu1 = f][cahea = f][cwksa = f][rrp2 = f][mp5 = t][nxto7 = t] V
[cimmt = f][cplu2 = f][cplu1 = f][cahea = f][rneac = f][rrp2 = f][srfil = t] V
[cimmt = f][cplu1 = f][rneac = f][rrp2 = f][rnear = f][mmp1 = t][mp5 = t] V
[cimmt = f][cplu1 = f][cwksa = f][rneac = f][rrp1 = t][rrp2 = f] V
[cimmt = f][rnea8 = f][rrp2 = f][rnear = f][mpmov = t][smain = t] V
[cimmt = f][cplu2 = f][cplu1 = f][rrp2 = f][mpmov = t][smain = t] V
[cimmt = f][rneac = f][mmp2 = t][btop5 = f][nxto7 = t] V
[cimmt = f][cwksa = t][rneac = f][mmp1 = t][btop5 = f] V
[cimmt = f][cwksa = f][rrp2 = f][mp5 = t][spran = t] V
[cimmt = f][cplu1 = f][cahea = f][cwksa = f][rrp2 = f][rnear = f][mp5 = t][spra7 = t][nxto7 = t] V
[cimmt = f][cplu1 = f][cahea = f][cwksa = f][ccrit = f][mp5 = t][spra7 = t] V
[cimmt = f][cwksa = f][btop5 = f][spran = t] V
[cimmt = f][rrp2 = f][mmp2 = t][mp5 = t][smain = t] V
[cimmt = f][cplu1 = f][rrp2 = f][rnear = f][mdiro = t][mp5 = t][smain = t] V
[cimmt = f][cplu1 = f][ccrit = f][smain = t][sint = t] V
[cimmt = f][cwksa = f][rrp2 = f][diro5 = t][spran = t] V
[cimmt = f][cwksa = f][ccrit = t][rnea8 = f][rrp2 = f][rnear = f][mpmov = t][mp5 = f][srfil = t] V
[cimmt = f][cplu1 = f][cahea = f][cwksa = f][ccrit = t][rrp2 = f][rnear = f][mp5 = f][srfil = t] V
[cimmt = f][cplu1 = f][cwksa = t][rrp2 = f][mpmov = t][smain = t] V
[cimmt = f][cplu1 = f][cahea = t][rneac = f][rrp2 = f][mmp2 = t][mp5 = t] V
[cimmt = f][ccrit = f][mmp1 = t][mp5 = t][smain = t] V
[cimmt = f][cplu1 = t][rrp2 = f][r5p6 = t][spran = t] V
[cimmt = f][ccrit = f][rneac = f][mpmov = t][smain = t] V
[cimmt = f][cahea = f][cwksa = f][ccrit = t][rrp2 = f][rnear = f][mp5 = t][spra7 = t][nxto7 = t] V
[cimmt = f][cplu1 = t][rrp2 = f][mp5 = t][spran = t] V
[cimmt = f][cplu1 = t][ccrit = t][rneac = f][rrp2 = f][rnear = f][mpmov = t][srfil = t] V
[cimmt = f][cahea = t][ccrit = f][mdiro = t][btop5 = f][smain = t]

Figure 20. Characteristic description for the KPK class *Won*.

The final important factor in this domain is that the performance of rules of all types is highly dependent on the set of events chosen for learning. Two rule sets were produced by induction over two learning sets of exactly the same size, yet rule correctness varied drastically. This suggests that events of a given class appear in many distinct clusters in the event space. If learning events are taken from only a few of the clusters, then rule performance will be poor, as was the case in the run shown in Table 2. If, however, the learning events contain at least one element from each cluster, the rules should have relatively good performance. Also, we would expect the more correct rules to have a larger number of complexes than the poorer rules. This was indeed the result obtained -- the good rules had, on the aver-

age, almost twice as many complexes as the poorer rules.

The results for the soybean data are shown in Table 3. Characteristic descriptions took the longest amount of time to induce, by a large margin. This may be due to the extra overhead involved in working with longer descriptions. Only the results of one run are shown in Table 3 – the second run, with a different set of learning events, produced similar relative induction times.

More interesting results are shown in the "Complexity" column. The simplest rules were the discriminant descriptions induced from examples. However, the rules produced during the second run through the data were considerably simpler in the case of the charactersitic descriptions. The discriminant rules induced from the examples had almost exactly the same complexity in both runs. It appears that characteristic descriptions are more performance sensitive to the distribution of learning events

Rule Type	Induction Time	Complexity	"OR"	"AND"	Normalized	%1st Rank	%Only Choice
Expert Rules	–	20.52	maximum	minimum	no	75.63	68.91
			maximum	average	no	78.99	71.43
			maximum	minimum	yes	79.83	70.59
			maximum	average	yes	76.47	73.05
Characteristic Description	412.23 mins	102.05	maximum	minimum	no	56.30	56.30
			maximum	average	no	97.48	94.12
			maximum	minimum	yes	71.43	69.75
			maximum	average	yes	97.48	94.96
Discriminant From Examples	296.25 mins	13.71	maximum	minimum	no	90.76	90.76
			maximum	average	no	91.60	91.60
			maximum	minimum	yes	93.28	87.39
			maximum	average	yes	94.12	88.24
Discriminant From Characteristic	22.88 mins	15.17	maximum	minimum	no	96.64	96.64
			maximum	average	no	96.64	96.64
			maximum	minimum	yes	99.16	92.44
			maximum	average	yes	99.16	92.44

Table 3. Comparison of four different rule types for the soybean disease diagnosis domain.

given to the program. This sensitivity is explained by the order of criteria in the LEF. Since the primary criterion is to maximize coverage of positive events, there may be cases where a longer description will not work. This will happen frequently if the learning events of a class are widely separated in the event space – events that are farther apart require a more general, and therefore shorter, description. If this is true, then the performance of characteristic descriptions should also vary with the learning events. Specifically, we would expect that a more complex description should not perform as well as a simpler one *if* we are evaluating conjunction as minimum. Similarly, we would expect that averaging would greatly increase the performance of the more complex rules.

The characteristic rules performed as predicted over the two runs. As shown in Table 3, the very complex characteristic descriptions performed poorly if "AND" was evaluated as minimum, but performance improved drastically if averaging was used. The second set of characteristic rules performed in the 85% accuracy range regardless of evaluation scheme.

In this domain, it appears that the method of inducing discriminant descriptions indirectly may be worthwhile. Although the rules induced in this way were slightly more complex, their performance was comparable to that of the rules induced from examples, and considerably better than that of the rules written by experts. Typical rules in this domain are shown in Figure 21.

5.2. Incremental Learning

This section presents experiments designed to evaluate the performance of the incremental learning algorithm presented in Chapter 4. In that chapter, questions were raised as to whether incremental learning with perfect memory is efficient. There are two concerns here. The first is that the incremental method, as described, might not be any faster than learning by the standard method (i.e. by starting over each time new examples are presented). The second concern is that the incremental method, even if it is faster than the single step method, will produce rules that are more complex. This concern is especially relevant because the only way to specialize a complex is by splitting it into two or more

Characteristic description:

[precipitation = above_normal][temperature = normal..above_normal][severity = minor..potentially_severe]
[condition_of_leaves = abnormal] [leaf_spot_color = brown]
[leaf_spot_growth = scattered_with_concentric_rings,necrosis_across_veins][leaf_spot_size = greater_than_eighth_inch]
[shot_holing = present][position_of_affected_leaves = scattered_on_plant]
[condition_of_leaves_below_affected_leaves = unaffected][stem_cankers = does_not_apply] [fruit_spots = colored_spots]

Discriminant description induced from characteristic description:

[leaf_spot_color = brown][leaf_spot_growth = scattered_with_concentric_rings,necrosis_across_veins]
[position_of_affected_leaves = scattered_on_plant][fruit_spots = colored_spots]

Discriminant description induced from examples:

[leaf_spot_growth = scattered_with_concentric_rings,necrosis_across_veins]

Description written by domain expert:

[leaf_spot_growth = scattered_with_concentric_rings]:0.90
+
[time_of_occurrence = august..october][shot_holing = present]:0.50
+
[leaf_spot_size = greater_than_eighth_inch]:0.45
+
[time_of_occurrence = august..october][fruit_pods = diseased][fruit_spots = colored_spots]:0.10
+
[seed_discoloration_color = black]:0.05
+
[leaf_spot_margins = water_soaked]:0.05
+
[yellow_leaf_spot_halos = absent]:0.05

Figure 21. Four different rule types for identifying the soybean disease alternaria leaf spot.

complexes. If the specialization step in the incremental method produces too many complexes, the resulting rules will be so complicated as to be useless. On the other hand, it may be that the generalization steps will simplify the rules enough so that they are acceptable.

The experiments in the last section showed that viable discriminant rules could be produced from characteristic descriptions in two of the test domains. Characteristic descriptions may be especially useful for incremental learning. Since such descriptions contain large conjuncts, they are more specialized.

Because a major concern here is that specialization will produce overly complicated rules, it may be worthwhile to learn incrementally using characteristic descriptions. Hopefully, these will not have to be changed much during rule specialization. This could lead to less complex incremental rules. As before, if shorter descriptions are necessary, we can induce them in a small amount of time from the characteristic descriptions.

In order to test these ideas, GEM was applied incrementally to each of the three application areas used in the previous section. From an initial set of learning events, three types of rules were induced: discriminant from examples, characteristic from examples, and discriminant from characteristic. For each class, a random number of new events (i.e. events not used in the previous learning step) were selected. These were added to the learning events. GEM was applied again using the rules formed in the last step. The entire process was repeated until no learning events remained. At each step in the learning process, rules were also formed in single step fashion (i.e. with no input hypotheses) for comparison purposes. For example, in Figure 22 the initial learning set consisted of one event per class (a total of seven events out of the thirty five available). A random number between zero and one was generated. This number was used to determine the percentage of the remaining events of the first class to be added for the second learning step. Another random number was generated to select the number of events of the second class, and so on. This resulted in a new learning set in which a total of fourteen events were distributed randomly among the classes. These events, and the rules formed in the initial step, were used as input to GEM. Once rules were formed, the event selection process was repeated, producing input for the third learning step.

The rules formed at each stage were tested on examples using the ATEST parameter settings that worked best in the given domain (as determined in Section 5.1). The entire experiment was repeated twice for each domain. In all cases, disjoint rather than intersecting covers were formed. As in the previous section, different results might be obtained if intersecting rules were used.

In the small mayfly nymph recognition domain, problems that may arise in the two larger domains should exist to a lesser degree. The results in Section 5.1 lead to the expectation that characteristic descriptions will not differ much from discriminant ones. All induction times and rule complexities should be small compared to the other two application areas.

Figure 22 shows the complexity of the four different rule types during the learning process. Somewhat different results were obtained when different learning events were used. In Figure 22, the discriminant rules induced incrementally from examples were the most complex. A second run produced simpler discriminant rules and more complex characteristic ones.

Figure 23 shows the induction times for three rule types. Again, times shown are for a Pascal implementation of GEM running on a VAX 11/780. The time to induce characteristic rules is included in the time to induce discriminant rules from characteristic ones; in general, about ninety percent of this time was used in inducing the characteristic rules. The results here show that the incremental method provides significant improvement in induction time. The second repitition of this experiment produced results similar to those shown in the figure.

Figure 24 shows the performance of the rules induced in this domain. All three rule types show a steady increase in performance. Similar results were obtained during the second run, although all the rules performed better (in absolute terms) with a different set of learning events.

The incremental method worked quite well overall. In this area the method of inducing discriminant descriptions from characteristic descriptions produced better results than learning incrementally with discriminant rules. Performance of the rules was dependent on which events were used for learning.

The KPK chess endgame domain is somewhat less complex than the soybean disease problem, but it presents problems of its own. As was discussed in Section 5.1, the nature of this problem is such that inducing simple rules from examples is very difficult, if not impossible. Also, results in that section showed that GEM did not produce good characteristic descriptions, and that attempting to induce

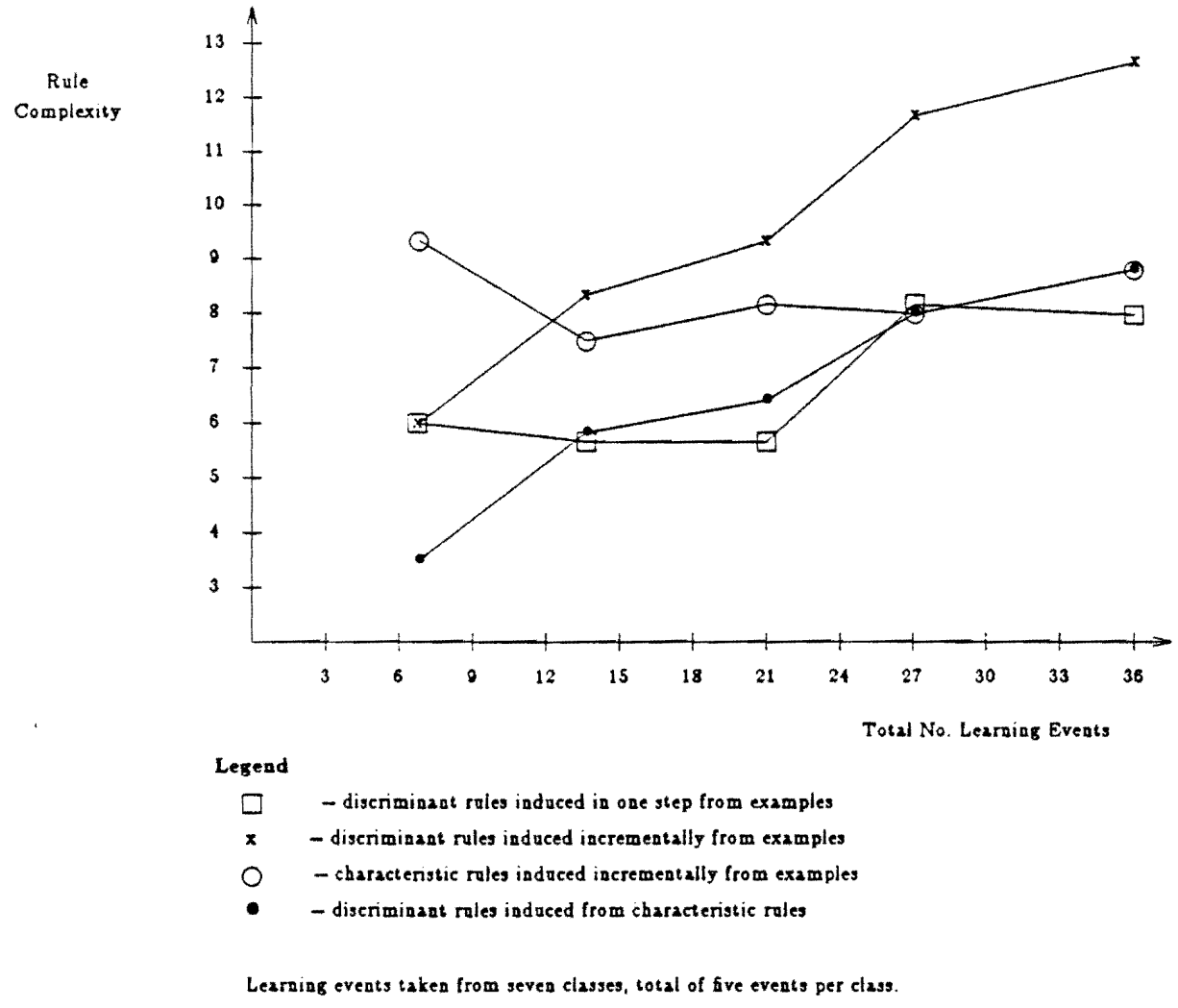


Figure 22. Complexity of four different rule types for identification of *Stenonema* mayfly nymphs.

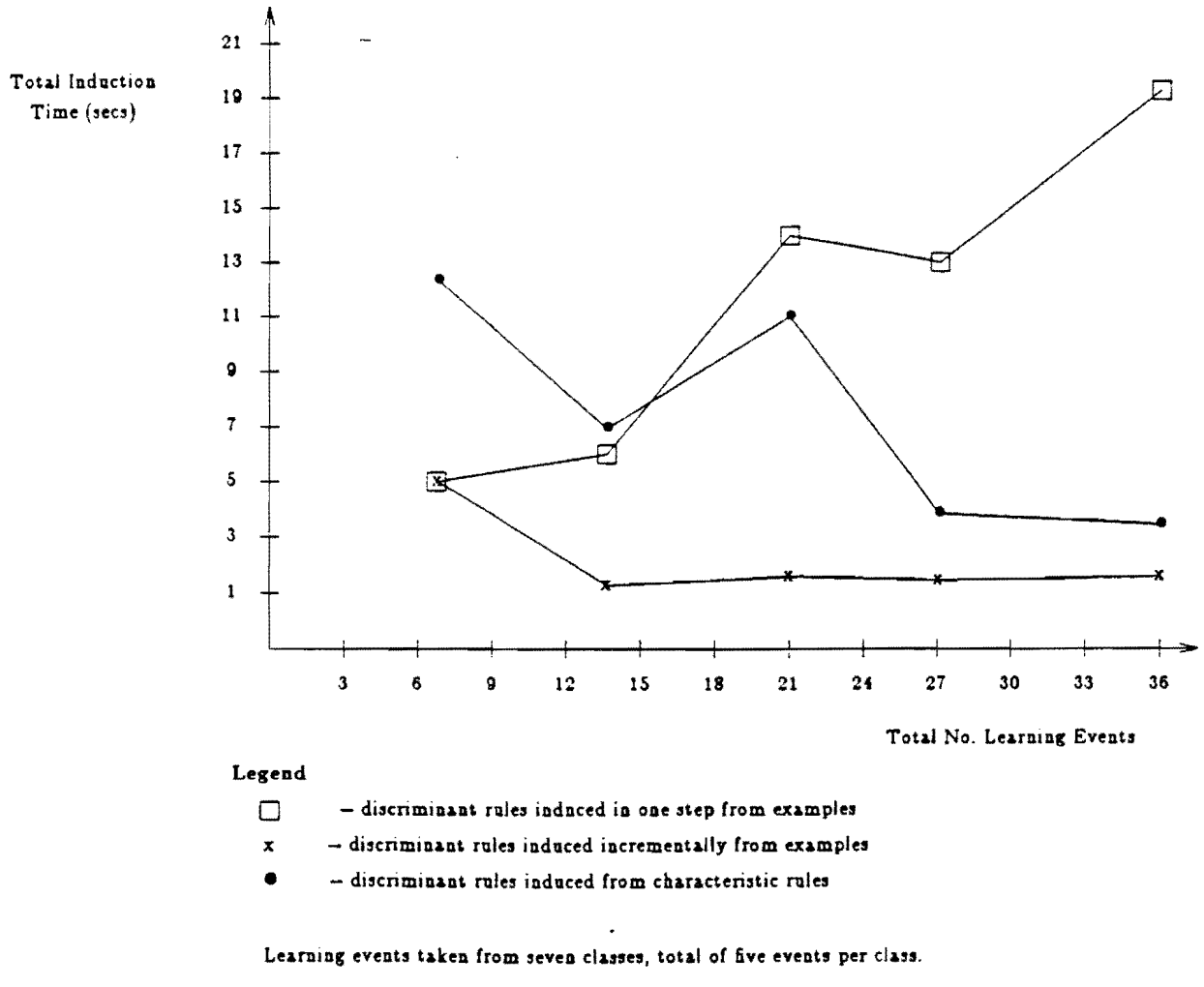


Figure 23. Induction times for three different rule types for the identification of *Stenonema* mayfly nymphs.

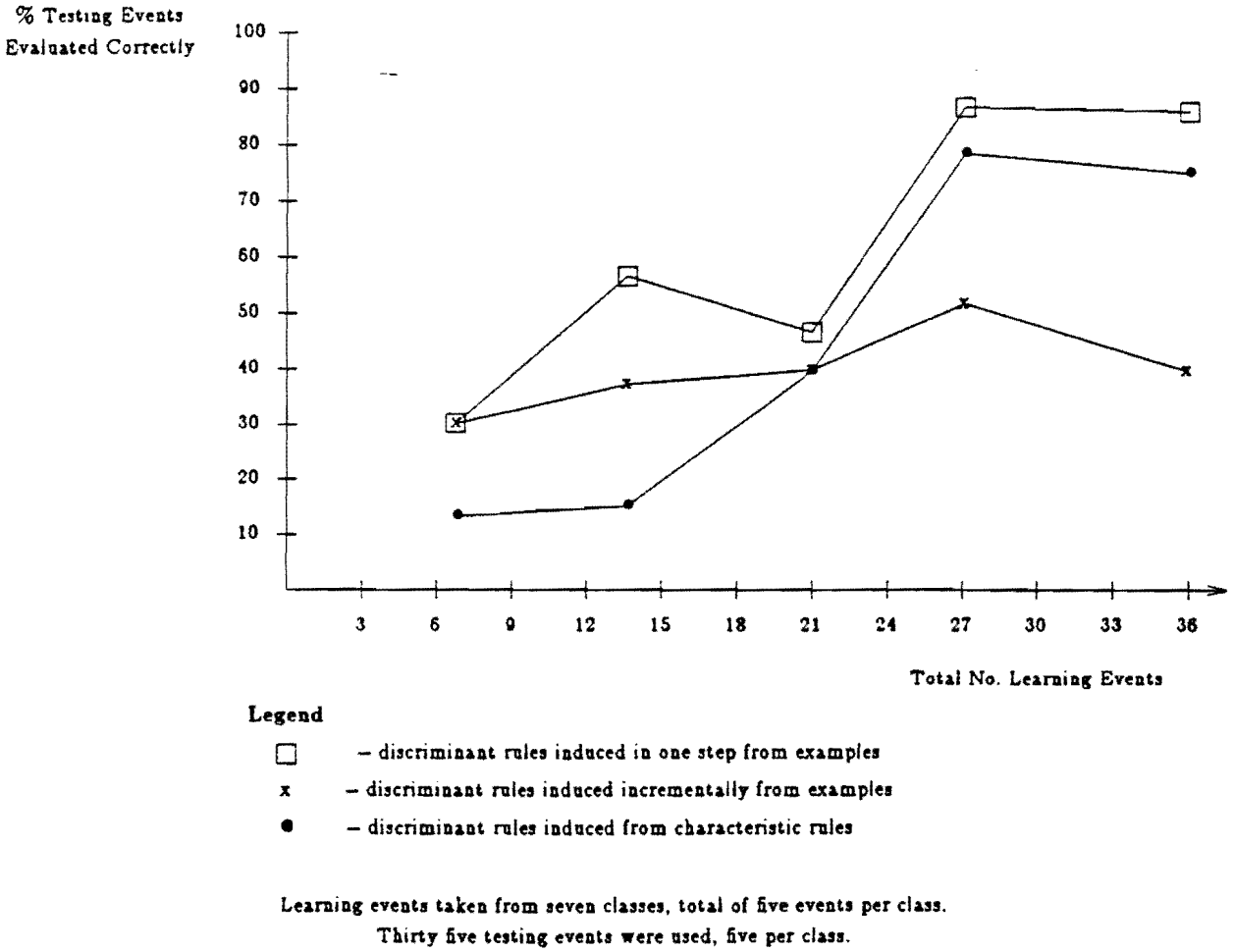


Figure 24. Performance of three different rule types for identification of *Stenonema* mayfly nymphs.

discriminant descriptions from them resulted in no changes to the characteristic rules. In other words, the discriminant descriptions produced were identical to the characteristic ones.

Figure 25 shows the complexities of the three different rule types in the three learning processes. Characteristic descriptions were more complex, but not markedly so, than discriminant descriptions. Unfortunately, the extra complexity generally came from the addition of a few selectors to complexes

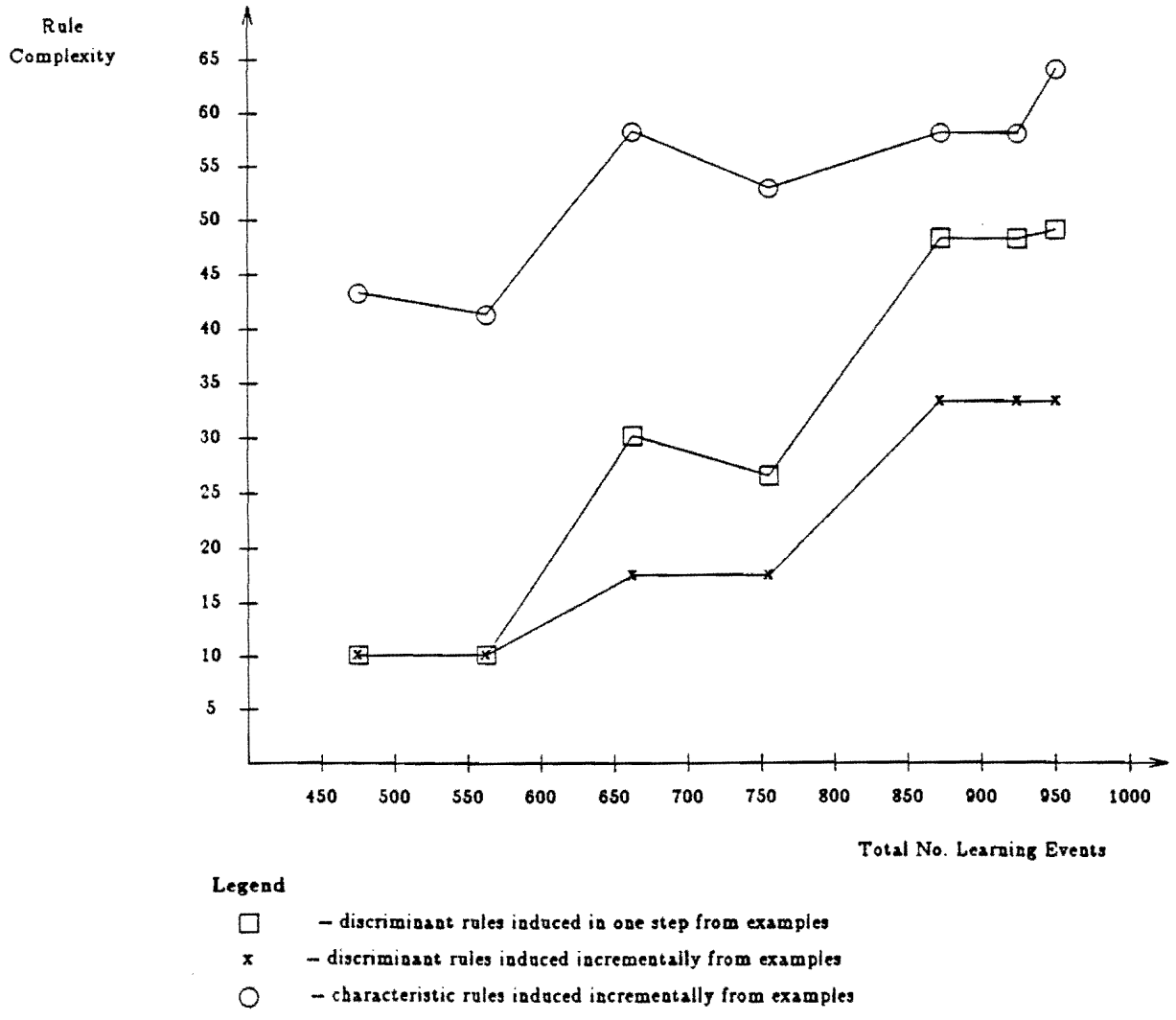
that existed in the discriminant descriptions; the characteristic descriptions were not more conjunctive. When this experiment was repeated, similar results were obtained. However, as in Section 5.1, the results for the second run produced rules that were considerably more complex than those shown in Figure 25.

Figure 26 shows the induction times for the induced rules in this domain. Here, there was little difference (except at the initial stage) between the time taken to induce characteristic and discriminant descriptions. As in the mayfly domain, the incremental method provided a considerable speed-up in learning time.

Figure 27 shows the "learning curve" for the induced rules. These rules performed very poorly and exhibit an odd behaviour -- the curve goes down for all three rule types. The second run in this domain produced better (though more complex) rules which improved in performance as new events were added. The behaviour shown in Figure 27 may be explained by the hypothesis presented in Section 5.1. If we assume that poor performance is due to poor representation of different disjuncts in the learning set, then the addition of new events to the existing disjuncts will cause them to be extended to cover negative events not in the learning set.

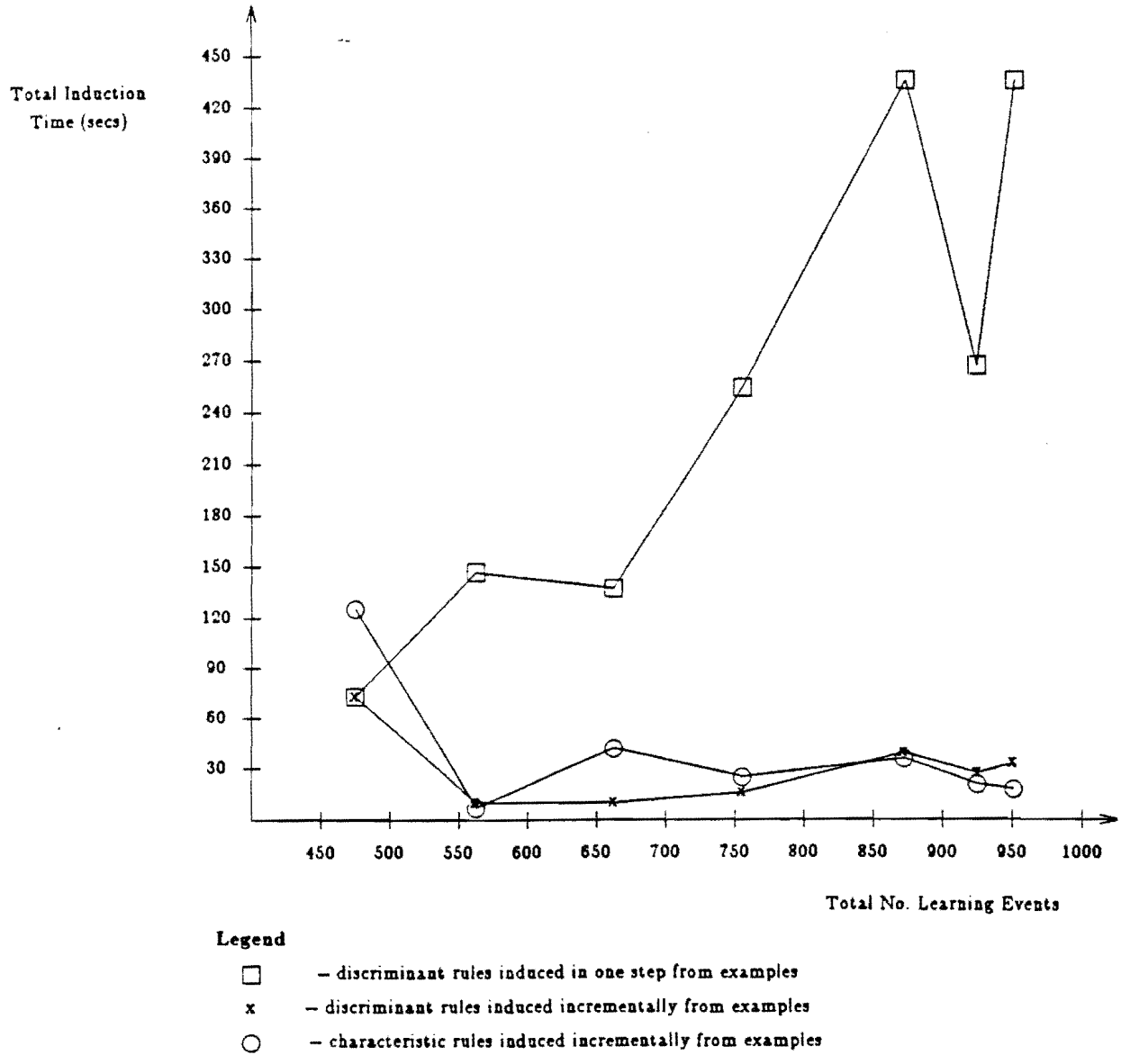
Since the soybean disease domain is the largest of the three being tested, it should present the most potential problems. The results in Section 5.1 show that induction in this domain takes a large amount of time, and that the descriptions produced may be quite complex. For incremental learning to be successful here, a large decrease in induction time is necessary. This must be coupled with the formation of rules that are not much more complex than those produced by non-incremental induction.

Figure 28 shows the complexity of four different rule types at various stages of the learning process. As expected, the characteristic descriptions were by far the most complex. However, the discriminant descriptions learned incrementally were not much more complex than those formed by the one step method. The discriminant rules induced from the characteristic descriptions were considerably larger than discriminant rules induced from examples. Similar results were obtained when the experi-



Learning events taken from two classes, total of 349 from class *DRAWN*, 602 from class *WON*.

Figure 25. Complexity of three different rule types for the KPK endgame.



Learning events taken from two classes, total of 349 from class *DRAWN*, 602 from class *WON*.

Figure 26. Induction times for three different rule types for the KPK endgame.

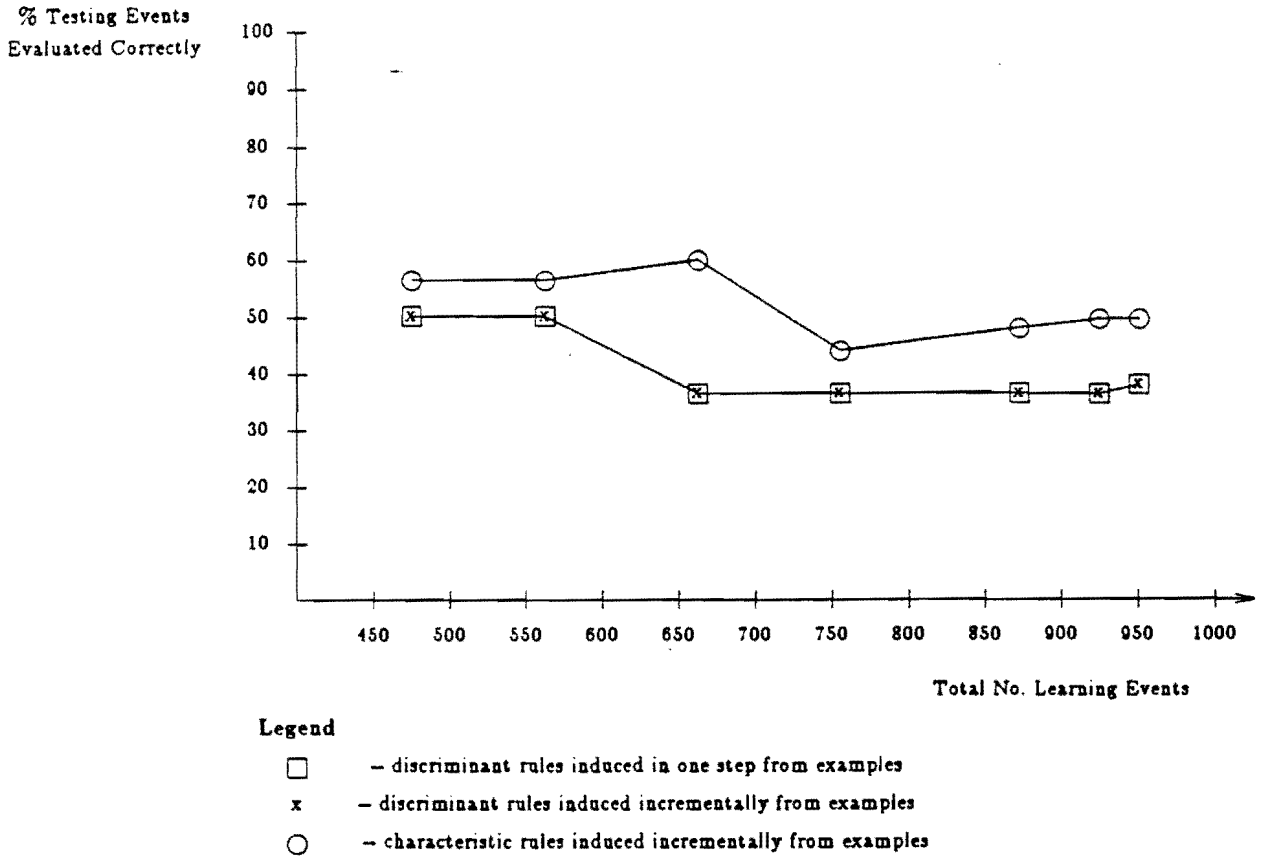


Figure 27. Performance of three different rule types for the KPK endgame.

ment was repeated with a different set of learning events.

Figure 29 shows the total induction time necessary for forming the rules described in Figure 25. Note here that the time to induce the "discriminant from characteristic" rules includes the time to induce the characteristic descriptions. The incremental method worked quite well; induction time for incremental learning appears to be a function of the number of new events rather than the total number

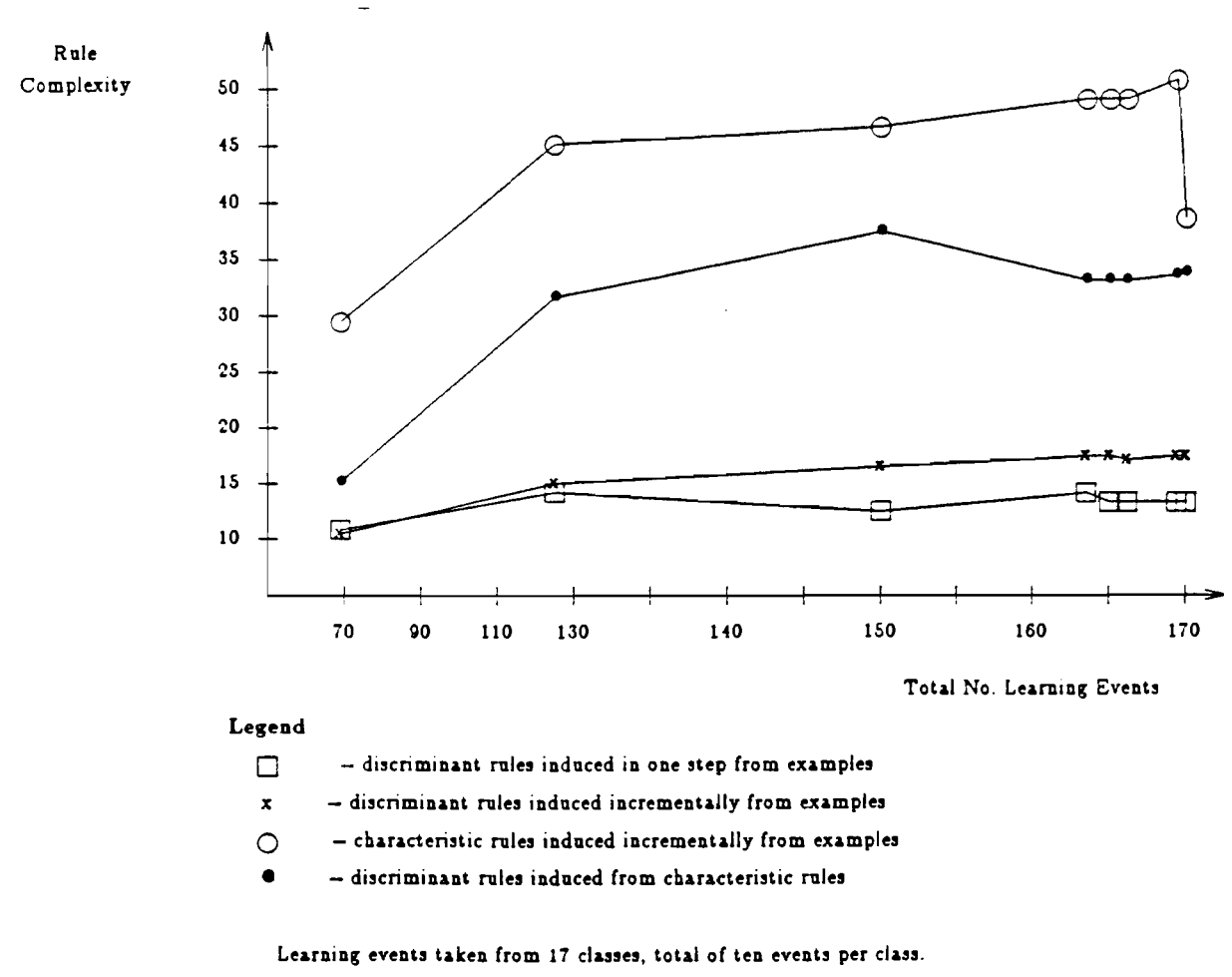


Figure 28. Complexity of four different rule types for soybean disease diagnosis.

of events, as desired. Similar results were obtained for the second run, although induction times for the characteristic descriptions were larger.

Figure 30 shows the performance of the three different rule types on 119 testing examples. All three rule types showed a fairly steady increase in performance as new events were added. The tem-

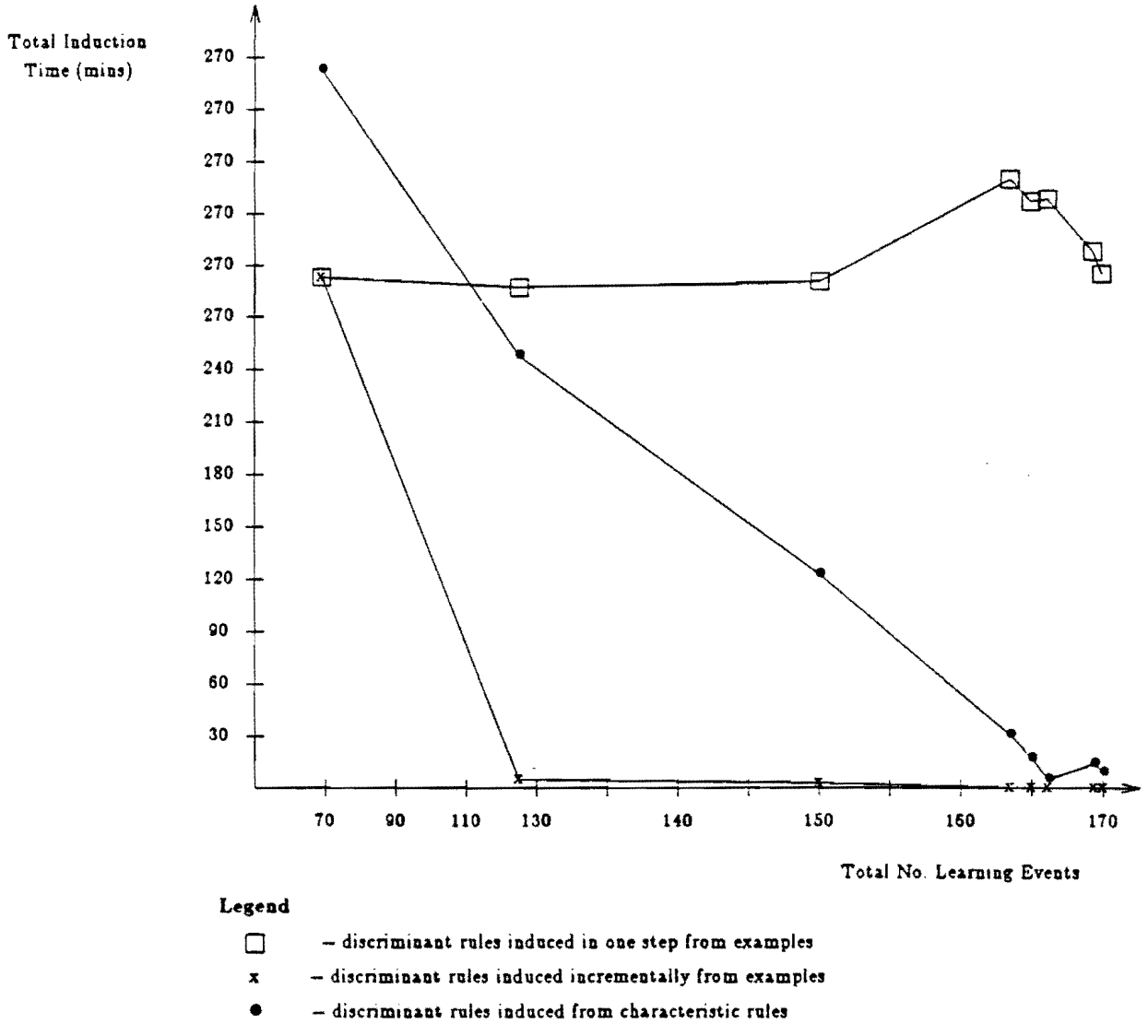


Figure 29. Induction times for three different rule types for soybean disease diagnosis.

porary dips in performance may be attributed to over-generalizations due to the addition of new, uncovered events. Such errors are corrected as new events are added later. The second run produced similar results, except that the discriminant rules induced indirectly had a markedly poorer performance.

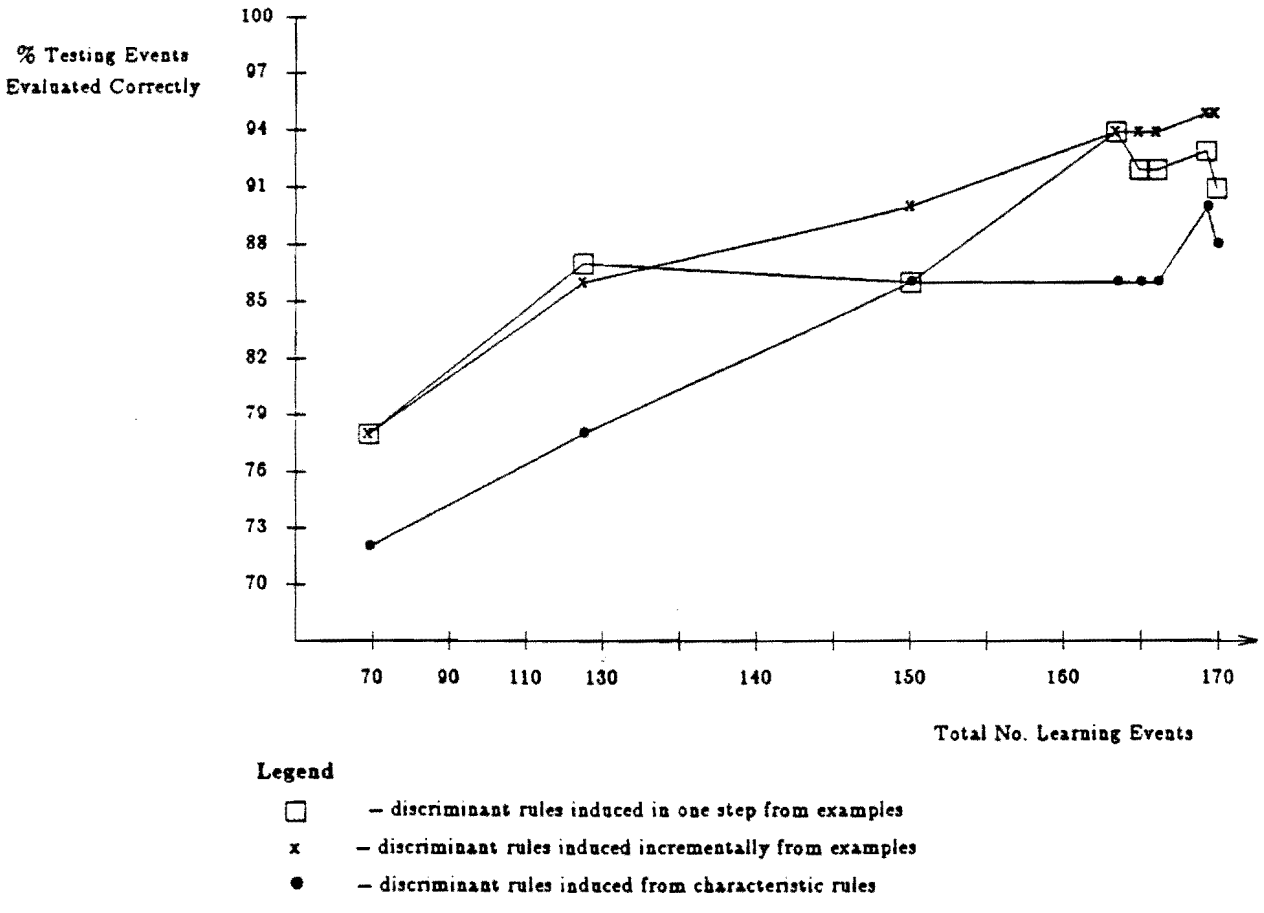


Figure 30. Performance of three different rule types for soybean disease diagnosis.

Overall, the incremental learning method worked as well as the non-incremental method and took considerably less time. However, in this domain, incremental learning with characteristic descriptions does not appear to be worthwhile. Compared to learning discriminant descriptions directly, this method produced larger rules and took considerably more time. These results are somewhat at odds with those obtained in the mayfly domain. Some ideas about the causes of these variations in rule performance and rule complexity are presented in the next chapter.

6. CONCLUSION

In general terms, the goal of the research described in this thesis was to build software tools that would aid a domain expert in expressing knowledge in a rule formalism. The paradigm for rule acquisition presented in Chapter two was used to generate criteria for these tools. Specifically, the aim was to provide useful programs for inductive inference and rule base testing. Chapter five described experiments which characterized the performance of the induction program and the rules it produced.

It is obvious from the results of these experiments that incremental learning with perfect memory is a viable way to form decision rules from examples. In all three application areas, incremental rule induction was faster than single step application. The rules produced tended to be slightly more complex, but their performance was comparable to rules induced using non-incremental means.

In section 5.2, reasons were given for believing that incremental learning might work best on characteristic descriptions. However, the characteristic descriptions produced by GEM tended to be large and took much time to produce. The performance of characteristic descriptions on testing events depended on the evaluation scheme to a larger extent than was the case for discriminant descriptions. Also, peformance was affected by the events chosen for learning. This effect was more marked for the characteristic rules.

None of these results suggest that characteristic descriptions are useless. It is likely that the problems discussed in chapter five were caused by the way in which characteristic descriptions were formed. It appears that a special method will have to be developed to produce such descriptions.

Another important point is that the performance of rules induced by GEM is dependent on the nature of the events used for learning. This is not terribly surprising, but the effect seems to be stronger in certain domains (e.g. the KPK chess endgame). Even more interesting, the performance of different evaluation schema depends not only on the domain and the rule type, but also on the learning event set used to induce the rules. This suggests that a method for selecting important learning events [Michalski and Larson 78] could be very useful.

Some hypotheses about these variations were presented in section 5.1. It seems probable that the variations are due to the relative locations of the learning events in the event space. If events of different classes are in small clusters packed closely together, then the quality of the rules produced depends on whether events from each cluster are available. In any case, good characteristic descriptions cannot be produced in such domains; this situation was observed in the KPK application. The chess endgame domain, due to the nature of the attributes used, is highly disjunctive. That is, if we view the problem as defining a function mapping events to classes, the chess problem is a densely specified function – there are not "don't care" areas. Any characterization of such a problem will be disjunctive. The soybean disease problem presented a different kind of event distribution – a large, sparse event space. In other words, the function defined by the examples is sparsely specified. Here, it seems likely that the events of a single class were distributed throughout a large area, most of which consists of "don't care" regions. In such a problem space, rules are often incorrect not because they cover negative events but because they fail to cover enough positive events. Only in the mayfly identification domain did the characteristic descriptions work as expected. However, the domain was so small that characteristic rules often did not differ much from discriminant ones.

The variations in performance observed for different evaluation schema was also related to the problem area and to the available learning events. This is not surprising, as changing the evaluation scheme changes the area of the event space that each rule covers. Using average for conjunction simply extends the boundary of the area covered by a conjunct. Therefore, averaging works well in areas like the soybean disease domain because the major problem is that rules do not cover events that they should. It fails in domains like the chess problem because rules are too general under averaging. Because the KPK problem is by nature disjunctive, averaging causes individual conjuncts to cover areas of the event space that they should not. A similar situation applies to continuous evaluation of selectors in linear variables (i.e. normalizing the difference between a selector and an event to a value between zero and one rather than treating selectors as boolean conditionals). Normalization had little effect in the soybean domain because there were few linear variables. However, in the mayfly domain, where

almost all the attributes were linear, normalization caused the rules to become overly general.

All of this suggests several possible directions for future research. The obvious first step is to find a good method for producing characteristic descriptions. The motivation for using characteristic rules is strong, so a workable means for producing such rules should prove quite useful. An algorithm for producing characteristic descriptions, once developed, could be easily tested using the methods presented here.

Further study is necessary on the issues involved in characterizing an application area. The results presented here, as well as common sense, suggest that different domains will require different learning methods. Learning tools will probably have to be applied to many more application areas before any coherent pattern emerges. Ideally, a learning system will be able to accept events in a problem area, select those that are most relevant for learning, suggest a learning method, and define an evaluation scheme to be used on the resulting rules.

The ADVISE system provides an excellent framework for research in this area to proceed. The tools described in Chapter four will be attached to the QUIN relational data base system, which, with associated editors for modifying knowledge, will provide an integrated interface for the domain expert building a knowledge base. Similarly, the ADVISE architecture provides a strong foundation for the addition of further learning and testing tools.

APPENDIX

USER'S GUIDE FOR GEM AND ATEST

GEM and ATEST are Pascal programs consisting of approximately 5,000 lines of code each. The programs run on the University of Illinois Department of Computer Science VAX 11/780 under the Berkeley Unix Operating System. The program load modules are each about 190K bytes. Run-time memory requirements vary with problem size, but a minimum of 192K bytes is necessary. GEM and ATEST are currently constrained to problems using no more than 60 variables with an overall total of 1,160 values. A single variable may have up to 58 values. All of these limits are constants defined in the program source code.

Both programs take their input in the form of relational tables. Relational tables are a convienient format for representing events of the type dealt with by ATEST and GEM. This also allows the programs to be used as operators by the QUIN relational database system [Spackman 83]. Both programs read from standard input and write to standard output.

Input to GEM and ATEST consists of a single file containing a series of relational tables. A relational table is composed of three parts: a table-name, a list of column names, and a set of tuples containing the data. In general, columns may be entered in any order. The columns accepted for each table type are defined in the section describing each program. The length of the tuples, and the type of information in them, must correspond to the appropriate column names. The table name and the column headings, as well as each tuple in the table, must be on a single input line. If all the columns in a table will not fit on a single line, the table may be split into several tables, each of which has some part of the columns. Individual items on a line are separated by any number of spaces.

Table names are of two types. First, there are tables which have only a single part name (such as "parameters"). There are also table names which consist of two parts. These are of the form "specific-

general", where each of "specific" and "general" is an alphanumeric string. Tables of this type (e.g. the "-names" tables) must have a specific name associated with them because there may be several tables of the same general type. In the table definitions that follow, any table whose name is given with an intial "-" must have a specific name preceding the "-" in program input. A specific name, and any other alphanumeric string entered as input, must be a continuous string of characters containing only letters and numbers and beginning with a letter. The maximum length of such strings is a program constant, currently set to twenty.

1. GEM Input and Output

1.1 The title table

This table provides a header for an input file. It is not used in any way by the GEM program. The title table is therefore optional. It consists of two columns:

- **#**

  Optional column which contains the row number of the text in the next column. Row numbers must begin with 1 and continue sequentially.

- **text**

  Each entry in this column consists of a string of characters that are a single line in the title of the input file. If there are any blanks or tabs in the row, the string must be surrounded with quotes. If single quotes appear in the string, double quotes must be used to surround it, and vice versa.

A sample title table is shown below.

```
title
#   text
1   "This is a sample title table"
2   "of the type input to GEM."
3   "It is not used by the program."
```

1.2 The parameters table

The parameters table is mandatory. This table contains values which control the execution of the program. Many of the parameters have default values, as noted below; these columns need not be entered in the table if the default value is acceptable. Each row of the parameters table represents one run of the program; this allows the user to specify many different runs on the same data in a single input file.

- **run**

Optional row number. The first row must be numbered 1 and rows must be numbered sequentially.

- **echo**

Optional specification of which tables are to be echoed to output. Values in this column consist of a string of characters, each of which represents a single table to be echoed. There must be no blanks or tabs in this string. Legal characters for the echo column, and the tables they represent are:

t	–	the title table
p	–	the parameters table
c	–	the criteria tables
d	–	the domaintypes table
n	–	the -names tables
v	–	the variables table
h	–	the -inhypo tables
e	–	the -events tables
b	–	the -children tables

The default value for the echo parameter is pcvh.

- **mode**

Optional specification of the way which GEM is to form rules. The legal values for this column are ic, dc and vl. In ic (intersecting cover) mode, GEM will produce rules that may intersect over areas of the event space where there are no learning events. This value is the default. In dc (disjoint cover) mode, GEM will produce rules that do not intersect at all. In vl (variable valued logic) mode, the rules produced will be order dependent. That is, the rule for class "n" will assume that the rules for the classes 1 through n-1 are not satisfied.

- **maxstar**

This parameter controls the size of the partial star kept during star formation (see Chapter 4). Default value is 10. Maximum value is a program constant, currently set to 100.

- **trim**

Boolean parameter (legal values are "yes" and "no") which, if on, causes GEM to trim the covers it produces. Trimming is done by removing values from the right hand sides of selectors in the cover. A value is removed if it does not appear in any event of the corresponding class. Trim will not change which variables appear in the rules. Default value is" on."

- **wts**

Boolean parameter (legal values are "yes" and "no") which, if on, causes GEM to associate two weights with each complex it produces. The first weight produced is the percentage of positive events that the complex covers. The second weight is the percentage of events that this complex, and no other complex in the rule, covers.

- **criteria**

The name of the criteria table to be used for this run. The name must be less than twenty alphanumeric characters with no blanks, and a -criteria table with that name must appear in the input file.

A sample parameters table is shown below. Values shown in the first row are the default values for the parameters. Note that the default value for the criteria column is the only -criteria table specific name for which it is not necessary to actually define a table (see below). Since the second row contains the string "maxim" for the criteria column, a table named "maxim-criteria" must be defined.

parameters

run	echo	mode	maxstar	trim	wts	criteria
1	pcvh	ic	10	yes	no	default
2	pc	dc	50	yes	yes	maxim

1.3 The -criteria tables

This table type is used to define a lexicographic functional (LEF). The LEF is used by GEM to judge the quality of complexes formed during learning. A LEF consists of several criterion - tolerance pairs. The ordering of the criteria in the LEF determines the relative importance of each. The tolerance specifies the estimated error within each criterion. See Chapter 4 for a more detailed discussion of when and how the LEF is used.

A criteria table name consists of two parts -- the specific name, which must appear in the "criteria" column of the parameters table, and the general name, -criteria. Any value in the criteria column of the parameters table except *default* must have a corresponding -criteria table.

The criteria table consists of three columns, all of which must be present:

- **#**

The order of this criterion in the LEF. The first row must be numbered "1", and the rows must be numbered sequentially. This column is optional.

- **criterion**

This column specifies the functional which is to be used for this row of the LEF (the rows of the table give the ordering in which the functional will be applied). There are five different criteria available:

1 -- Maximize coverage of positive events that are not covered by previous complexes. Complexes in a cover are produced sequentially; this criterion specifies the selection of complexes that cover events not covered by earlier complexes in the sequence.

2 -- Minimize the number of selectors.

3 -- Minimize the total cost of the variables used (see section 1.5).

4 -- Maximize the total number of positive events covered.

5 -- Maximize the number of selectors.

- **tolerance**

This must be a real number between 0 and 1. The tolerance specifies the uncertainty in the associated criterion. For example, say the best complex in a list had a value of 100 for some criterion and the tolerance for the criterion was 0.1. The absolute tolerance allowed is computed by multiplying the tolerance by the best value, yielding an absolute tolerance of 10. Then any complex with a value between 90 and 100 would be regarded as having the same value as the best complex for this criterion.

The default-criteria table is shown on the next page. This is the only incarnation of the -criteria table which need not be entered explicitly.

default-criteria

#	criterion	tolerance
1	1	0.00
2	2	0.00

1.4 The domaintypes table

The domaintypes table is used to define domains for attributes. This table is optional, but it is convienient if several attributes have the same set of possible values. The table consists of three columns, all of which must be included:

- **name**

  This is the name of the domain being defined. Must be a string of less than twenty alphanumeric characters with no white space.

- **type**

  The type of the domain being defined. Three domain types are legal: nominal (nom), linear (lin) or cyclic (cyc). A nominal domain consists of discrete, unordered values (e.g. color is a typical nominal domain). A linear domain consists of discrete, ordered values (e.g. size). A cyclic domain is discrete values in a circular ordering (e.g. the integers modulo 4).

- **levels**

  An integer value between 1 and 57 specifying the size of the domain. The maximum domain size is related to the size of sets allowed in the Pascal implementation. GEM was originally implemented on a machine which allowed sets to have a cardinality of no more than 58. On other machines, the maximum domain size is a declared program constant.

The domaintypes table is used in conjunction with the variables table and the -names table. An example of the use of these three table types is shown after the definition of the -names table (see section 1.6).

1.5 The variables table

The variables table is mandatory -- it specifies the names and types of the variables used to describe events. It may contain up to five columns:

- **#**

Optional numbering of variable declarations. The first row must be numbered "1", and rows must be numbered sequentially.

- **name**

Optional column associating a name with the variable. If this column is omitted, variables will be given names of the form x#, where # is the row the variable appears in. If a domaintypes table is being used, then the variable name may consist of two parts — "name"."domain-name", where "domain-name" is a string appearing in the name column of the domaintypes table.

- **type**

Same as the type column in the domaintypes table.

- **levels**

Same as the levels column in the domaintypes table.

- **cost**

A real number specifying how "expensive" this variable should be to use compared to other variables. Used in computing criterion 3 in the LEF (see the definition of the -criteria table).

The variables table may be used in conjunction with the domaintypes and -names table. An example of a variables table is shown after the definition of the -names table, below.

1.6 The -names tables

This table is optional. The -names table is used to specify names for values in a domain. If no -names table appears for a variable or domain, then the values for that domain are assumed to be the integers beginning with 0. The specific name of a -names table must be the name of a variable in the variables table or an entry in the name column of the domaintypes table. A -names table consists of two columns, both of which are mandatory:

- **value**

This is the integer equivalent of the value to be defined in the next column.

- **name**

The name of the value being defined.

Below is a typical example of the use of the domaintypes, variables and names table. In this example, two variables ("long" and "wide") are defined to be boolean with the values "false" and "true". The variable "color" may take any of the values "red", "blue" or "green". The variable "size" takes on integer values between 0 and 5. Note that if the domaintypes table were excluded, then the "type" and "levels" column would have to appear in the variables table.

```
domaintypes
name          type      levels
boolean        nom        2
colors         nom        3
range          lin        6


variables
#       name
1       long.boolean
2       wide.boolean
3       color.colors
4       size.range



boolean-names
value     name
0         false
1         true

colors-names
value     name
0         red
1         blue
2         green
```

1.7 The -inhypo tables

The -inhypo table is used to input rules to GEM for incremental learning. The specific name of this table must be the name of a decision class. If the name given for a -inhypo table has not been seen

before (i.e. in a -children table, see below), the associated class is assumed to be at the top of a structured rule base. The rules input in a -inhypo table are used as the initial covers when doing incremental learning. If incremental learning is not desired, then this table may be excluded.

- **#**

  Mandatory column associating a number with each complex in the rule. In -inhypo tables *only*, a single relational tuple may span more than one line. However, there must be only one # entry for every complex in the table.

- **cpx**

  Mandatory column giving a VL_1 declaration of the complex. A complex is presented as a series of selectors. Selectors may be separated by any amount of white space or new lines. A new complex is started only when a new entry for the # column (i.e. a number) is found. Each selector is an expression of the form [variable = values]. The brackets are mandatory. The variable may be any variable declared in the variables table, but the same variable may not appear twice in one complex. The values must be defined values for the variable given on the left of the "=" sign. Several values may be specified in one selector in any of the following forms:

  $$value_1,value_2$$

  $$value_1..value_2 \text{ (valid only for linear and cyclic variables)}$$

  $$value_1..value_2,value_4..value_5 \text{ (also valid only for linear and cyclic variables)}$$

  The symbol "," in a selector means "or" and the symbols ".." specify a range of acceptable values. So, the selector [color = blue,green] is read "color is blue or green." The selector [size = 0..3] means "size is between 0 and 3, inclusive."

Below is an example of a -inhypo table which uses the variables defined in the example at the end of the previous section. This rule would be used as an initial hypothesis for the class "ONE". The rule consists of two complexes, and is read " If long and wide are false or size is 1 then the event is of class ONE."

ONE-inhypo

#	cpx
1	[long = false][wide = false]
2	[size = 1]

1.8 The -events tables

These tables are used to input events to GEM. The specific name given to a -events table corresponds to a single decision class. In GEM, if the specific table name has not been seen previously (in a -children table, see below), then a new class is created at the top of the rule base structure.

The column headers for this table type consist of variable names defined in the variables table. The values in the rows of the table must be legal values for the appropriate variables. Since many attributes may be used to describe an event, it is possible to split a -events table into several tables. This is done by repeating the table name (both specific and general), and using different column headings in each occurence. Column headings may not overlap, and each table must have the same number of events.

The -events table shown below uses the attributes defined in the example for the -names table, above. This table would associate four events with the class "ONE".

ONE-events			
long	wide	color	size
false	false	blue	0
true	false	red	1
false	false	red	0
false	false	blue	1

1.9 The -children tables

GEM accepts -children tables in order to define a structuring on a rule base. The specific name of the table must be the name of an already defined class, i.e. the name must have appeared as the name of a -events table. The rule base may be structured to arbitrary depth.

The -children table consists of two columns:

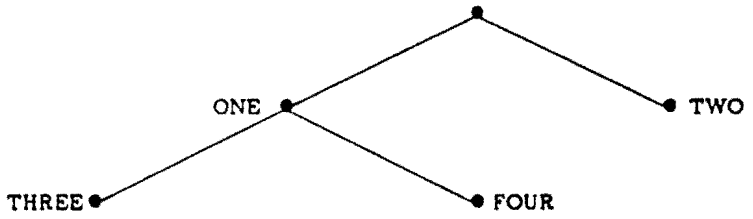
- node

This column is a string of characters giving the name of the node to be defined.

- **evts**

  This column specifies which events attached to the parent class also belong to this child node. It consists of a string of integers separated by commas or by "..", as in selectors. These numbers correspond to events associated with the parent. The parent's events are numbered in the order they appear in the -events table. Classes more than one level deep in the rule base use the event numbers associated with their ancestor at the *top* of the structure. This allows the user to specify all events with the same set of numbers.

The tree below shows how a sample rule base might be structured. In this rule base, classes ONE and TWO are siblings at the top of the structure. The class ONE has two sub-classes, THREE and FOUR.



The tables below define the rule base structure given in the tree. In this example, class ONE contains four events. The -children table assigns the first and last of these to class THREE, and the remainder to class FOUR.

ONE-events

#	long	wide	color	size
1	false	false	blue	0
2	true	false	red	1
3	false	false	red	0
4	false	false	blue	1

TWO-events

#	long	wide	color	size
1	true	true	green	2
2	true	true	red	3
3	false	true	green	3

ONE-children

#	node	evts
1	THREE	1,4
2	FOUR	2..3

1.10 An Example of Input to GEM

This subsection contains a detailed example of input to the GEM program. The sample input file shown in the left column below uses every table GEM accepts. Comments in the right column explain the corresponding table and provide pointers to the previous subsections for more detailed explanations. To input this example to GEM, the tables would be entered (in the order given) into some file using a standard text editor. If this file was called "gem.input," then executing:

gem < gem.input > gem.output

under the UNIX shell would cause GEM to run the example and save the results in the file "gem.output."

GEM Input — *Comments*

```
title
#       text
1       "Sample input file"
```

The title table is used only for reference and may be omitted.

```
parameters
echo  mode   maxstar   trim  wts   criteria
t     dc     50        yes   yes   maxim
```

The parameters table defines the way in which GEM will run. The table here tells GEM to run once, echo only the title table, form disjoint rules, use a maxstar of 50, trim the results, report weights and use the LEF defined in the maxim-criteria table. The parameters are defined in detail on pages 65-66. See Chapter 4 for an explanation of the terminology.

```
maxim-criteria
#       criteria tolerance
1       1        0.10
2       5        0.00
```

This table specifies a LEF (see section 4.1.1) of two criteria: maximize the number of new events covered and maximize the length of the rules. The first criteria has a tolerance of 10% – any complex whose value for this criterion is within 10% of the best complex will be regarded as equivalent to the best complex. The criteria table is defined in detail on pages 66-68.

GEM Input

domaintypes

name	type	levels
sizes	lin	3
colors	nom	4
nums	lin	10

Comments

Three types of variables with 3, 4, and 10 values, respectively, will be used. The definition of the domaintypes table is on page 68.

sizes-names

value	name
0	small
1	medium
2	large

Names to be associated with values of a variable with domain "sizes." The -names table is explained on page 69.

colors-names

value	name
0	red
1	blue
2	green
3	yellow

variables

#	name
1	size.sizes
2	color.colors
3	num1.nums
4	num2.nums

The variables that will be used to describe events for this problem. If the domaintypes table was not being used, then the *levels* and *type* columns would be in this table. As it is, we need only specify the name of the variable and its domain type. The variables table is defined on page 68.

ONE-events

#	size	color	num1	num2
1	small	red	0	0
2	medium	red	4	2
3	medium	blue	1	1
4	small	blue	2	3

This table contains events of the class ONE. Since this is the first class to be defined, it will be at the top of the rule base. Note that since no -names table was used for the "nums" domain, the values entered are integers. The "#" column here is optional.

GEM Input

TWO-events

#	size	color	num1	num2
1	medium	yellow	7	3
2	medium	yellow	8	3
2	large	green	4	4
3	medium	green	9	5

Events of class TWO, also at the top of the structured rule base. The -events tables are discussed on page 72.

ONE-children

node	evts
THREE	1,4
FOUR	2..3

This table defines the structure on the rule base. Class THREE is a child of class ONE, and contains its parent's first and fourth events. Class FOUR is defined similarly. The method of structuring rule bases is described in detail on pages 72-73.

ONE-inhypo

#	cpx
1	[size = small..medium][num1 = 0..7]

This table defines an initial rule for class ONE. This rule is input for the incremental learning algorithm. See page 70 for details on format and Chapter 4 for information on the incremental learning method.

TWO-inhypo

#	cpx
1	[size = small,large]
2	[size = medium][num1 = 8..9]

This table defines the rule for class TWO. This rule will serve as input for the incremental algorithm.

THREE-inhypo

#	cpx
1	[size = small]

This table defines the rule for class THREE. This rule will serve as input for the incremental algorithm.

FOUR-inhypo

#	cpx
1	[size = medium]

This table defines the rule for class FOUR. This rule will serve as input for the incremental algorithm.

1.11 An Example of Output From GEM

If GEM is given the input in part 1.10, it will produce output consisting of two parts: an echo of certain input tables (as per the echo parameter), and a summary of the results. This output is shown below.

GEM Output — *Comments*

```
title
#       text
1       "Sample input file"
```

The title table is the only table echoed because the value of the echo parameter in the input file was "t."

```
ONE-outhypo
#       cpx
1       [size = small..medium][color = red,blue]
        [num1 = 0..2,4] : 1.00, 1.00
```

The rule produced for class ONE are output in this table. This rule consists of a single conjuct; disjuncts are separated by entries in the "#" column, as in the next table. The first number following the complex is the percentage of positive (i.e. class ONE) events covered by this complex and no other complex. The second number is the percentage of class ONE events covered by this complex.

```
TWO-outhypo
#       cpx
1       [num1 = 7..9] : 0.50, 0.75
2       [size = large] : 0.25, 0.50
```

The rule produced for class TWO.

```
THREE-outhypo
#       cpx
1       [size = small] : 1.00, 1.00
```

The rule produced for class THREE.

```
FOUR-outhypo
#       cpx
1       [size = medium] : 1.00, 1.00
```

The rule produced for class FOUR.

```
This run used (milliseconds of CPU time):
System time     :    34
User time       :    184
```

Time taken to form the rules. This does not include input and output time.

2. ATEST Input and Output

2.1 The title table

This table provides a header for an input file. It is not used in any way by the ATEST program. The title table is therefore optional. It consists of two columns:

- **#**

  Optional column which contains the row number of the text in the next column. Row numbers must begin with 1 and continue sequentially.

- **text**

  Each entry in this column consists of a string of characters that are a single line in the title of the input file. If there are any white space charaters in the row, the string must be surrounded with quotes.

A sample title table is shown below.

```
title
#   text
1   "This is a sample title table"
2   "of the type input to ATEST"
```

2.2 The parameters table

The parameters table is mandatory. This table contains values which control the execution of the program. Many of the parameters have default values, as noted below; these columns need not be entered in the table if the default value is acceptable. Each row of the parameters table represents one run of the program; this allows the user to specify many different runs on the same data in a single input file.

- **run**

  Optional row number. The first row must be numbered 1 and rows must be numbered sequentially.

- **echo**

Optional specification of which tables are to be echoed to output. Values in this column consist of a string of characters, each of which represents a single table to be echoed to output. There must be no blanks or tabs in this string. Legal characters for the echo column, and the tables they represent are:

t	–	the title table
p	–	the parameters table
c	–	the criteria tables
d	–	the domaintypes table
n	–	the -names tables
v	–	the variables table
h	–	the -outhypo tables
e	–	the -test tables
b	–	the -children tables

The default value for the echo parameter is pcvh.

- **test**

This parameter tells ATEST whether to test rules it is given on events. Legal values are "yes," "no" and "sum." If test is "yes," ATEST will produce a confusion matrix for each testing class. If test is "sum," ATEST will only report a summary of the results for all classes. If test is "no," then the rules will not be tested on any events. The default value is "yes."

- **misclass**

If misclass is on (legal values are "yes" and "no," default value is "no"), then ATEST will print a trace of every event that was evaluated incorrectly.

- **cc**

If cc is on (again, values are "yes" and "no," default is "no"), then ATEST will perform consistency and completeness checking on the rules input in the -outhypo table.

- **andtype**

Determines how conjunction is evaluated. Legal values are "average" and "minimum;" the default value is "minimum."

- **ortype**

Determines how disjunciton is evaluated. Legal values are "maximum" and "psum;" the default value is "maximum."

- **norm**

Determines whether selectors in linear variables will be evaluated to a range between zero and one (normalized) or as a boolean conditional. Legal values are "yes" and "no," the default is "no."

- **threshold**

Real-valued parameter in the range 0 to 1. ATEST uses this parameter as the threshold for rule satisfaction. If a rule has a degree of consonance with an event greater than the value specified in this column, then the rule is deemed satisfied by the event. The default value is 0.50.

- **dropa2**

Real-valued parameter in the range 0 to 1. This parameter is used to determine when to stop using the α_2 weight in rule evaluation. See section 4.2.1 for a detailed explanation. The default value for this parameter is 1.00.

- **dweight**

Real-valued parameter in the range 0 to 1. This parameter is used to determine which modules to use when multiplying during consistency and completeness checking. Modules whose α_1 weights are below dweight are not used. The default value is 0.50.

A sample parameters table is shown below. Values shown are the default values for the parameters.

parameters

run	echo	test	misclass	cc	andtype	ortype	norm	threshold	dropa2	dweight
1	pcvh	yes	no	no	minimum	maximum	no	0.50	1.00	0.50

2.3 The domaintypes table

The domaintypes table is used to define domains for attributes. This table is optional, but it is convienient if several attributes take the same values. The table consists of three columns, all of which must be included:

- **name**

This is the name of the domain being defined. Must be a string of less than twenty alphanumeric characters with no white space.

- **type**

The type of the domain being defined. Three domain types are legal: nominal (nom), linear (lin) or cyclic (cyc). A nominal domain consists of discrete, unordered values (e.g. color is a typical nominal domain). A linear domain consists of discrete, ordered values (e.g. size). A cyclic domain is discrete values in a circular ordering (e.g. the integers modulo 4).

- **levels**

An integer value between 1 and 57 specifying the size of the domain. The maximum domain size is related to the size of sets allowed in the Pascal implementation the program is running under. GEM was originally implemented on a machine which allowed sets to have a cardinality of no more than 58. On other machines, the maximum domain size is a declared program constant.

The domaintypes table is used in conjunction with the variables table and the -names table. An example of the use of these three table types is shown after the definition of the -names table.

2.4 The variables table

The variables table is mandatory – it specifies the names and types of the attributes used to describe events. It may contain up to five columns:

- **#**

Optional numbering of variable declarations. The first row must be numbered "1", and rows must be numbered sequentially.

- **name**

Optional column associating a string of up to 20 alphanumeric characters with the variable. If this column is omitted, variables will be given names of the form x#, where # is the row the variable appears in. If a domaintypes table is being used, then the variable name may consist of two parts – "name"."domain-name", where "domain-name" is a string appearing in the name column of the domaintypes table. If a variable is specified in this way, then it is assumed to have values corresponding to those in the appropriate row of the domaintypes table.

- **type**

Same as the type column in the domaintypes table.

- **levels**

Same as the levels column in the domaintypes table.

The variables table may be used in conjunction with the domaintypes and -names table. An example of a variables table is shown after the definition of the -names table, below.

2.5 The -names tables

This table is optional. The -names table is used to specify names for values in a domain. If no -names table appears for a variable or domain, then the values for that domain are assumed to be the integers beginning with 0. The specific name of a -names table must be the name of a variable in the variables table or an entry in the name column of the domaintypes table. A -names table consists of two columns, both of which are mandatory:

- **value**

  This is the integer equivalent of the value to be defined in the next column.

- **name**

  The name of the value being defined.

Below is a typical example of the use of the domaintypes, variables and names table. The domaintypes table must be entered first. The table shown below defines three domains: boolean, colors and range. In this example, two variables ("long" and "wide") are defined to be boolean with the values "false" and "true". The variable "color" may take any of the values "red", "blue" or "green". The variable "size" takes on integer values between 0 and 5. Note that if the domaintypes table were excluded, then the "type" and "levels" column would have to appear in the variables table. Every variable must have a defined type and a defined domain size (number of levels). If some variables have domains defined in the domaintypes table and some do not, the "type" and "levels" columns must still appear in the variables table. The declarations must match for those variables whose domains are declared twice.

```
domaintypes
name          type      levels
boolean        nom        2
colors         nom        3
range          lin        6
```

```
variables
#       name
1       long.boolean
2       wide.boolean
3       color.colors
4       size.range

boolean-names
value     name
0         false
1         true

colors-names
value     name
0         red
1         blue
2         green
```

2.6 The -outhypo tables

The -outhypo tables are used to provide ATEST with rules to test. The specific name of this table must be the name of a decision class. If the name given for a -outhypo table has not been seen before (i.e. in a -children table, see below), the associated class is assumed to be at the top of a structured rule base. This name is somewhat confusing since -outhypo tables provide *input* rules. The name is used to correspond to the name of the output tables from the GEM program. This allows GEM output files to be used as ATEST input files with only minor modifications (changing the parameters and entering testing events).

Rules input in a -outhypo table may also have weights associated with them. Chapter 3 provides a syntax and semantics for these α-weights. When weights are encountered by ATEST, they are automatically associated with whatever expression was read in since the last time weights were seen.

- #

  Mandatory column associating a number with each complex in the rule. In -outhypo tables *only*, a single relational tuple may span more than one line. However, there must be only one # entry for every complex in the table.

- **cpx**

Mandatory column giving a VL_1 declaration of the complex. A complex is presented as a series of selectors. Selectors may be separated by any amount of white space or new lines. A new complex is started only when a new entry for the # column (i.e. a number) is found. Each selector is an expression of the form [variable = values]. The brackets are mandatory. The variable may be any variable declared in the variables table, but the same variable may not appear twice in one complex. The values must be defined values for the variable given on the left of the "=" sign. Several values may be specified in one selector in any of the following forms:

$$value_1,value_2$$

$$value_1..value_2 \text{ (valid only for linear and cyclic variables)}$$

$$value_1..value_2,value_4..value_5 \text{ (also valid only for linear and cyclic variables)}$$

The symbol "," in a selector means "or" and the symbols ".." specify a range of acceptable values. So, the selector [color = blue,green] is read "color is blue or green." The selector [size = 0..3] means "size is between 0 and 3, inclusive."

Below is an example of a -outhypo table which uses the variables defined in the example in part 2.5. This rule would be used as an initial hypothesis for the class "ONE". The rule consists of two complexes, and is read " If long and wide are false or size is 1 then the event is of class ONE."

ONE-outhypo

#	cpx
1	[long = false][wide = false]
2	[size = 1]

2.7 The -test tables

These tables are used to input testing events to ATEST. They are identical (except in table name) to the -events tables in GEM. The specific name given to a -test table corresponds to a single decision class. In ATEST, if the specific table name has not been seen previously (in a -children table, see below), then a new class is created at the top of the rule base structure.

The column headers for this table type consist of variable names defined in the variables table. The values in the rows of the table must be legal values for the appropriate variables. Since many attributes may be used to describe an event, it is possible to split a -test table into several tables. This is

done by repeating the table name (both specific and general), and using different column headings in each occurence. Column headings may not overlap, and each table must have the same number of events.

The -test table shown below uses the attributes defined in the example in part 2.5. This table would associate four testing events with the class "ONE".

ONE-test

long	wide	color	size
false	false	blue	0
true	false	red	1
false	false	red	0
false	false	blue	1

2.8 The -children tables

ATEST accepts -children tables in order to define a structuring on a rule base. A -children table specifies the children of a class in the rule base. The specific name of the table must be the name of an already defined class, i.e. the name must have appeared as the name of a -outhypo table. There will be one -children table for every class that has subclasses. The rule base may be structured to an arbitrary depth. The range of structures allowed is a slight generalization of tree structuring wherein a child may have have more than one parent class. No recursion is allowed.

The -children table consists of two columns:

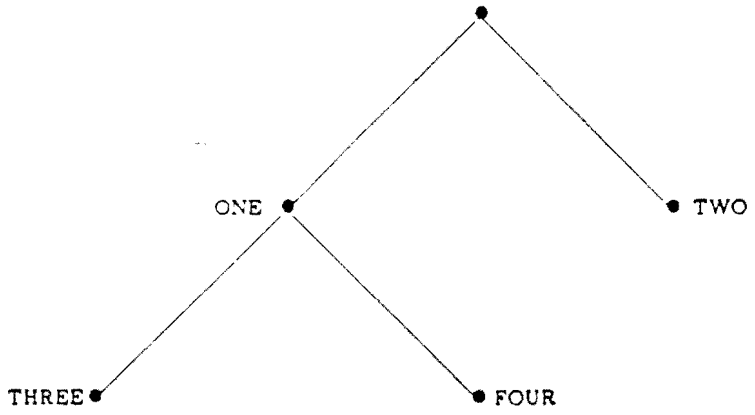
- **#**

  The number of the row. The first row must be numbered "1," and the rows must be numbered sequentially.

- **node**

  This column is an alphanumeric string of up to twenty characters giving the name of the node to be defined.

The tree below shows how a sample rule base might be structured. In this rule base, ONE and TWO are siblings at the top of the structure. The class ONE has two sub-classes, THREE and FOUR.



The tables below define the rule base structure given in the tree. Again, the variables defined in part 2.5 are used. The -children table is only used for rules that are not at the top of the rule base structure. Unlike the GEM -children table, events are not associated with a child node in this table. Note that the -outhypo tables for the child nodes are not entered until after the nodes are defined in the -children table.

ONE-outhypo

#	cpx
1	[long = false][wide = false]

TWO-outhypo

#	cpx
1	[long = true][size = 3]

ONE-children

#	node
1	THREE
2	FOUR

THREE-outhypo

#	cpx
1	[size = 0..1]

FOUR-outhypo

#	cpx
1	[size = 2]

2.9 An Example of Input to ATEST

This subsection contains a detailed example of input to the ATEST program. The sample input file shown in the left column below uses every table ATEST accepts. Comments in the right column explain the corresponding table and provide pointers to the previous subsections for more detailed explanations. To input this example to ATEST, the tables would be entered (in the order given) into some file using a standard text editor. If this file was called "atest.input," then executing:

atest < atest.input > atest.output

under the UNIX shell would cause ATEST to run the example and save the results in the file "atest.output."

ATEST Input — *Comments*

```
title
#       text
1       "Sample input file"
```

The title table is used only for reference and may be omitted.

```
parameters
run   echo  test   misclass  andtype  ortype
1     t     yes    yes       average  maximum
```

The parameters table defines the way in which ATEST will run. This table tells ATEST to echo only the title table, report on the evaluation of misclassified events, evaluate "and" as average and evaluate "or" as maximum. The parameters are defined in detail on pages 78-80.

```
domaintypes
name    type    levels
sizes   lin     3
colors  nom     4
nums    lin     10
```

Three types of variables with 3, 4, and 10 values, respectively, will be used. The definition of the domaintypes table is on page 80.

```
sizes-names
value   name
0       small
1       medium
2       large
```

Names to be associated with values of a variable of type "sizes." The -names table is explained on page 82.

ATEST Input

Comments

colors-names

value	name
0	red
1	blue
2	green
3	yellow

variables

#	name
1	size.sizes
2	color.colors
3	num1.nums
4	num2.num

The variables that will used to describe events. If the domaintypes table was not being used, the *levels* and *type* columns would be in this table. The variables table is defined on page 81.

ONE-outhypo

#	cpx
1	[size = small..medium][color = red,blue] [num1 = 0..2,4]

The input rule for class ONE. This rule will be tested on all testing events. This rules contains a single complex; complexes are separated by entries in the "#" column, as in the next table. See pages 83-84 for more details on the outhypo table.

TWO-outhypo

#	cpx
1	[num1 = 7..9]
2	[size = large]

The input rule for class TWO.

ONE-children

#	node
1	THREE
2	FOUR

The children of class ONE in the structured rule base. Here, class ONE has two children, THREE and FOUR. See page 85 for an explanation of the -children table.

THREE-outhypo

#	cpx
1	[size = small]

The input rule for class THREE.

ATEST Input

Comments

```
FOUR-outhypo
#    cpx
1    [size = medium]
```

The input rule for class FOUR.

```
TWO-test
size        color    num1    num2
medium      blue     0       0
large       green    8       4
```

The testing events of class TWO. These events will be applied to all the rules and the results reported. Note that only leaf nodes in the rule base structure can have testing events (there is no -test table for class ONE). The -test table is defined on page 84.

```
THREE-test
size        color    num1    num2
small       blue     0       3
small       red      3       3
```

The testing events of class THREE.

```
FOUR-test
size        color    num1    num2
medium      red      0       4
medium      blue     2       0
```

The testing events of class FOUR.

2.10 An Example of Output from ATEST

If ATEST is given the input in part 2.9, it will produce output consisting of two parts: an echo of certain input tables (as per the echo parameter), and a summary of the results of testing. This output is shown below.

ATEST Output

Comments

```
title
#       text
1       "Sample input file"
```

The title table is the only table echoed because the value of the echo parameter in the input file was "t."

ATEST Output

Comments

TEST RESULTS FOR CLASS THREE

CORRECT DECISION CLASS = D1 (class THREE)

EVENT	#TIES	ASSIGNED DECISION D1	D2	D3
THREE-1		*1.00*	0.00	0.00
THREE-2		*1.00*	0.00	0.00
# 1st RANK EVENTS		2	0	0

NUMBER OF EVENTS SATISFYING CORRECT RULE : 2

The confusion matrix shows how the rules performed when evaluated on testing events of class THREE. The numbers in the matrix are degrees of consonance. Numbers surrounded by asterisks are cases where the correct rule (in this matrix, the rule for class THREE) evaluated to a first rank decision. The #TIES column specifies the number of first rank decisions for the event if there were more than one. See section 4.2.1 for definitions of terms.

TEST RESULTS FOR CLASS FOUR

CORRECT DECISION CLASS = D2 (class FOUR)

EVENT	#TIES	ASSIGNED DECISION D1	D2	D3
FOUR-1		0.00	*1.00*	0.00
FOUR-2		0.00	*1.00*	0.00
# 1st RANK EVENTS		0	2	0

NUMBER OF EVENTS SATISFYING CORRECT RULE : 2

This confusion matrix shows how the rules performed when evaluated on testing events of class FOUR.

TEST RESULTS FOR CLASS TWO

CORRECT DECISION CLASS = D3 (class TWO)

EVENT	#TIES	ASSIGNED DECISION D1	D2	D3
TWO-1		0.00	1.00	0.00
TWO-2		0.00	0.00	*1.00*
# 1st RANK EVENTS		0	1	1

NUMBER OF EVENTS SATISFYING CORRECT RULE : 1

This confusion matrix shows how the rules performed when evaluated on testing events of class TWO. Note that the event TWO-1 satisfied the rule for class FOUR instead of the rule for class TWO.

ATEST Output

Comments

The testing event

TWO-1 : [size = medium][color = blue]
[num1 = 0][num2 = 0]

was evaluated as follows:

Rule ONE : Degree of consonance = 1.00
cpx
1 [size=small,medium][color=red,blue][num1=0..4]

Rule THREE : Degree of consonance = 1.00 x 0.00 = 0.00
cpx
1 ??[size = small]??

Rule FOUR : Degree of consonance = 1.00 x 1.00 = 1.00
cpx
1 [size = medium]

Rule TWO : Degree of consonance = 0.00
cpx
1 ??[num1 = 7..9]??
2 ??[size = large]??

This part of the output is reporting a trace of evaluation for an event that was not evaluated correctly. The selectors surrounded with question marks are those which were not satisfied by the testing event. Note that the degree of consonance for a child node is the product of its dc and that of its parent. See section 4.2.1 for a detailed explanation of the terminology.

SUMMARY OF TEST RESULTS

OVERALL % CORRECT : 83.33
OVERALL % CORRECT 1ST RANK : 83.33
OVERALL % CORRECT ONLY CHOICE : 83.33

This portion of the output is the only part reported if the test parameter is set to "sum" (see page 79). The first number gives the percentage of *all* testing events for which the correct rule was above the value set by the threshold parameter. The second number is the percentage of all testing events for which the correct rule was a first rank decision. The last number is the percentage of events for which the correct rule was the only first rank decision.

This run used (milliseconds of CPU time)
CPU user time : 367
CPU system time : 184

The amount of CPU time used in testing the rules.

REFERENCES

Baim, P.W., "The PROMISE Method for Selecting Most Relevant Attributes for Inductive Learning Systems," Report No. UIUCDCS-F-82-898, Department of Computer Science, University of Illinois, 1982.

Boulanger, A.B., "The Expert System Plant/CD: A Case Study in Applying the General Purpose Inference System ADVISE to Predicting Black Cutworm Damage in Corn," Report No. UIUCDCS-R-83-1134, Department of Compute Science, University of Illinois, 1983.

Bramer, M.A., "A Survey and Critical Review of Expert Systems Research," *Introductory Readings in Expert Systems*, Michie, D. (Ed.), pp. 3-29, Gordon and Breach, New York, 1982.

Buchanan, R.O., "Partial Bibliography of Work on Expert Systems," Report No. HPP-82-30, Heuristic Programming Project, Stanford University, 1982.

Davis, R., "Applications of Meta-level Knowledge to the Construction, Maintenance and Use of Large Knowledge Bases," Report No. STAN-CS-76-552, Department of Computer Science, Stanford University, 1976.

Feigenbaum, E.A., "The Art of Artificial Intelligence," *Fifth International Joint Conference on Artificial Intelligence*, pp. 1014-1029, 1977.

Forgy, C. and McDermott, J., "OPS, a Domain-Independent Production System Language," *Fifth International Joint Conference on Artificial Intelligence*, pp. 933-934, 1977.

Gaschnig, J., "Prospector: An Expert System for Mineral Exploration," *Introductory Readings in Expert Systems*, Michie, D. (Ed.), pp. 47-64, Gordon and Breach, New York, 1982.

Hayes-Roth, F., Gorlin, D., Rosenschein, S., Sowizral, H. and Waterman, P., "Rationale and Motivation for ROSIE," Technical note N-1648-ARPA, Rand Corp., Santa Monica, CA, 1981.

Hoff, W., Michalski, R.S. and Stepp, R.E., "INDUCE-2: A Program for Learning Structural Descriptions from Examples," Report No. UIUCDCS-F-83-904, Department of Computer Science, University of Illinois, 1983.

Lewis, P., "Taxonomy and Ecology of *Stenonema* Mayflies (Heptageniidae:Ephemeroptera)," Environmental Protection Agency Report No. EPA-670A-74-006, 1974.

McDermott, D.V., "R1: A Rule-based Configurator of Computer Systems," *Artificial Intelligence*, Vol. 19, No. 1, pp. 39-88, 1982.

van Melle, W., "A Domain Independent System that Aids in Constructing Consultation Programs," Report No. STAN-CS-80-20, Department of Computer Science, Stanford University, 1980.

Michalski, R.S., "Discovering Classification Rules Using Variable-Valued Logic System VL_1," *Third International Joint Conference on Artificial Intelligence*, pp. 162-172, 1973.

Michalski, R.S., "Synthesis of Optimal and Quasi-Optimal Variable-Valued Logic Formulas," *Proc. of the 1975 International Symposium on Multiple Valued Logic*, pp. 76-87, 1975.

Michalski, R.S., "A System of Programs for Computer-Aided Induction: A Summary," *Fifth International Joint Conference on Artificial Intelligence*, pp. 319-321, 1977.

Michalski, R.S., "A Theory and Methodology of Inductive Learning," *Machine Learning*, Michalski, R.S., Carbonell, J. and Mitchell, T. (Eds.), pp. 83-134, Tioga, Palo Alto, CA, 1983.

Michalski, R.S. and Baskin, A.B., "Integrating Multiple Knowledge Representations and Learning Capabilities in an Expert System: the ADVISE System," *Eighth International Joint Conference on Artificial Intelligence*, pp. 256-258, 1983.

Michalski, R.S., Baskin, A.B., Seyler, M.R., Boulanger, A.B., "A Technical Description of the ADVISE Meta-expert System," Internal Report, Intelligent Systems Group, Department of Computer Science, University of Illinois, 1984.

Michalski, R.S. and Chilausky, R.L., "Learning by Being Told and Learning from Examples: an Experimental Comparison of 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*, Vol. 4, No. 2, pp. 125-160, 1980.

Michalski, R.S., Davis, J.H., Bisht, V.S. and Sinclair, J.B., "A Computer-Based Advisory System for Diagnosing Soybean Diseases in Illinois," *Plant Disease*, April, 1983.

Michalski, R.S. and Larson, J.B., "Selection of Most Representative Training Examples and Incremental Generation of VL_1 Hypotheses: the Underlying Methodology and Description of Programs ESEL and AQ11," Report No. 867, Department of Computer Science, University of Illinois, 1978.

Michalski, R.S. and Stepp, R.E., "Automated Construction of Classifications: Conceptual Clustering versus Numerical Taxonomy," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI Vol. 5, No. 4, pp. 396-410, 1983a.

Michalski, R.S. and Stepp, R.E., "Learning from Observation : Conceptual Clustering," *Machine Learning*, Michalski, R.S., Carbonell, J. and Mitchell, T. (Eds.), pp. 331-363, Tioga, Palo Alto, CA, 1983b.

Michie, D., "Experiments on the Mechanization of Game Learning," *Computer Journal*, Vol. 25, No. 1, pp. 105-112, 1982.

Paterson, A., "An Attempt to Use CLUSTER to Synthesize Humanly Intelligible Subproblems for the KPK chess endgame," Report No. UIUCDCS-R-83-1156, Department of Computer Science, University of Illinois, 1983.

Quinlan, J.R., "Discovering Rules from Large Collections of Examples: A Case Study," *Expert Systems in the Micro-Electronic Age*, Michie, D. (Ed.), pp. 168-201, Edinburgh University Press, Edinburgh, 1979.

Quinlan, J.R., "Learning Efficient Classification Procedures and their Application to Chess End Games," *Machine Learning*, Michalski, R.S., Carbonell, J. and Mitchell, T. (Eds.), pp. 463-481, Tioga, Palo Alto, CA, 1983.

Reinke, R.E., "A Structured Black-to-Win Decision Tree for the Chess Endgame KP vs. KR (pa7)," Internal Report, Intelligent Systems Group, Department of Computer Science, University of Illinois, 1982.

Reinke, R.E., "PLANT/ds: An Expert System for Diagnosing Soybean Diseases Common in Illinois. User's Guide and Program Description," Report No. UIUCDCS-F-83-911, Department of Computer Science, University of Illinois, 1983.

Rodewald, L.E., "BABY: An Expert System for Patient Monitoring in a Newborn Intensive Care Unit," M.S. Thesis, Department of Computer Science, University of Illinois, 1984.

Shapiro, A. and Niblett, T., "Automatic Induction of Classification Rules for a Chess Endgame," *Advances in Computer Chess*, Clarke, M.R. (Ed.), Edinburgh University Press, Edinburgh, 1982.

Shortliffe, E.H., Scott, A.C., Bischoff, M.B., Campbell, A.B., van Melle, W. and Jacobs, C.D., "ONCONCIN: An Expert System for Oncology Protocol Management," *Seventh International Joint Conference on Artificial Intelligence*, pp. 876-881, 1981.

Spackman, K.A., "QUIN: Integration of Inferential Operators within a Relational Data Base," Report No. UIUCDCS-F-83-917, Department of Computer Science, University of Illinois, 1983.

Stepp, R.E., "Learning Without Negative Examples via Variable-Valued Logic Characterizations: The Uniclass Inductive Program AQ7UNI," Report No. 982, Department of Computer Science, University of Illinois, 1979.

Suwa, W., Scott, A.C. and Shortliffe, E.H., "An Approach to Verifying Completeness and Consistency in a Rule-Based Expert System," *AI Magazine*, pp. 16-21, Fall 1982.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-F-84-921	2.	3. Recipient's Accession No.

4. Title and Subtitle

Knowledge Acquisition and Refinement Tools for the ADVISE META-EXPERT System

5. Report Date: July 1984

6.

7. Author(s)

Robert Eugene Reinke

8. Performing Organization Rept. No.

9. Performing Organization Name and Address

Department of Computer Science
Unviersity of Illinois
Urbana, IL 61801

10. Project/Task/Work Unit No.

11. Contract/Grant No.

No. 000 14-82-K-0186

12. Sponsoring Organization Name and Address

Office of Naval Research
Arlington, VA

13. Type of Report & Period Covered

14.

15. Supplementary Notes

16. Abstracts

This paper presents a paradigm for rule acquisition and refinement in expert systems. Under this paradigm, a knowledge acquisition system should be able to learn rules from examples and provide a domain expert with an efficient means for testing rules. Two computer programs designed to fulfill these roles are described: GEM, for automated inductive inference and ATEST, for automated testing of rules.

To test the feasibility of the paradigm and the effects of various parameter settings, GEM and ATEST were applied to three application domains of varying complexity. The results show the programs to be useful in all three areas, and reveal some of the factors which should be considered when choosing learning and testing criteria.

17. Key Words and Document Analysis. 17a. Descriptors

Inductive Inference
Knowledge Acquisition
Expert Systems
Knowledge Base Testing
Learning

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 99
	20. Security Class (This Page UNCLASSIFIED	22. Price