Designing a Testing Strategy for Expert Systems

by

Lee Anne Hite

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

APPROVED:

David P. Miller, Chairman

J.A.N. Lee

Osman Balci

February, 1988

Blacksburg, Virginia

Designing a Testing Strategy for Expert Systems

by

Lee Anne Hite

David P. Miller, Chairman

Computer Science

(ABSTRACT)

Testing programs with tractable algorithms is one area in which software engineers have made numerous advances over the past few decades. Testing rule-based expert systems, however, is a new area in software engineering which requires new testing techniques.

For the most part, traditional software engineering testing strategies assume modular program development. This assumption is impractical to make for expert system development, for the knowledge base of an expert system is quite simply a huge non-modular program. It consists almost entirely of non-ordered, multi-branching decision statements. In traditional programming, the module interfaces are limited and well defined. For rule-based expert systems, the interaction among rules is combinatoric and highly data-driven. Thus, the testing of a completed expert system via traditional path analysis is impractical.

The design of a testing strategy for expert systems focuses on the generic phases of expert system development. Briefly, these phases include system definition, incremental

system implementation, and system maintenance. Using this simplified breakdown of the expert system development process as a guide, certain testing techniques can be generalized enough to work for any expert system application.

ACKNOWLEDGEMENTS

My co-workers, supervisors, and friends at Educational Technologies have shown me so much patience and understanding. I want them to know that they have helped a great deal. I would especially like to thank Lynn Cunningham for being there in the final hours.

My advisor and friend, David Miller, is probably most to blame for this thesis actually being written. To him I say, "Thanks, Dave." I do not regret the time I spent wandering through various AI fields before settling on this topic. In fact, I am grateful that he gave me the opportunity to do so. I would also like to thank my other committee members, J.A.N. Lee and Osman Balci, for their helpful comments and contributions to this research.

I would like to express my deepest gratitude to those people closest to my heart, my parents and my husband. My parents, Larry and Twila Wilson, have given me their unconditional love and support throughout my life. I have always known I could depend on their advice when I needed it.

I can only begin to acknowledge the help and encouragement I have received from my husband, Richard Hite. I doubt he will ever fully know how much his love, support, and understanding has meant to me.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

1.0 INTRODUCTION

Expert systems contain the knowledge (in the form of rules of thumb or heuristics) and the inference procedures needed to solve domain-specific problems. The intricacy of the domain itself rules out the use of traditional programming techniques and precludes testing by traditional means.

Testing programs with tractable algorithms is one area in which software engineers have made numerous advances over the past few decades. Testing rule-based systems, however, is new and requires some different techniques. Such systems achieve reasonable performance by using inference heuristics for pruning the search space. These heuristics are dependent on individual rules stored in the domain knowledge base.

For the most part, traditional software engineering testing strategies assume modular program development. This assumption is impractical to make for expert system development, for the knowledge base is quite simply a huge non-modular program. It consists almost entirely of non-ordered, multi-branching decision statements. In traditional programming, the module interfaces are limited and well defined. For rule-based expert systems, the interaction among rules is combinatoric and highly data-driven. Thus, the testing of a completed expert system via traditional path analysis is impractical.

The design of a new testing strategy for rule-based expert systems focuses on the generic phases of expert system development. Briefly, these phases include system definition, incremental system implementation, and system maintenance. Using this simplified breakdown of the expert system development process as a guide, certain testing techniques can be applied to each of the phases. These techniques can be generalized enough to work for any rule-based expert system application, and they evaluate those attributes of software quality which are desirable in any system.

1.1 MOTIVATION

"Water, water everywhere, nor any drop to drink" (Ode to France, Samuel Taylor Coleridge 1772-1834).

Expert systems, like drops of water in the oceans, are all around us, but are they usable or reliable? How can we be sure? The only answer is to test, re-test, and test them again.

Testing, however, costs money in the real world and should not be taken lightly. Poorly planned testing strategies can be as expensive as good ones, and they do not necessarily deliver reliable systems to the consumer. These unreliable systems must either be corrected or re-built at some point, requiring both time and money (which might have been saved

if the original testing strategy had caught the problems during development).

1.1.1 Prevalence of Expert Systems

The successes of the first expert systems, as well as the difficulties involved in producing them, did not go unnoticed. Tools to aid the development of expert systems are being produced almost daily [Weiss 1984]. By the year 2000, some predict that the widespread use of knowledge-based systems will become very evident in the industrial realm [Harmon 1985].

The widespread applicability of expert system programming techniques is leading to their increasing prevalence in the general software industry. By offering "artificially intelligent" or "expert" modules with their normal products, software vendors hope to out-sell their competitors. Whether researchers in the field of Artificial Intelligence (AI) consider these modules to be true "AI" or not, these techniques are finding their way into mainstream software.

Expert systems are making their mark especially in the military software arena. As more and more defense systems become automated, the desire to use expert systems increases as well. However, their inclusion in military decision support systems (DSS) is not widely accepted even though other traditional programming methods are not capable of

handling the job [Parnas 1985; Akey 1987]. A contractor delivering an expert decision support system must first of all be sure that the application is amenable to current expert system technology. Otherwise, he will not be able to deliver a reliable product.

1.1.2 Economics of Testing

Typically, general software testing requires a large part of a software development and implementation budget [Berrill et al. 1985]. Depending upon the complexity of the code and the cost of letting even one slight error get by, testing may (or may not) be an expensive, time-consuming part of software development.

The cost of not finding and leaving an error in a program may indeed be very high. Consider only the significance of erroneous output from a military DSS, and the cost may itself be incalculable. Even in business or industry applications, one seemingly insignificant inconsistency in a program may, in time, cause enormous financial loss. The importance of software testing, though dependent upon the importance of the software itself, cannot be disputed.

1.1.3 Confidence in Experts

Human experts generally have some training, which may be either formal or experiential, in their field. They are often tested in some fashion, and many receive diplomas, licenses, and certifications as proof of having passed these tests. Once licensed, a human expert is generally treated with respect by his clients until his expertise is disputed. At this point, the de-licensing of an expert can be a complex procedure in just about any field.

Expert systems are subjected to a slightly different regimen. Before being let loose upon the public, an expert system, like a human expert, must pass some testing standard. But an expert system never receives a license to distribute advice. User confidence in the expert system comes initially in the ability of the testing routines to catch errors. Inconsistencies found in the program start to cast doubt in the minds of the users. Once their confidence in the system is shaken, their acceptance of the product will also waver.

1.2 OPEN AREAS OF RESEARCH

The increasing interest in expert system development has opened several challenging research issues. Each one is deserving of concentrated research and more thorough

development. These issues include (but certainly are not limited to):

- knowledge acquisition and representation,
- truth maintenance,
- development of shells, languages, and tools,
- development of a formal lifecycle model, and
- development of formal testing strategies.

None of these issues is completely segregated from the others. In as much as is possible they are individually and briefly discussed below.

1.2.1 Knowledge Acquisition and Representation

Knowledge acquisition involves extracting domain knowledge from the content expert. It is usually performed by a trained knowledge engineer whose job requires him to know just which questions to ask and how to encode the responses. The representational form is usually dependent upon the application domain, and it is chosen early in the development process.

1.2.2 Truth Maintenance

Truth maintenance provides a way of ensuring the integrity of the knowledge base. Rules in the knowledge base should be neither contradictory not redundant. Contradictory rules ruin the reliability of the system; redundant ones make it difficult to modify.

1.2.3 Shells, Languages, and Tools

Many existing expert system programming environments can provide a solid basis for a development effort. These environments usually consist of some combination of expert system shells, languages, and development tools. If a particular environment fits an application, it is certainly easier for the developer to use that one rather than developing one on his own.

1.2.4 Lifecycle Model

There is no dispute in related literature that expert systems can be extremely difficult, as well as expensive, to develop. Traditional software lifecycles do not fit the special needs of the development of these systems. Alternative lifecycle models that allow for incremental software development must be used to meet these needs.

However, both traditional software and expert system lifecycle models tend to contain the same main stages of development. These generic phases are: definition, development, and maintenance. These phases are further discussed in later chapters.

1.2.5 Testing Techniques

Some verification and validation techniques used with other types of software either do not apply to or can not be applied to the development of knowledge-based expert systems. For example, exhaustive path testing is generally ruled out for expert systems because of the intricacies of the structure of the knowledge base. Currently, testing techniques for expert systems have not been standardized, and there are no formal means for evaluating existing or future systems.

1.3 EXPERT SYSTEM TESTING: AN OVERVIEW OF EXISTING PROBLEMS

In past years, Artificial Intelligence (AI) systems have typically been housed in the confines of academic research labs, away from the rigors of traditional software engineering practices. Programming practices in these labs consist mainly of run-debug-edit loops as developers constantly tune programs to produce desired results.

Experimentation with new ideas and implementational issues has been the prime motivation for working in this area; and such experimentation has only encouraged the development of new programming techniques. With new marketing interests and contracts for AI systems, however, a need for testable, valid end-products has arisen.

Using a simplified breakdown of the expert system development lifecycle, there are certain "tests" that can (and should) be performed at each stage. These "tests" can be generalized enough to work for any expert system application, and they will fit into the major development stages common to expert system development.

1.3.1 Lack of Standards

Many issues involving "correctness" for an expert system need to be resolved. In the case of an expert system which "predicts" either causes or effects, should the system be judged according to what human experts "predict" or what the actual outcome is? If the latter standard is chosen, then is more expected from the expert system than from its human counter-parts? If the former is chosen, then "wrong" answers from the expert system could realistically be judged as correct (if the human experts give the same answer). Some compromise needs to be made here.

Another problem involving the evaluation of expert systems is that biases of the evaluators need to be taken into consideration. Gaschnig [1983] points out that separate evaluations of MYCIN were extremely different because in one case, the human evaluators knew they were evaluating a computer program, and in the other, they did not. Certainly, these results indicate that blinding the evaluators with Turing tests is necessary. A Turing test, also known as the "imitation game" [Turing 1950], might pit an expert system against a human expert. Then, a third party evaluator judges which is which based on how each answers domain-related questions.

1.4 PREVIEW OF WHAT'S TO COME

Chapter two of this thesis presents a menagerie of background information of software engineering and research on expert system development. Chapter three, entitled "Testing Strategy Design Issues," provides discussions of the major concerns which compose a testing strategy for expert systems. Chapter four follows through from this material by supplying a more detailed analysis of the parts of a testing strategy. Chapter five is an attempt at making concluding remarks and pointing out fertile areas for future research.

2.0 REVIEW OF PREVIOUS WORK

This chapter is broken into two main sections. First, software engineering topics related to expert systems and/or general AI program development is presented. Next follows an overview of general research related specifically to the development of expert systems.

2.1 SOFTWARE ENGINEERING

With the evolution of expert system development in an academic environment, it is no great surprise that the design and development of these systems in a commercial setting is a software engineer's nightmare. In the area of expert system development, there are several genuine software engineering concerns that have only begun to come to light. These concerns include the development and usage of a proper lifecycle model, software testing theories and practices, and other issues. Each of these concerns is outlined below.

2.1.1 Software Lifecycles

Software engineering experts agree that constant monitoring of the entire development of a software system is imperative in determining the reliability or validity of that

system. This monitoring process is usually modeled as a "lifecycle" of software development.

There is no single "best" software lifecycle model. Quite a few of these models exist, and most of them contain the same basic elements. The different known models range from being very general in nature (that is, general problem solving techniques) [Berrill et al. 1985] to being much more specific (with specific software applications in mind) [Overstreet et al. 1986; Giddings 1984].

A Generic Software Lifecycle: For simplicity, a software development lifecycle described by Berrill [1985] is used in this paper. The stages used in this model are very similar to the main lifecycle phases (where each phase includes one or more stages) described in General Electric's Software Engineering Handbook [1986]. The stages of Berrill's model are as follows: inception and definition (specification before implementation), design, production, and acceptance (development work during implementation), and operation and evaluation (general use after implementation). As is true with most software lifecycle models, the interaction and monitoring between the different stages is highly stressed. Figure 1 shows the interactions of the stages of this software development lifecycle.

The first stages of this generic software lifecycle are inception and definition. It is during the inception stage

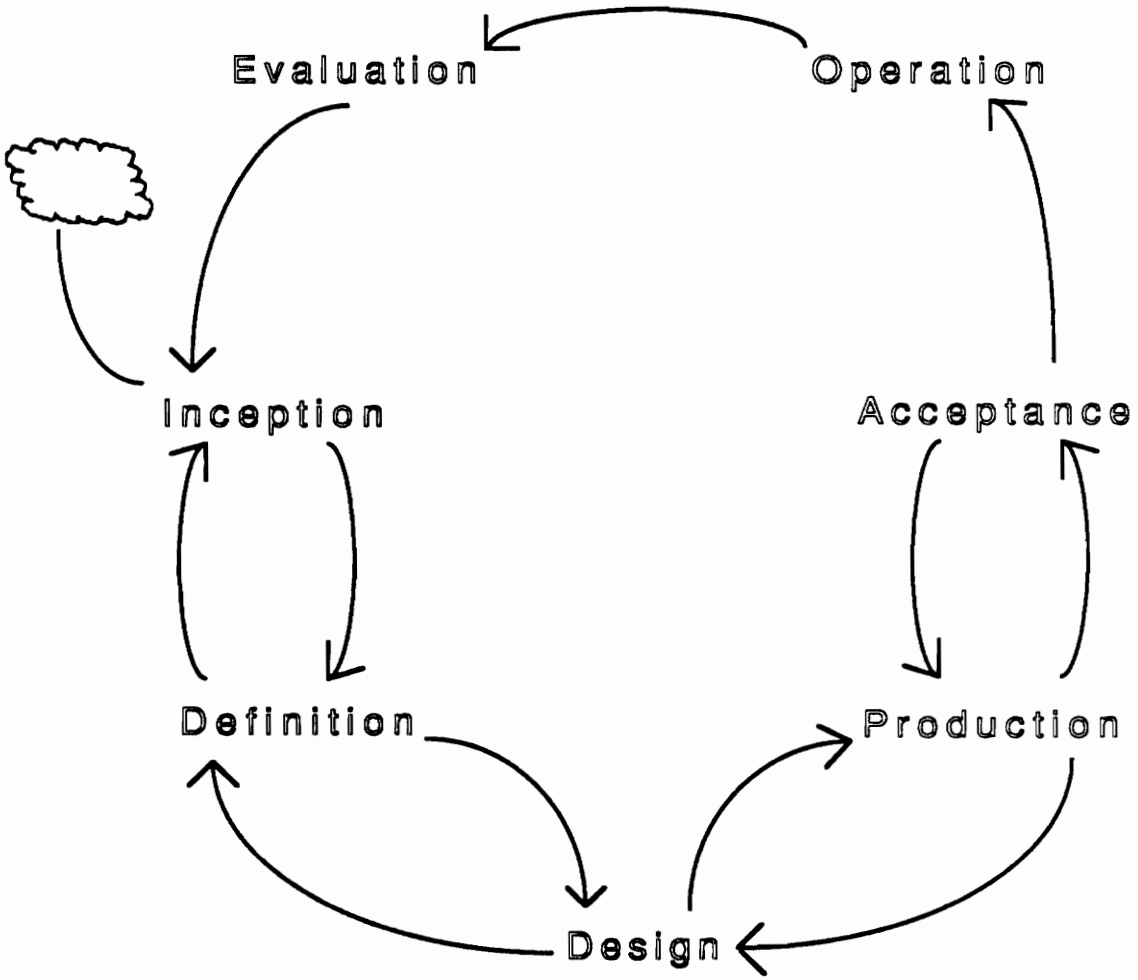

Figure 1. Generic software development lifecycle stages

that the initial request for the software project is expressed, and the first requirements for the system are noted.

The definition stage of the lifecycle is used to specify various software production issues. This stage is most useful later in the lifecycle since the project managers as well as the programmers look to it for the overall production plan.

The system implementation stages typically require the largest amount of the development effort. Project specifics concerning programming issues are established during the design phase. The transformation from system specifications to the actual program must by established during this phase.

The production stage is the actual implementation of the system design. This stage also includes such necessities as the development of user documentation, detailed program modules, subsystem integration, and test modules. After this stage is completed, the software is ready to by tried and tested during the acceptance stage.

The acceptance stage gives the project manager a final chance to find any problems with the software. Typically, a software project will not make it out of this stage without some error being detected. The system must not only be checked for programming faults, but it must also be checked for specification faults. Once a system has been accepted, it is put into operation and is continued to be evaluated for

possible enhancements. At this point, it has entered the post-implementation stages, where it remains during general use.

Backtracking through the lifecycle model is allowed at any time in the lifecycle. Errors that can not be fixed at one level probably have deeper roots and must be traced back through the model.

Sub-cycles to the Generic Lifecycle: The software lifecycle presented above is by no means inclusive of all the possible development phases for a software lifecycle model. Two sub-cycles, which will be investigated in full later, can also be appended to this model. These two sub-cycles are known as prototyping and maintenance cycles and are particularly important to the development of reliable AI software. Figures 2 and 3 show a traditional software development model modified by the inclusion of these two sub-cycles.

The prototyping sub-cycle is best used to explore unknown areas in software applications design. By including this cycle in the overall software lifecycle model, system designers can get a better feel for what the overall design of the project should be without putting a lot of wasted time and effort into the definition stage. The prototype should contain a representative subset of the proposed system specifics. By investigating the problems that arise during

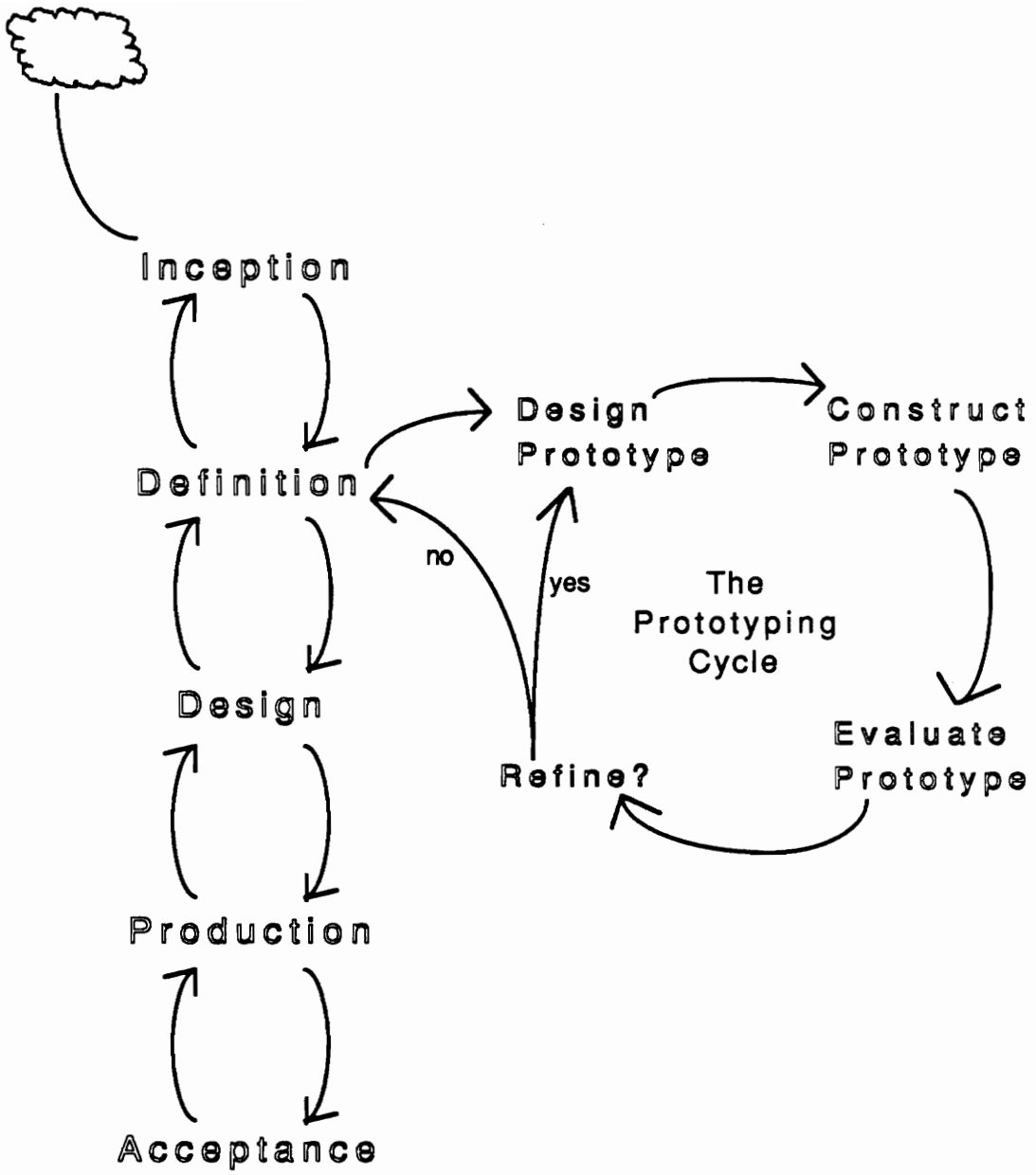

Figure 2. Software development lifecycle with prototyping sub-cycle

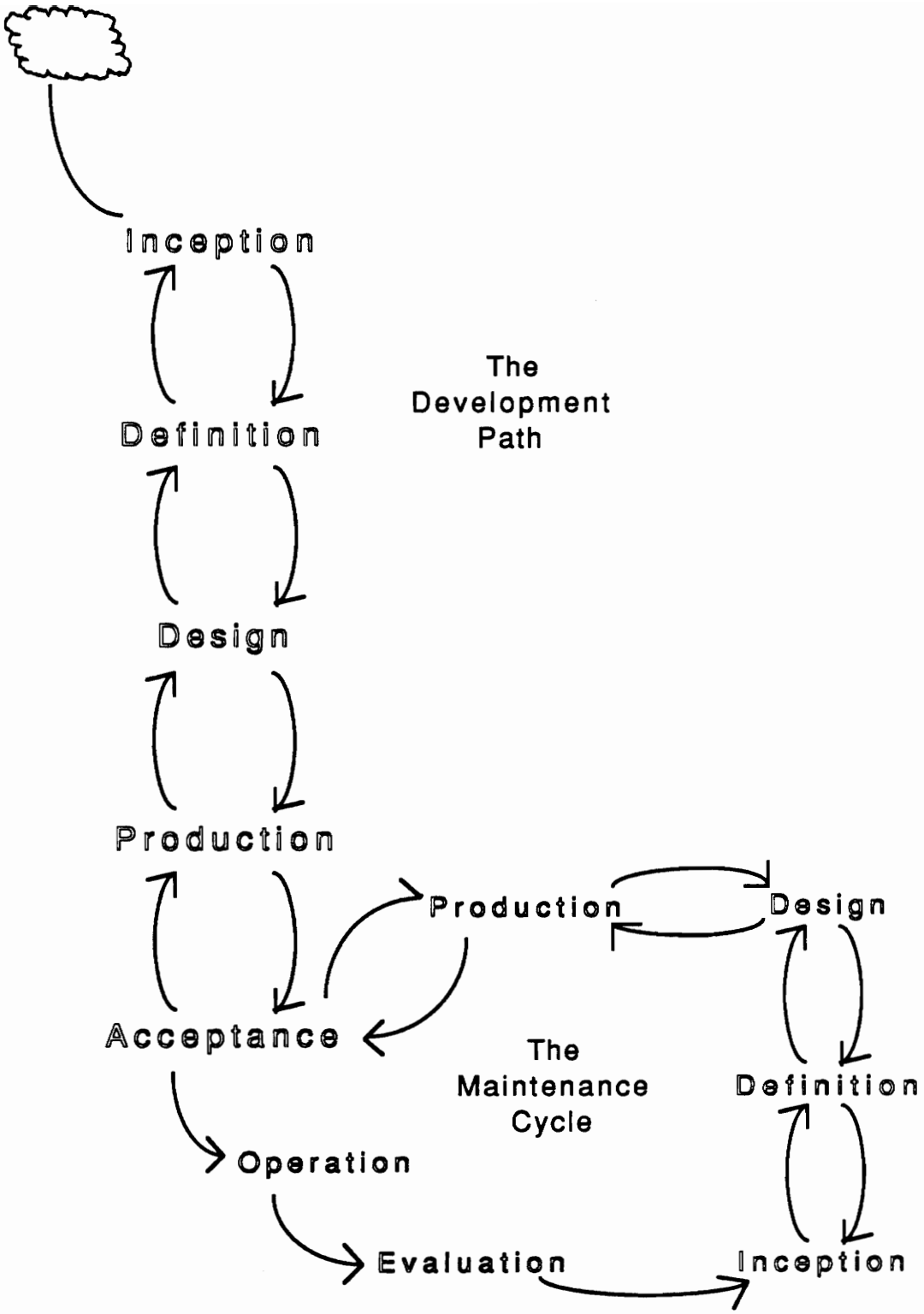

Figure 3. Software development lifecycle with maintenance sub-cycle

the prototyping phase, the designers can get a better feel for what kinds of problems will arise when production of the entire system begins. The prototype can help to indicate what type of programming support will be needed and what programming methodologies will work best.

The maintenance sub-cycle can mean the difference between software longevity and early software obsolescence. Time can change the user's needs, so if a software system is properly maintained, it can change with the user. Also, information stored in knowledge databases needs to be maintained so that it is as current and as up-to-date as possible.

2.1.2 Software Engineering Testing Techniques

Research in software testing has identified two major classifications in the field. The first is known as functional testing, and the second is structural testing. In either case, complete testing is generally not considered to be practical [Beizer 1983]. Software testing is mainly at the modular or "unit" level of a system. In systems that are not modular, however, some of the techniques do not work as well.

Functional testing, also known as "black box" or requirements testing, is applicable to almost any programming domain. With this type of testing, the system is subjected to possible input streams, and the output is evaluated for

correctness, where correctness is defined in the system requirements. Except for very small, limited modules, complete functional testing is impractical because of its size and the amount of time it requires.

Testing with garbage input is just one kind of functional testing. Again, due to the size of the set of possible illegal inputs to a system, data validation may also not be a completed testing area. Input can also be used to check the syntax of a program. Different input streams are used to see if the program responds in an expected manner.

Structural testing is also known as "white box" testing, but it is not completely the antithesis of functional testing. Structural testing techniques examine the internal structuring of a program. With structural testing, the implementation details, including programming and documentation style and control methods used, are evaluated.

Path testing is a major technique used in structural testing. Complete path testing, which could consist of exhaustive programming branch and segment testing, may only be possible on small programming modules. Path testing involves making the program execute every option or combination of options in the code. Because of looping mechanisms and run-time calculations, complete path testing is not generally feasible.

2.2 DEVELOPMENT RESEARCH ON EXPERT SYSTEMS

Even though expert systems and AI development efforts are relative newcomers in the area of computer science research, a substantial amount of research on their development has already been accomplished. For the most part, information on expert system development comes from the actual experiences encountered in academic research labs.

2.2.1 Lifecycle Models for Expert System Development

Because of the large size and lack of modularity of most artificial intelligence systems (including rule-based expert systems), verification and validation methods are not as powerful when applied only to the end product. The size of the system itself does not create all of the problems, for large, modular systems can be integrated in a well-designed approach. But expert systems are not modular, so their size makes testing that much more important. Errors found after the system has been released for general use are much more expensive and difficult to correct than errors found during the development of the same system. For this reason, program evaluation during the entire lifecycle of these systems is very important, so some effort is placed on the development of such a lifecycle model.

The design of rule-based expert systems requires support for knowledge acquisition, knowledge representation, knowledge inference, and search-based computation. The systems should also provide facilities for explanation, probabilistic reasoning, and the development of expert system shells [Subrahmanyam 1985].

Generally, the basic phases of the generic lifecycle model (see section 2.1.1 above) are applicable to almost all types of software development. With AI systems, however, most experts seem to agree that emphasis should be placed on the prototyping and maintenance sub-cycles of this model, for these sub-cycles provide the implementation support needed for expert system development [McCallum 1985; Fickas 1985; Mostow 1985; Francez et al. 1985; Bobrow 1985; Subrahmanyam 1985].

Prototyping: The incremental development of artificial intelligence systems is generally agreed to be the major production methodology. There are several different abstract tools available to the programmer to help him through this stage of system development. A list of these tools includes break/fix/continue program execution cycles, tracing of the program's call stack, sub-system analysis and performance tuning, and stepwise monitoring of program execution [Bobrow 1985].

The prototype model should be designed so that it contains a subset of the knowledge required for the system to operate. The prototype is tested and evaluated; then, the specifications and the design of the system can be re-evaluated based on how well the prototype works. It is an entirely circular process requiring heavy communication between the system expert and the system designer and programmers [Weiss et al. 1984; Birrell et al. 1985].

Prototyping is also deemed to be important in situations where the initial system specifications are ill-defined. The system designers should use this stage in the lifecycle to better develop their specifications of the system and to increase the capabilities of their software [Fickas 1985].

Maintenance: The success of the maintenance phase is almost always a matter of long-life versus system obsolescence. If the maintenance phase is successful, the system remains in use; otherwise, it does not. There are many reasons why any large system should allow for easy maintenance, ranging from financial factors to the easier development of future systems [Birrell et al. 1985].

Several studies have indicated that over time, maintenance processes consume a large portion of the overall software budgets [Birrell et al. 1985]. Therefore, anything that improves the efficiency of this process should also reduce long-term software costs.

In order to aid the addition of new information or the reviewing or testing of existing systems, McCallum [1985] suggests that each bit of "knowledge" (often a rule) should be tagged with the following documentation: source of the information, date that the knowledge was determined, comments on the information, and possible interactions with other information. Unfortunately, this type of rule documentation is expensive and impractical in several ways. In a very large knowledge base, this documentation requires quite a bit of manpower to input as well as electronic storage space. Also, it is impossible to know all rule interactions during the creation of the knowledge base.

2.2.2 Programming Environments

Generally, it is agreed that programming environments which support AI techniques and methods can aid the development of AI software [Bobrow 1985; Subrahmanyam 1985; Francez 1985; Fickas 1985; Teitelman et al. 1981].

Several different AI programming environments have been developed over the past few years. These environments are designed to increase the reliability of AI software and to increase AI programming efficiency. Some of the established systems include Interlisp [Teitelman et al. 1981], ORBS [Fickas 1985], and the Programmer's Apprentice [Waters 1981]. Although these systems are designed with different goals and

procedures in mind, they do have some similarities. For example, most programming environments provide an integrated package including a special-purpose program editor.

The choice of a specific programming environment usually indicates the choice of a programming language also. Programming environments have been designed for both of AI's major programming languages: LISP and PROLOG. For this reason, program specifications need to be considered before selecting a particular programming environment. Some programming environments naturally support software development techniques such as prototyping while others do not [Teitelman et al. 1981]. All of these factors need to be considered together when choosing a language, environment, and model for software development.

2.2.3 Maintaining Expert Systems

Rule-based expert systems consist mostly of the rules and the facts which make up the knowledge base of the system. The consistency and correctness of these rules hold the key to the reliability of the entire system. Rules must interact correctly with each other in order for the system to make correct inferences.

Many things, however, can affect the reliability of a knowledge base. Even if the rules of a knowledge base are proclaimed to be correct immediately after its design and

implementation, the knowledge base can become invalid after it has been in use for any amount of time. Maintenance of an expert system's knowledge base, therefore, is crucial to its longevity.

The major functions required of rule-based system maintenance include the following [McCallum 1985]:

- changing weights on rules due to refinement of knowledge
- replacing or discarding knowledge for various reasons
- updating of time dependent information
- adding new rules
- adding or replacing comments

Tuning rule-based expert systems almost always requires changing the weights of the rules to allow for more exact reasoning. Since these weights may combine in many different ways, they must be as precise as possible. Sometimes, the problem is not with the weight of the rule, but with the rule content. In this case, the content must either be modified or discarded. Rules which contain time-dependent information must be updated as often as necessary so that the system will not make any invalid conclusions based on this information. To allow for the continued success of the expert system

maintenance, the comment fields for each rule must be modified along with the rule.

2.2.4 Expert System Testing

Much work remains to be done if AI software is to become as "reliable" as traditional types of software. The issue of expert system reliability is brought up in more detail in the next chapter.

Techniques applicable to the area of general software testing do not apply in full to AI program testing. The main reason for this is the large size of most AI programs. Also, systems such as rule-based expert systems are not modular in design. Each element in the knowledge base should be independent of all others. It may or may not need to be connected to other elements; there is no way to know until the system is executing. There are no known ways to chart the data patterns the way programs are normally charted (that is, with flowcharts or pseudocode). When some piece of information is changed during a normal updating process, the whole knowledge base may or may not be affected. Currently, there is no way to indicate how one piece of information may affect another without directly examining all of the other rules.

AI systems must be designed for ease of change if they are to last into the future. For this purpose, the design of AI

systems in general and expert systems in particular should be looked into with more detail.

2.3 SUMMARY

The purpose of this chapter has been to provide the minimal background information needed to implement a testing strategy for expert systems. In the next chapter, the major pieces needed to compose this strategy are presented. Later in chapter four, a more detailed picture of this strategy is pursued.

3.0 TESTING STRATEGY DESIGN ISSUES

This chapter details some of the design issues for a testing strategy for expert systems. Included in this chapter are a breakdown of the generic expert system lifecycle, a brief description of what software attributes and qualities are desirable in expert systems and the techniques used to test for them, and an introduction to the issues of standardization of expert system testing.

Each of these issues is used to form the basis for the overall testing strategy which is presented in more detail in the next chapter.

3.1 GENERIC EXPERT SYSTEM LIFECYCLE

The lifecycle models for expert system development contain three generic phases: definition, implementation, and maintenance. These phases are common also to the realm of traditional software engineering [General Electric 1986]. Admittedly, these phases represent a simplification of the existing models, but the point is that they all have these generic parts in common. In different models, these phases contain different stages, and interaction between these may also differ. Figures 4 and 5 show these phases and their probable interaction.

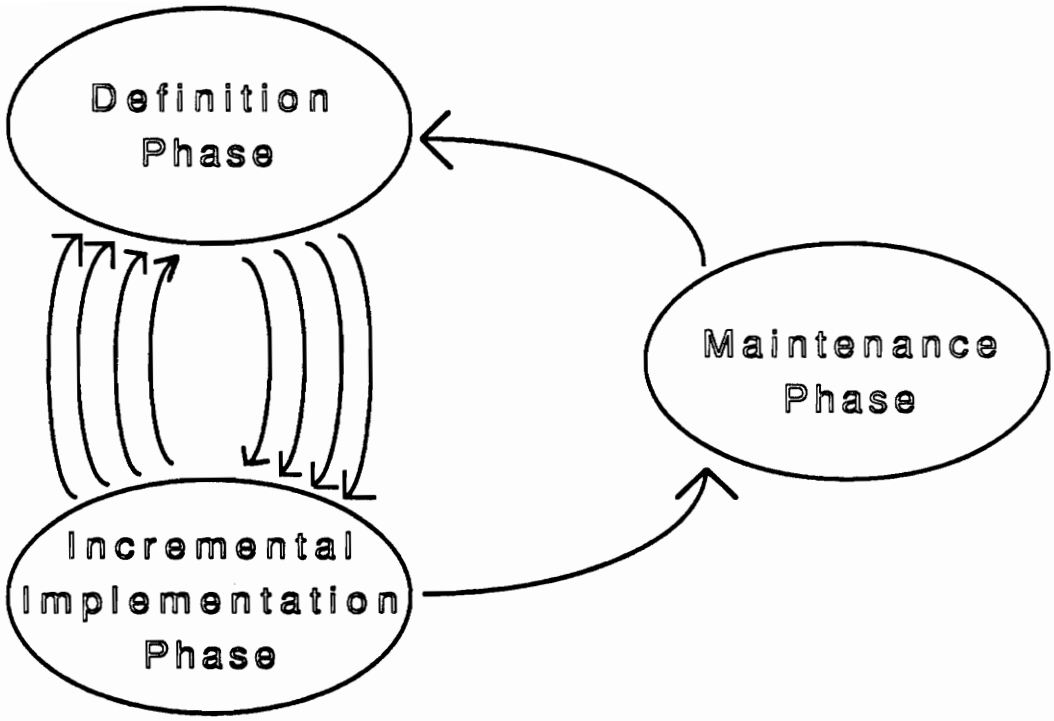

Figure 4. Generic phases of expert system development

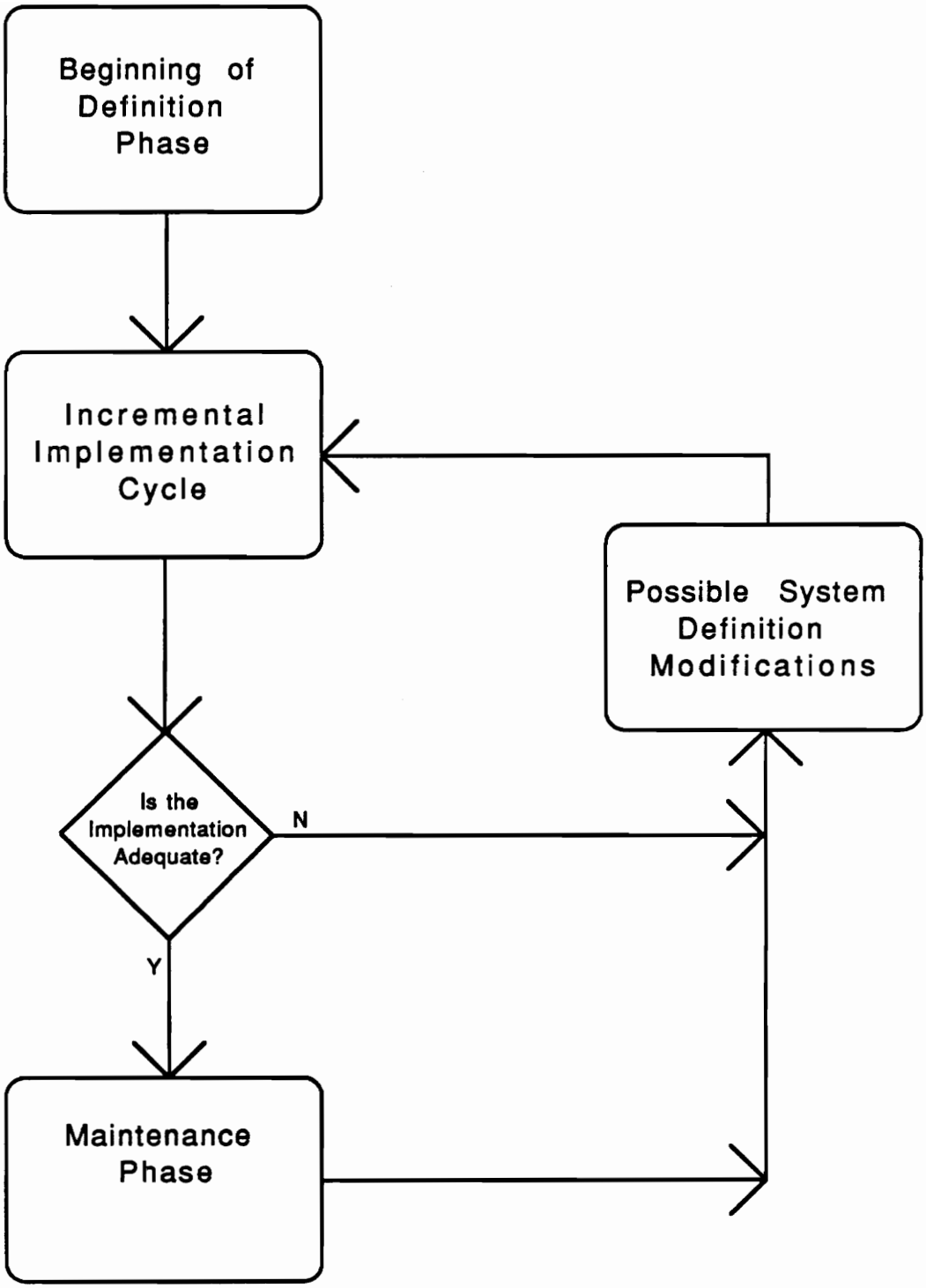

Figure 5. Interactions between phases of expert system development

It is during the definition phase that a general description of the purpose of the project, a top-level design of the system, and a report on the feasibility of its implementation are completed by the expert system developers. Poor initial specifications are the norm in expert system development [Partridge 1986]. Rarely will system developers completely understand the entire application domain in the beginning. Some trial and error is expected before the specifications are adequate. Usually, this stage is not completed until during (or after) the implementation phase.

The implementation phase is the major stage of expert system development. Usually, expert systems are developed incrementally, that is, the process consists of several programming cycles. In theory, each successive cycle comes closer to delivering the final product, the integrated system. During this phase, knowledge acquisition and representation takes place. The knowledge base is created and refined, and informal system tests are used to evaluate it. Also, prototypes of the desired end-product are delivered and tested. Formal and informal evaluations of the prototypes judge the anticipated overall quality of the system before it is released for its intended use.

The standards established in the definition phase are used as a basis for judging the implementation phase. After the successful completion of this middle phase, the expert system enters the maintenance phase. During the system's lifetime,

repetitive evaluations are made to see if the system needs to be updated or modified. If timely information is contained in the knowledge base, chances are that updates are necessary.

Different (and yet similar) versions of this simplified model of expert system development have already gained acceptance and are currently in use [Partridge 1986]. To this extent, there is a "standardized" lifecycle for expert system development. What has not been standardized, however, are methods of evaluation and validation for these systems; there are no formal means of evaluating or testing existing or future systems. There are also no guidelines for programming style or documentation available for expert system developers. Each application seems to have set its own standards.

Using the simplified breakdown of the expert system development lifecycle presented above, there are certain "tests" that can be performed at each of the phases. These tests (or evaluations) can be generalized enough to work for any expert system application. The next chapters of this thesis will describe in detail what these tests are. The generic lifecycle phases will be used as a guide to indicate when a test should be performed. The rest of this chapter presents other issues which must be understood before testing can begin.

3.2 WHAT TO TEST FOR IN AN EXPERT SYSTEM

The list of requirements to test for in an expert system is similar to the list of what to test for during the development of traditional software. The tests described in the following chapters will address these requirements common to any software package.

Peter Sell [1985] claims five basic requirements must be met for an expert system to be valid:

> "... if its pronouncements are free from contradictions,
> if it can tackle any problem within its domain,
> if it can deliver the right answers,
> if the strength of its conviction is commensurate with the data and the knowledge at hand, and
> if it can be used with reasonable facility by those for whom it is intended." (emphasis mine)

In short, these requirements are consistency, completeness, soundness, precision, and usability.

Ten factors for traditional software quality are spelled out in General Electric's Software Engineering Handbook [1986]. Three of these, correctness, reliability, and usability, are similar to Sell's list. Other factors which Sell does not consider include efficiency, maintainability, testability, flexibility, portability, reusability, and interoperability. Depending upon the expert system application and specifications, some of these factors may be more important than others. Nevertheless, expert system developers, like traditional software developers, should not

be exempt from considering all of these factors in the design and implementation of their systems.

Figure 6 provides a brief description of these factors. For the most part, these factors apply to expert systems just as they do to traditional software systems. The main difference between these factors lies in the description of what "correctness" means to an expert system developer. Partridge [1986] proposes that expert systems should not be judged on the traditional meaning of correctness, but rather on their "adequacy" of performing their assigned functions.

3.3 TESTING TECHNIQUES APPROPRIATE FOR EXPERT SYSTEMS

William E. Perry [1983] organizes a list of testing techniques into a dichotomy of structural and functional techniques. Figure 7 shows how some of these techniques are classified. Structural techniques attempt to assure the soundness and the completeness of the design and implementation of the system's structure. Functional testing, on the other hand, is not concerned with how a program works, but rather that is produces appropriate results. Typically, complete structural testing is more difficult to accomplish and requires more extensive test data. Functional testing is generally an easier technique to utilize.

Factors of Software Quality	
Correctness	The extent that it adequately satisfies its objectives
Reliability	The extent that it can perform with adequate precision
Usability	The extent that it can be used to achieve its objectives
Efficiency	The amount of resources (time, money, etc.) required to perform a function
Maintainability	The extent that it can be modified (errors found and fixed)
Testability	The effort required to evaluate its performance
Flexibility	The effort required to make changes to it
Portability	The effort required to transfer it to another hardware/software environment
Reusability	The extent that modules can be re-used in other applications
Interoperability	The effort required to couple one system with another

Figure 6. Descriptions of software quality factors

Testing Techniques	Types of Tests
Structural Determines the soundness of the structure of the system	Stress Testing Execution Testing Recovery Testing Operations Testing Compliance Testing Security Testing
Functional Determines the functionality of the system according to the original system requirements	Requirements Testing Regression Testing Error-handling Testing Manual Support Testing Inter-system Testing Parallel Testing

Figure 7. Classification of testing techniques

3.3.1 Structural Testing Techniques

There are a half a dozen testing techniques which can be classified as structural [Perry 1983]. These include: stress testing, execution testing, recovery testing, operations testing, compliance testing, and security testing.

Stress testing is commonly used for on-line systems, but it is also useful for testing all systems. It determines how the system handles a larger-than-normal work load. Some areas of expert systems which should be stressed include storage space for the knowledge base and an increasing load on the inference engine (Is there a limit to the number of inferences which can be made? Are there any limitations in terms of internal memory or other processing constraints?). If the system is supposed to be an on-line system, what is the maximum number of users which can access the knowledge base at a given time? Although it is a useful technique, stress testing is usually expensive in terms of the amount of time and the resources which it requires.

Execution testing techniques determine the performance efficiency of the production system. This technique should be used early in expert system development to see if the design will meet performance criteria established during the definition phase. For example, response time for user requests and processing time for different types of transactions can be evaluated with execution testing. In an

expert system, each query or transaction may require different amounts of processing time. Execution testing can best be used to estimate the average amount of time a particular search will require.

Recovery testing evaluates what happens to the system if a disaster (either major or minor) should occur. Possible calamities in any system include hardware or operating system crashes, damage to storage areas (the knowledge base for expert systems), or any unexpected problem with the application system itself. Recovery testing techniques should include an evaluation of the built-in recovery procedures as well as the introduction of failures to the system to further evaluate the ability to recover from accidents.

Operations testing assesses how well the system integrates with its intended operating environment. For expert systems, this environment certainly consists of the intended users. Operations testing determines whether existing user documentation, training, and/or support is adequate.

Compliance testing determines whether the development process is in accordance with the chosen (or required) development methodology or not. Compliance testing should occur early in the development process since it becomes more difficult to correct procedural deviations in late stages of the life cycle. Compliance testing makes up a large part of the management team's quality assurance methods.

Security testing evaluates the possibility of resource access by unauthorized users. This type of assurance is very important to expert system use and maintenance. If unauthorized users are allowed to make changes to the global knowledge base, the knowledge base will lose its integrity over time, and any errors could be very costly to find and correct. Security testing is also used to make sure that the tested system stays within memory bounds and does not over-write other applications in memory.

3.3.2 Functional Testing Techniques

There are also six techniques which Perry [1983] classifies as functional testing techniques. These techniques are typically used to determine if a product has achieved the original system requirements and specifications. The individual techniques defined by Perry are: requirements testing, regression testing, error-handling testing, manual support testing, inter-system testing, and parallel testing. Each of these functional techniques is briefly discussed below.

Requirements testing evaluates whether or not the system functions as designated in the system specifications. This type of testing helps to ensure that user requirements are successfully implemented. The process of requirements

testing should be continued throughout all phases of system development.

Regression testing techniques are very important to expert system testing. Regression testing calls for re-testing sections of the system to see if they still function properly after modifications are made to other parts of the system. In an incremental development environment, it is so important to ensure that later development cycles do not undo any of the work which has already been validated and verified. Regression testing should be considered a valuable technique in expert system testing.

Error-handling evaluates whether or not the system can identify and control expected errors. The major difficulty with error-handling testing involves being able to predict what sorts of errors the system is apt to encounter. When a natural language interface is integrated with the expert system, the range of input is almost infinite and so are the possible input errors. This aspect of the natural language interface can sometimes be evaluated apart from the expert system, allowing for some parallel development and testing.

Manual support testing evaluates the processes of preparing input for and using output from a software system. These processes are evaluated in terms of both their adequacy and their execution. Such testing of the user interface should occur throughout the development of the system.

Inter-system testing determines whether interfaces between different application systems or modules functions properly. Many expert systems are coupled with a natural language interface. The reliability of this type of interface or any other in the system is very important to the overall success of the integrated system.

Parallel testing determines whether results from the new version of the system are consistent with old results. This technique is particularly useful when a data format and not a processing operation is modified; this is equivalent to changing the knowledge representational form in an expert system. Parallel testing ensures that the processing or inferential portions of the system still work with the new data format.

3.4 ISSUES OF STANDARDIZATION

There are several issues concerning the standardization of tests or testing theories with expert systems. It would be unrealistic to propose absolute answers to each of these issues. Rather, as each issue is encountered in the description of the tests in later chapters, possible solutions or suggestions will be presented. Some of these issues are briefly introduced and discussed below.

The first (and foremost) issue in the way of standardizing tests for expert systems is how to measure the idea of

"correctness" (or "adequacy") in expert systems. Currently, it is up to the system developers to decide against what standard the program's answers are to be measured. Does the expert system only have to be as "correct" as a human expert? Humans are unique and are not error-free. Also, which expert or experts does one choose as a standard? If human expertise is not "correct" enough, what then? Is the system to be allowed a margin of error smaller than a human's? Does the system just have to be correct 75% of the time? or 99%? or more?

Some expert systems that predict happenings (such as weather forecasters) or diagnose problems (such as some medical expert systems) can be evaluated against reality [Sell 1985] by using known case studies. In this case, human errors do not play a part in the evaluation of the system. The system is fed information on a given situation, and its response is judged against what is already known to be true. In some applications, the expert system must be judged against a human expert's diagnosis of the problem. In these cases, evaluation of the computer system's answer (either as correct or incorrect) can only be based on the assumed reliability of the human expert.

More issues arise when it is time for the system's "peers" (human experts in the same domain) to evaluate the program. Should they be told that it is a computer program giving these answers? Individuals with negative biases towards

computerized systems may be biased against any solutions given by a machine. Gaschnig [1983] indicates that human evaluators should be blinded to the identity of the system. He proposes the use of Turing tests for such a problem. However, if the expert system is expected to perform at better than human standards, it may be that by withholding this information from the evaluators, the system is not properly evaluated.

3.5 TESTING APPROACHES

Before any testing strategy can be implemented, it must be well thought out. In traditional software engineering, two testing approaches are generally used in testing strategies [General Electric 1986]. These are top-down and bottom-up testing.

Top-down testing involves testing the integration of the upper level system drivers and overall logic flow and then adding the other modules to the system, one level at a time. Testing for correct integration continues at each level until the last module checks out. This approach tends to work best when the entire system structure is well-understood before implementation begins.

Bottom-up testing works just the opposite. Modules are individually tested and added to sub-systems. As sub-systems are completed, they are combined until the overall system is

assembled. Therefore, the integration of all the pieces is not an issue until the sub-systems are finished. For this reason, bottom-up testing tends to work best when the sub-systems are better understood than is the entire system.

Neither of these approaches seems directly applicable to expert systems by themselves. The top-down method requires complete knowledge of the existence of individual modules, and the modularization of expert systems is not feasible before implementation of the knowledge base begins. The bottom-up method perhaps places little emphasis on module integration to the entire system for an expert system. A combination of these, however, might work better for expert system testing.

A top-down approach is used up to the point of knowledge acquisition and integration into the knowledge base. The inference engine of the expert system is the overall driver. It should be domain-independent and relatively static, that is, it should not change much if any during the development of the knowledge base. In many cases, a programming environment will supply the inference engine for the developer. If so, it will still need to be evaluated to ensure its appropriateness to the application. Portions of the system related to the inference engine can and should be very well defined. Changes to the overall driver should be resisted once the knowledge integration has begun, so the inference engine needs to be completed early in the process.

Once the integration of individual rules has begun, bottom-up testing can be employed. Since no ordering of rules should be imposed on expert systems, each rule is truly a mini-module all by itself. If an explicit rule ordering is required in a system, the ordering must be very well defined. Modifications to the rule-base can be difficult enough to accomplish without having to worry about where the new rule must be placed. Each rule should be individually tested for syntactic and semantic correctness by someone other than the rule coder. The semantics is especially difficult to test since it involves a translation of information gleaned from the content expert. Weighted rules must also be checked for "appropriateness" of the weights. Also, at this point, the rules are treated individually for the most part, so interactions with other rules is not yet an issue.

This modified approach to expert system testing provides a guideline for the overall testing strategy. In general, the control mechanisms (such as the inference engine) should be evaluated first. Efforts to test the rule base are crippled without a working inference engine.

4.0 REALIZING A TESTING STRATEGY FOR EXPERT SYSTEMS

An overall testing strategy should be firmly embedded in the entire expert system development process. During the definition phase, the testing strategy should be thoroughly documented and then continuously updated during the lifetime of the system. The implementation phase contains the most rigorous application of tests and evaluations. During use and maintenance of the system, similar testing should be applied systematically to assure the continued success of the project.

4.1 STEPS TOWARDS DEVELOPING A TESTING STRATEGY

The overall conception of a complete testing strategy for expert systems comprises of several steps. First, desirable system attributes are identified for each testing phase. Next is the determination of which testing techniques are appropriate to the application. Finally, appropriate testing tools are selected for use in each of the development stages.

4.1.1 Identifying Software Attributes

Individual software attributes were discussed in section 3.2. Once the developer determines which of these attributes

he wants his system to have, he must decide how much time and money he is willing to spend to assure that they are indeed present in the end product. He could rate each attribute on its importance to the project (For example, "On a scale of 1 to 10, how important is 'portability' to the system?"). Software testing is not cheap. It is a necessity consuming both time and money from a development project. If a project manager knows in advance which attributes are more important, then he can allocate the needed resources more efficiently and more effectively.

4.1.2 Determining Testing Techniques

Once the test factors are identified, the next step is to determine which testing technique or techniques are to be used for each factor. Individual testing techniques were discussed in more detail in section 3.3.

The relationship between techniques and tools should be understood before proceeding. The overall objective to keep in mind is to test for certain attributes in a software system. There are several different techniques, both structural and functional, which may be used to achieve each objective. Once appropriate techniques are chosen, the next step is to select tools which are suited to these techniques.

This relationship is analogous to the art of communication. In this case, the objective is to communicate

some message. There are several different techniques, both verbal and non-verbal, which may be employed to reach this objective. Some of these techniques include the written word, spoken word, or signing (verbal), and drawing, motion, or sounds (non-verbal). Communication tools may include pen, paint, yelling, whispering, hands, etc. The catch is to appropriately apply a tool to the chosen technique. For example, paint may (or may not) be an appropriate tool for communicating a written message; it simply depends on the circumstance. In a similar fashion, testing tools must be evaluated for their appropriateness in a given situation. Each tool may (or may not) be applicable to more than one technique.

4.1.3 Choosing Testing Tools

In the final analysis, the actual selection and application of testing tools may be dependent upon the expert system application domain. There are several categories of tests, however, which should be included in any software system. For each of the three main stages of expert system development (system definition, incremental development, and maintenance), a series of testing methods appropriate for that stage are outlined in the remainder of this chapter.

4.2 DEFINITION PHASE TESTING

In the past, the evaluation of specifications for any expert system have not received as much attention as have the other phases of software development [Perry 1983]. More recently, however, software engineers have begun to point out the importance of starting the system development off on the right foot and of setting a solid foundation for the work which is to come. Steps have been made in testing during the system definition phase, but it is not yet a science.

If system definition testing is so difficult for traditional software development, it is even more so for expert system development. For one thing, the systems definition phase does not near completion until the implementation phase is well under way. In some cases, it is not until the system has been implemented that a more thorough understanding of the functions is reached [Partridge 1986]. Nevertheless, if the system definition (including requirements analysis and specifications) is intelligent and well thought-out in the beginning, the overall project cannot help but be improved. In some ways, the performance of this phase is assessed by the facility or difficulty of the next phases.

Expert systems programming generally falls into the category of exploratory programming, and complete system specifications are an impractical requirement at the

project's inception. The definition phase continues alongside the development phase, gleaning new information as it becomes available.

The overall testing strategy should be an outgrowth of the system's specifications. As the project proceeds, system requirements firm up to reflect "practical" goals. "Impractical" (or poorly understood) goals are taken care of by system prototyping. If ways to meet these goals are found, the goals become "practical." Otherwise, development expectations must be modified to represent the reality of current hardware or software technology. If it can not be demonstrated during the early prototypes that initial goals and expectations can be met, and if these goals can not be redefined, then the project has died an early death. At least the system did not have to go all the way through the development phase before gasping its last breath.

The incremental development and refinement of the system definition is known as system sculpture. This model of development "... supports the implementation of applications in which knowledge of the system requirements is refined or generated during the implementation process" [DeMillo et al. 1987]. System design begins with a rough functional model and is molded to reflect heavy interaction with the implementation phase.

If rapid prototyping of system specifications always worked well, there would be fewer headaches involved with the

design of expert systems. Unfortunately, however, many specifications can not be demonstrated via incomplete system prototypes. There is also a risk of giving up on the project too soon. For the most part, early developmental efforts of the system specifications can only be used to guide the rest of the development process.

4.2.1 Attributes for the Definition Phase

The system requirements should be the major influence on the overall system testing strategy. Since the requirements may change as the definition phase reacts to information from the implementation phase, the definition phase must be readily modifiable or maintainable. There is also a need for the system requirements to be usable during the development phase and, at some level, complete (that is, the initial intended domain must be specified). The main goal, however, is for the initial and subsequent designs to be adequate for the intended application environment [Partridge 1986].

These four test factors, adequacy, maintainability, usability, and completeness, are probably the most important ones for the definition phase. Other test factors, however, can also be emphasized depending on the application and the project management.

4.2.2 Testing Techniques for the Definition Phase

Three testing techniques are most appropriate for the definition phase. They are compliance, operations, and requirements testing.

Compliance testing during the definition phase is used to ensure the correct usage of the chosen expert system development methodology. If this methodology supports an incremental development cycle, it probably also includes support for an initially incomplete system definition. Compliance testing is used throughout the expert system's lifecycle to make sure that unnecessary errors are not included in the system.

During the definition phase, operations testing is used to determine the completeness and usability of the requirements. It is also used to evaluate the initial design of the system processes.

While compliance testing ensures that methodological requests are met, requirements testing ensures that user requirements are met. If the user's needs are not addressed during the definition phase of the expert system's development, chances are that they will not be included in the final product.

4.2.3 Evaluating the Definition Phase

Several different testing methods can be used to implement the three techniques suggested above. It seems imperative, however, that a checklist exist containing original and updated methodological and user requisites, as well as those operating functions which have been specified. This checklist can be used by the developer to prepare for:

- desk checks and walkthroughs evaluating the adequacy of the existing specifications in the design,

- peer reviews evaluating the completeness and potential usability of the initial system definitions, and

- design reviews evaluating the methodological compliance.

For the most part, previous experience, error-guessing, and common sense make up the basis for judging these definition phase evaluations. Only proper use of the checklist can render any type of an objective evaluation during this phase. The system definition and design should not contradict or leave out any of the items on this checklist.

4.3 IMPLEMENTATION PHASE TESTING

More emphasis on the actual acts of system testing and evaluation is found during the implementation phase. In practice, expert systems are incrementally developed. Therefore, in theory, the output from each successive cycle should come closer to meeting the requirements established by the definition phase.

Knowledge acquisition and integration of the knowledge into the knowledge base are the major activities of this phase. Domain information is gleaned from human experts and translated into the selected form of knowledge representation. The knowledge base for each expert system application will have different characteristics. In some, rules will be highly connected throughout the knowledge base. In others, they will not. The only factor which affects the connectivity of the information is the domain area. The extent to which the expert system knowledge base can be modularized cannot be predicted before the knowledge acquisition process begins.

The use of expert system technology itself places no constraints on the ordering or the connectivity of the knowledge base (unless an ordering is imposed for reasons of efficiency). In theory, it is not necessary to modularize the information, and indeed, this lack of imposed modularity is one of the benefits of choosing expert system technology

to solve a problem. Rules are considered independent of each other as they are added to the system. Because the knowledge bases are not modular in design, however, expert system evaluators are not able to systematically apply a modular testing strategy when implementing the system.

Incremental development of the expert system is used to provide successive approximations to the system specifications [Partridge 1986]. At each evaluation stage in the cycle, tests are performed to see how close the existing system is to meeting the requirements. Since modular testing, for the most part, is not supported by the design of the knowledge base, the same tests must be repeated throughout the implementation phase. There is no assurance that successful test results will be the same after the knowledge base is changed [Sell 1985; Gaschnig et al. 1983].

The results of the implementation phase are guided by and judged against the specifications of the definition phase. However, with expert system development, like all exploratory programming, the developer should document new-found capabilities or limitations and suggest appropriate changes to the system definition. In this sense, the evolution of the definition phase is, in turn, guided by and judged against the outcomes of the development phase.

4.3.1 Attributes for the Implementation Phase

In theory, the development system should contain all of those software attributes defined in section 3.2. In practice, however, each implementation cycle may only be concerned with a few (or only one) of these attributes at a time. If the system is tested for one factor at a time, it is easier to pin-point the problem area should the test fail [Gaschnig et al. 1983].

During early cycles, more emphasis is placed on correctness (does it perform correctly?), completeness (does it cover the application domain?), and reliability (is it consistent?). In the beginning, these issues are usually paramount to the others. When early system prototypes have shown the feasibility of the processing requisites, then evaluation of the other software attributes can gain momentum. Maintainability, usability, and efficiency are probably the next most important factors for the implementation phase. In any given situation, however, the relative importances of these six factors (correctness, completeness, reliability, maintainability, usability, and efficiency) may change drastically depending on the specific needs of the application.

4.3.2 Testing Techniques for the Implementation Phase

Several functional testing techniques can be used to assess the results of the processing elements. For example, to account for changes in the system from cycle to cycle, parallel and regression testing should be performed.

Error-handling can be used in several different ways. First, erroneous data can be entered into the knowledge base and results of different transactions predicted. This technique can be used to gain insight on how to go about spotting hidden errors in the knowledge base. Error-handling can also be used to determine the completeness of the system, for illegal queries which do not fall within the domain of the application should be recognizable.

Requirements testing is used during the implementation phase to ensure that those functional and user requirements are correctly (or adequately) implemented.

Maintainability of a system can best be determined by compliance testing. If the knowledge engineer complies with project standards, modifications to the system should be easier to accomplish. Again, this type of testing throws much of the responsibilities on the correctness or appropriateness of the development methodology.

System usability can be determined by two different techniques: operations and manual support testing. Operations testing validates that users are able to interact

with the system. Manual support testing verifies that they can do it correctly.

Testing for efficiency can be determined by using execution and stress testing. Execution testing evaluates the normal proficiency of the system. Stress testing, on the other hand, pushes the system to find the extreme processing times.

4.3.3 Evaluating the Implementation Phase

Many different testing methods can be generically applied to implementation phase testing. Some of these are very familiar to traditional software testing, and others are not. If the amount of resources and time available were infinite, testing during this phase could become never-ending. But resources are (almost) never infinite, and neither is the amount of time usually allotted for testing. Each cycle during the incremental development receives its own dose of testing, so it is important that the available tools render good judgement on the current approximation. To this end, several different types of testing methods and tools are suggested below.

Parallel operation of different system versions can be used to determine if new changes to the system have caused any undesirable side-effects (such as delivering the wrong answer!). To perform this test, identical queries are made

to the system during each implementation cycle. The results are then contrasted with each other to ensure the integrity of newer versions of the system. In expert system testing, it is important to process each of the test cases during each testing stage. In this way, if a modification to the system has a strange and unexpected side-effect on one of the queries, the problem has a better chance of being spotted sooner.

During the definition phase, values for acceptance test criteria must be specified. These values can be used during the implementation phase to verify that the system is giving correct (or adequate) answers or not. Specific queries can be obtained from a variety of sources, such as case studies where available. The system must be able to perform up to par on any query which rests in its domain, but there should be an acceptable threshold into which answers can fall. When case studies are not available, face evaluations of system responses can be used. System developers, end users, and content experts can judge the adequacy of the responses, indicating whether they fall within an accepted range or not [O'Keefe et al. 1987].

Boundary value analysis can be used to judge the application domain (completeness) requirements. This testing tool requires the expected domain of the knowledge base to be spelled out during the definition phase.

Error-seeding a knowledge base is a tool used to see if the error-handling capabilities of the system perform as predicted. Specific errors are entered into the knowledge base. Then, system responses are evaluated to see how these errors affect the system queries.

Compliance with syntactic rules can best be judged with the use of compiler-based analysis. Some type of syntactic analysis should be used over the entire knowledge base, even if it is designed only to catch inconsistencies in the knowledge representation form. The time involved with the application and re-application of this type of tool may be high, but it is probably the surest way to maintain the correct formatting of information. Research in the area of knowledge acquisition has produced tools which help provide a semantic analysis of rules. One system, Teiresias, is discussed in section 5.2.1.

As in almost all software applications, judicious use of checklists, design reviews, desk checks, peer reviews, and walkthroughs can only help improve the system. The problem with using some of these techniques with expert system development is that a complete examination of the knowledge base and its connectivity is rarely feasible. What can be done, however, is detailed exploration of pieces of it. The completeness of the testing is then dependent on how many pieces are explored. Modifications during the incremental development should also be planned before they are

implemented and reviewed afterwards to ensure that they were implemented correctly.

Methods for manual support and operations testing can include disaster testing and evaluations of the user interface. Disaster testing determines the level of training of the user, and it can also evaluate how well the system can recover from fatal errors. The user interface can best be evaluated by supplying a typical user with typical data and letting him run the system while an evaluator watches. In expert system testing, field testing (see below) is used to evaluate the operation of the system outside the lab. Some earlier evaluations of the user interface, however, can be very informative, even though they are performed in semi-controlled environments.

Checking the execution proficiency of the system can best be performed by using test data, system logs, and utility programs to evaluate the execution of the tests. Efficiency in an expert system can usually be improved by "tweaking" or adjusting the rules in the knowledge base. Volume testing (a form of stress testing) can also be performed to see how well the system handles larger than normal loads of work. If the system efficiency does not fall between some accepted range of values, changes will have to be made. But before any adjustments can be made, the developer must be able to follow the exact execution of a transaction. Especially after such changes are made, the system needs to be run again

through the whole gauntlet of testing to make sure other factors have not become invalid.

This list of testing methods is, of course, not exhaustive. The actual range of methods and tools used or available for a given application will depend on the application domain, the programming environment and the relative importance the system developers place on some of the testable factors. The next section discusses some specialized tools which may be most applicable to testing rule-based expert systems.

4.3.4 Specialized Expert System Testing Tools

So far in this chapter, only those testing tools familiar to the realm of traditional software engineering and applicable to expert system testing have been presented or discussed. This is not to say that expert system technologists have not developed some of their own tools along the way. Several specialized tools exist, and although some of them have, for the most part, been limited to a few specific applications, the potential utility of all of these tools is great. Some of these tools are automated, so their portability across all expert system applications is in question. Others, however, are manual tools, and they can be used in almost any situation.

The automated tools discussed in this section include rule base consistency checkers and rule cross-referencing tools. The non-automated tools which are most applicable to expert system testing are field testing and sensitivity analysis.

Rule Base Consistency Checking: Rule base consistency checking is potentially one of the most powerful tools available to expert system testing. Typically, the consistency checker is an automated tool, another program, and it is usually designed to work with a particular type of expert system shell or representational form.

One such tool in use is known as ARC (for ART rule checker), and it works with applications built on the ART (Automated Reasoning Tool) framework. The designers of ARC claim its techniques can be applied to most rule-based systems [Nguyen 1987], but the tool itself has only been used with applications produced with ART.

As the development of an expert system proceeds, the complexity of individual rules and their relationships with other rules increases. ARC is designed to help the expert system developers find certain logical errors in the knowledge base. These errors include: redundant rules, conflicting rules, subsumed rules, unnecessary IF conditions, and circular chain rules.

Redundant rules are those which are repeated one or more times in the rule base. Changes made to one rule in the

system are not noticed if the rule was redundant, for the other rules were not changed. Conflicting rules cause many different semantic problems in the rule base (such as responding with both "true" and "false" to the same query). Subsumed rules cause problems similar to redundant rules. For example, the rules "If A and B THEN C" and "IF A THEN C" are said to be in subsumption, and the knowledge engineer should examine them more closely. The knowledge engineer should also be aware of rules with unnecessary IF conditions. For example, in the rules "IF A and B THEN C" and "IF A and not B THEN C," the clauses "B" and "not B" are unnecessary. ARC also locates circular chain rules. The rules "IF A THEN B," "IF B THEN C," and "IF C THEN A" form a circular chain. These rules could cause the equivalent of an infinite loop and should be avoided in any rule-based expert system.

Rule Base Cross-referencing: Rule base cross-referencing tools are expensive in terms of the amount of processing time they require [Fickas 1987]. Typically, such a system links rules which rely upon or satisfy requirements of other rules in the knowledge base.

Cross-referencing the rule base provides several different kinds of information to the expert system developer. First of all, it provides a complete view of the inter-connectivity of the rule base. Rules which are isolated from others can be singled out and individually checked for correctness.

Clusters of high connectivity can be viewed as related knowledge, providing a certain level of knowledge modularization for the application. This information benefits domain experts as well as system developers, for it can aid in a classification of knowledge in that domain. Also, if a rule needs to be changed in the system, it is very easy to see which other rules are connected to this rule and which are not. This last point provides the testing developer with an indication of which rules should be affected when certain changes are made to the system and which should not.

Of course, the cross-referencing work should be repeated every time the rule base changes. Otherwise, the cross-references could be invalid and useless to the developers.

Field Testing: Although field testing is not unknown in the world of traditional software testing, few system developers depend on it as much as expert system developers seem to. Some applications, such as the Mentor project [Cochran et al. 1987], relied almost exclusively on field testing before releasing the system for general use.

Field testing is usually one of the last testing tools used in the development phase. Up to this point, the system has been internally tested, but the tests are controlled by the developers. Field testing lets the actual users get

their hands on the system and put it through its paces as if it were an accepted production system.

There are several difficulties with field testing which should be understood before using it. First, most of the results are very subjective and hard to analyze. The entire integrated system, including the intended user, is being tested. If one component fails, it may be difficult to pin-point the cause of the failure. Also, there is a potential problem with getting users to spend time evaluating a system which is not yet a tool they must use. Some users may not understand the importance of this test, and the results may be biased. It is clear that since field testing is such an important tool, it must be well planned to avoid any testing pitfalls.

Sensitivity Analysis: System adjustments can also be made using sensitivity analyses. To use sensitivity analysis, the tester enters the data for one typical query and receives the system's response. Then, he changes one parameter of the original query and evaluates the new response. If the difference between the two responses is not logical, then modifications need to be made to some rules in the knowledge base.

Although this testing tool is not known to have been used explicitly in expert system testing [O'Keefe et al. 1987], it has a high potential for being an extremely useful testing

tool. Sensitivity analysis gives the developer a way to look at the reasoning process from the outside. It is a relatively inexpensive testing tool requiring only existing test data, but it could consume a lot of time if used extensively.

4.4 MAINTENANCE PHASE TESTING

By the time the expert system reaches the maintenance phase, it has passed a battery of testing situations, been integrated into its intended operating environment, and has passed some more tests. Expert systems, unlike some conventional software applications, are not static programs. They continue to change and evolve over time, making it necessary for an on-going development process to continue through the life-span of the system.

It is expected that the system's performance levels may degrade over time for several reasons. Timely information contained in the knowledge base may become outdated, or ill-planned modifications to the system may simply invalidate it over time. These factors may affect the overall correctness or reliability of the system. Problem solving techniques used by the system to answer the queries may also become outmoded, which might decrease the users' confidence in the system's responses. Therefore, in order to increase

the longevity of the usefulness of the system, the on-going maintenance of the system must be well planned.

The frequency of testing in this phase is application-dependent. It may be pre-determined by system specifications, or it may be determined by usage or even by user reports (but if users have begun to complain about a system's performance, then proper maintenance procedures are probably not being followed).

The majority of changes required by the system after it has been put into use are domain-dependent. New discoveries and acceptance of new techniques may lead to the need to insert new data and correct outmoded or incorrect information in the knowledge base. If present, timely information in the system must also be updated from time to time as appropriate.

For these and other reasons, many expert system developers leave their operational systems in a kind of development environment [Harmon 1985]. This environment facilitates modifications and subsequent testing. Since the system remains in its developmental form, testing techniques and tools used during the development phase are still appropriate for use during the maintenance phase. Only the frequency or duration of the incremental cycles may change during the maintenance phase.

5.0 IMPLICATIONS OF THE RESEARCH

So where does any of this research leave the area of expert system testing?

Conclusions about the welfare of expert system testing have been posted before, and the results have not been all that favorable [Bobrow 1985; Pohl 1986]. Perhaps a part of the blame lies with the high expectations of expert performance from these systems. So, should software engineers lower their standards? No. Should they re-define their standards? Maybe. After all, this technology does not have a true algorithmic basis, and it is difficult (if not impossible) to judge it by the traditional standards set for algorithmic systems. But first, before anything else is done, there are a few more areas related to expert system testing which should be investigated and standardized to some extent.

5.1 PROGRAMMING ENVIRONMENTS

Perhaps the topic with the most influence on overall expert system development and testing is the selection of a programming environment (including the selection of expert system shells and programming languages). For a particular application, this decision may be paramount to all others.

Choice of a particular programming environment probably dictates (or at least narrows the field of decisions about) the form of knowledge representation to be used, programming style, a development methodology, and, in some ways, a strategy for testing. A single programming environment, when evaluated for its own merits and faults, is probably on equal footing with others. But, when one considers some of the extra tools which have been designed to be used with some expert system shells (such as Teiresias for MYCIN or ARC for ART), the overall perceived value of some environments may increase.

Of course, the expert system developer is not forced to choose an existing programming environment. Use of an existing environment, however, may relieve the developer from having to make other decisions and from having to build his own development and testing tools.

The real problem with having so many different programming environments available is trying to compare similar systems which were developed in different environments. There is no standardization between the different environments; the tools available in one environment may not be available in another. Mainly for this reason, inter-system testing can be very difficult to accomplish.

5.2 DEVELOPMENT TOOLS FOR EXPERT SYSTEMS

One area which can certainly use some improvement is the availability of development tools across the spectrum of expert system programming environments. While some such tools have been designed and built [Francez 1985; Teitelman et al. 1981; Waters 1982; Davis et al. 1982], most of these are still experimental and need to be polished before they will be of much use to the generic expert system developer. Most development tools currently available aid in the knowledge acquisition stage of expert system development. Since this stage has been identified as the "bottleneck" [Weiss 1984; Partridge 1986] of expert system development, knowledge acquisition tools are very much in demand.

5.2.1 Knowledge Acquisition Tools

Perhaps the best known of these experimental development tools is Teiresias, a knowledge acquisition tool built by Randall Davis [1982]. Teiresias is designed to be used exclusively with MYCIN-like expert systems or systems built with the E-MYCIN expert system shell.

Teiresias is most helpful during the knowledge acquisition stage of the implementation phase. Briefly, here is how Teiresias works. The knowledge base contains rules pertaining to the application domain; the inference engine

contains information on how to use these rules; and Teiresias provides rules about the rules, called meta-rules. By using meta-rules, some semantic errors in the rules can be caught as they are introduced to the system. This preventative measure not only helps to unclog some the knowledge acquisition bottleneck which plagues many development attempts, but it also acts as a test for semantic errors.

Teiresias will second-guess the correctness of a newly entered rule based on the existence of similar rules in the knowledge base. For example, there is a meta-rule in Teiresias which states:

> "... most rules about what the category of a (sic) organism might be, that mention
> the sight of the culture (and)
> the infection
> also mention ... the portal of entry of a (sic) organism." [Davis et al. 1982]

With this meta-rule, whenever a rule contains information on the sight of the culture and the infection but does not mention the "portal of entry for the organism," Teiresias interrupts the process of entering the rule and questions the user.

One important point to make about Teiresias is that the system develops these meta-rules by itself. Teiresias also provides a rule editor which facilitates changing erroneous rules as well as a natural language module which translates the rules into English so that the user can also verify their correctness.

Tools such as Teiresias are very useful during the implementation phase for both knowledge acquisition and for incremental testing for rule base consistency. This tool is also very useful during the maintenance phase. By automating the consistency checking part of the system maintenance, the system developers relieve themselves of part of the heavy burden of system testing.

5.2.2 Automating Generic Testing Tools

There is a great need across all expert system programming environments for automated testing tools. Only a few exist, such as ARC (see section 4.3.4). All or some of the functions performed by ARC should be applicable to all rule-based expert systems.

5.3 A LESSON LEARNED THE HARD WAY

If expert system developers had to list the most important issues of a testing strategy for expert systems, one point would rank the highest on most of the lists: early user involvement [Gaschnig et al. 1983; Weiss 1984; Roach 1988]. As in almost all software projects, it is extremely important for the development team to be able to correctly identify the client (the party paying for the system) and the user (the party who will eventually have to use the system).

Distinguishing between these two sets (which may not be disjoint) of people should be a high priority before a testing strategy is defined for the project.

Software politics may determine if pleasing the client and satisfying the needs of the user lead to the same software goals or not. If they do not, it is up to the developer to make sure that the project meets all expectations. Otherwise, the project is doomed.

Early user involvement is important in several ways. First, it can help to eliminate the "middle man" in software development, the client. In cases where the client is not an end-user, it may be difficult for him to specify all of the needed functions of the system. Only by communicating with typical users can the developer come close to correctly defining the system. This type of early communication can help to alleviate the chances of designing and building the wrong system, which can be just as disastrous as an implementation which does not work correctly. Also, mistakes are easier and cheaper to correct if they are found early.

5.4 FROM THE BEGINNING

The overall testing strategy begins even before the project specifications are outlined. Before expert system technology can be considered as a viable problem solving

technique for an application, there are several concerns which must be addressed first.

To begin with, the problem domain must be thoroughly investigated. It may be that expert system technology is not the most appropriate approach to solving the problem. An algorithmic solution might be applicable. If so, this solution might be more desirable to the project developers. If an algorithmic approach is not applicable, however, current expert system technology could be considered. Can the knowledge in the problem domain be stated in the form of rules? If so, a rule-based expert system is indicated; otherwise, some other form of expert system would be better suited for the job.

Once it is decided that rule-based expert system technology is applicable to solving the problem, some developmental concerns can be looked at. The development team can begin to investigate the available programming environments. Some costs, such as the monetary cost of obtaining and using particular programming environments can be estimated at this point.

Other developmental needs can also be investigated. Potential knowledge sources must be identified early. Most rule-based systems rely on a single knowledge source (usually a human expert), but this is not always the case [Harmon 1985; Jackson 1986]. Multiple human experts or a combination of human experts and library references could be used.

Regardless of which is chosen, the development team must insure that the knowledge source will be available during the development of the system. Apart from identifying the knowledge source, the developers must also distinguish between the project client and the end user (see section 5.3).

Once these initial concerns have been addressed (and expert system technology has been chosen as the desired problem solving technique), the system definition phase can begin in earnest. At this point, the developmental stages of the project lifecycle must be clearly identified. Then for each of these developmental stages, software attributes can be ranked by importance and appropriate testing techniques can be selected. This information provides the expert system developer with guidelines for the development as well as the testing of the overall system.

5.5 DIRECTIONS FOR FUTURE RESEARCH

In the past, AI systems have been tainted by unfulfilled expectations. Grand predictions in the 1960's did not come to be, for the existing technology just could not support some of the innovative ideas [Weiss 1984]. Research in AI has continued, and many of the newest break-throughs (the development of learning systems and parallel machines) in software and hardware have shown promising results. The

entire field of AI is on the verge of changing, and many of these changes will, of course, affect expert system development and testing. Of course, the field still has some problems which need to be solved before venturing further into the unknown, but the unknown is currently looming on the near horizon.

When considering directions for future research in expert systems, two of these break-throughs appear to bring the most promise. The implications of testing an expert system which has learning capabilities or which is implemented on a parallel processing machine are presented below.

5.5.1 Expert Systems that Change with Ease

So far, this thesis has only discussed expert systems with large global knowledge bases. Modifications to the system are only made by trained maintenance personnel, and all modifications are available to all the users. On the horizon, however, are systems which correct errors in the knowledge base or make changes depending on the queries they encounter during use by themselves. These systems are often referred to as learning systems, for they appear to learn, both from their experiences and from their mistakes.

Learning Systems: The main problem with an expert system that learns is that each system must, almost by definition,

be unique. Even if two systems begin as "twins," there is no guarantee that the knowledge bases will remain identical once they are separated.

The main differences between testing these future systems and testing existing ones will arise during the maintenance phase. With non-learning systems, if an error is found in the knowledge base, it can be corrected the same way in each system. With learning systems, however, each system would have to be corrected independently of each other. Some of the systems may already have found and fixed the inconsistency internally. On the other hand, it is also possible that some systems could have modified other rules or added new rules based on the original erroneous material. The expert system maintenance personnel can not be sure what has happened because of the error until he thoroughly checks out each system, one by one.

Local Knowledge Bases: Most current expert systems can only be modified by maintenance personnel. These systems have a global knowledge base, so any changes made to the system can be reached by all users. Davis [1982] proposes an environment somewhere in between current systems and future learning systems. In Teiresias, individual users are allowed to make changes to the knowledge base, but these changes are kept in a local area for each user. In this way, changes entered by Mr. Jones will not affect how Mr. Smith wants to

interact with some rules. In this environment, maintenance personnel can still make changes to the global knowledge base, but it is up to the individual users to maintain their local changes.

Several potential problem with this system also exist. The main concern is that changes made to the global system may cause the local environment to function improperly or inconsistently. Clearly, there is a need for strong communication between maintenance personnel and the users. Still, each user will probably need to evaluate his own local environment before being able to trust it again.

5.5.2 Expert Systems and Parallel Processing.

Existing expert systems have only been implemented on single processor machines. With the advent of parallel processors, however, there are many new directions which expert system development can travel

Parallel processors allow more than one machine operation to occur simultaneously. The implementation of an expert system on such a machine could potentially solve some existing quandaries and will most assuredly cause some new ones.

Parallel Processing - Approaching Real Time: One of the biggest complaints about some proposed expert systems is

their inability to process a query in real time (or as least as fast or faster than a competent human expert could process it). By using a parallel processor (with n independent processors), the amount of time required to process one query could decrease by a factor of n, but it probably will not decrease that much.

Even though an ordering of rules is not supposed to be imposed on a knowledge base (by one of the few standards of coding which exist for expert system development), some expert systems do have an explicit need for ordering some rules [Hayes-Roth et al. 1983; Davis 1982]. Where a specific ordering is not necessary, however, each processor could proceed down a path in the knowledge base.

There are, of course, many problems which must be solved before such a system can be implemented. The level of communication between each processor must be very high and very fast so that processors do not duplicate each other's work. Testing such a system would also raise many new questions. There is no way to assure that identical queries on identical systems will return identical responses because there is no way to pre-determine which processor does which operation at which time.

Parallel Processing and Alternative Solutions: Rarely in the real world will any two human experts agree exactly on how to solve a complex problem. Existing expert systems,

however, tend to only present a single method for solving a problem (or answering a question). If it is important for an application to receive information from all of the possible solution methods or paths, an expert system should be modified to allow for the extra processing. Of course, this requires extra processing time in a single processor system. An expert system allowing for alternative solutions could really benefit from parallel execution in a multi-processor system.

In expert systems allowing for alternative solutions, there is a small dilemma in choosing which solution to accept when they are equally weighted. If more than two alternative methods are used, the system could accept the response presented by the majority of the alternatives. This type of fault tolerance measure is known as "protective redundancy" [Anderson et al. 1979]. Protective redundancy could also be used with multiple single solution systems which cover the same application domain.

REFERENCES

Addis, T.R. (1986), Designing Knowledge-based Systems, Prentice-Hall, Englewood Cliffs, N.J.

Adelson, B. and E. Soloway (1985), "The Role of Domain Experience in Software Design," IEEE Transactions on Software Engineering SE-11, 11 (Nov.), 1351-1360.

Akey, M.L. and K.A. Dunkelberger (1987), "Building Near-term Military AI Systems: Formalisms and an Example," In Proceedings of the Third Conference on Artificial Intelligence Applications (Kissimmee, Fla., Feb. 23-27), IEEE, Piscataway, N.J., pp. 202-206.

Anderson, T. and B. Randell (1979), Computing Systems Reliability, Cambridge University Press, N.Y.C.

Balzer, R., T.E. Cheatham, and C. Green (1983), "Software Technology in the 1990's: Using a New Paradigm," IEEE Computer 16, 11 (Nov.), 39-45.

Beizer, B. (1983), Software Testing Techniques, Van Nostrand Reinhold Company, N.Y.C.

Beizer, B. (1984), Software System Testing and Quality Assurance, Van Nostrand Reinhold Company, N.Y.C.

Berrill, N.D. and M.A. Ould (1985), A Practical Handbook for Software Development, Cambridge University Press, N.Y.C.

Berry, D.C. and D.E. Broadbent (1986), "Expert Systems and the Man-machine Interface," Expert Systems 3, 4 (Oct.), 228-231.

Berry, D.C. and D.E. Broadbent (1987), "Expert Systems and the Man-machine Interface - Part Two: The User Interface," Expert Systems 4, 1 (Feb.), 18-28.

Bobrow, D.G. (1985), "If Prolog is the Answer, What is the Question? or What it Takes to Support AI Programming Paradigms," IEEE Transactions on Software Engineering SE-11, 11 (Nov.), 1401-1408.

Boehm, B.W. (1984), "Verifying and Validating Software Requirements and Design Specifications," IEEE Software 1, 1 (Jan.), 75-88.

Buchanan B.G. (1986), "Expert Systems: Working Systems and the Research Literature," Expert Systems 3, 1 (Jan.), 32-50.

Budde, R. et al. (1984), Approaches to Prototyping, Springer-Verlag, N.Y.C.

Butler, K.A. (1987), "Application of Correlation Measures for Validating Structured Selectors," In Proceedings of the Third Conference on Artificial Intelligence Applications (Kissimmee, Fla., Feb. 23-27), IEEE, Piscataway, N.J., pp. 327-330.

Cochran, E.L. and B.L. Hutchins (1987), "Testing, Verifying, and Releasing an Expert System: The Case History of Mentor," In Proceedings of the Third Conference on Artificial Intelligence Applications (Kissimmee, Fla., Feb. 23-27), IEEE, Piscataway, N.J., pp. 163-167.

Cuadrado, C.Y. and J.L. Cuadrado (1986), "Handling Conflicts in Data," BYTE 11, 12 (Nov.), 193-202.

Davis, R. and D.B. Lenat (1982), Knowledge-Based Systems in Artificial Intelligence, McGraw-Hill International Book Company, N.Y.C.

DeMillo, R.A. et al. (1987), Software Testing and Evaluation, Benjamin Cummings Publishing Company, Reading, Mass.

Elithorn, A. and R. Banerji (1981), Artificial & Human Intelligence, North-Holland, N.Y.C.

Fickas, S. (1985), "Design Issues in a Rule-Based System," In Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments 20 (Seattle, Wash., July 7-11), IEEE, Piscataway, N.J., pp. 208-315.

Fickas, S. (1987), "Development Tools for Expert Systems," In Expert Systems: The User Interface, J. Hendler et al., Eds., Ablex Publishing Company, N.Y.C.

Francez, N. et al. (1985), "An Environment for Logic Programming," In Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments 20 (Seattle, Wash., July 7-11), IEEE, Piscataway, N.J., pp. 179-190.

Gaschnig, J. et al. (1983), "Evaluation of Expert Systems: Issues and Case Studies," In Building Expert Systems,

F. Hayes-Roth, D.A. Waterman, and D.B. Lenat, Eds., Addison-Wesley Publishing Company, Reading, Mass.

General Electric (1986), Software Engineering Handbook, McGraw-Hill Book Company, N.Y.C.

Giddings, R.V. (1984), "Accommodating Uncertainty in Software Design," Communications of the ACM 27, 5(May), 428-434.

Glass, R.L. (1979), Software Reliability Guidebook, Prentice-Hall, Inc., Englewood Cliffs, N.J.

Harmon, P. and D. King (1985), Expert Systems: Artificial Intelligence in Business, John Wiley & Sons, Inc., N.Y.C.

Hausen, H. (1983), Software Validation, North Holland, N.Y.C.

Hayes-Roth, F., D.A. Waterman, and D.B. Lenat (1983), Building Expert Systems, Addison-Wesley Publishing Company, Reading, Mass.

Hewitt, C. (1985), "The Challenge of Open Systems," BYTE 10, 4 (Apr.), 223-239.

Jackson, P. (1986), Introduction to Expert Systems, Addison-Wesley Publishing Company, Reading, Mass.

Karna, K.N. (1985), Expert Systems in Government Symposium, IEEE Computer Society Press, Washington, D.C.

Klahr, P. and D.A. Waterman (1986), Expert Systems: Techniques, Tools and Applications, Addison-Wesley Publishing Company, Reading, Mass.

Levine, R.I., D.E. Drang, and B. Edelson (1986), A Comprehensive Guide to AI and Expert Systems, McGraw-Hill Book Company, N.Y.C.

Liebowitz, J. (1986), "Useful Approach for Evaluating Expert Systems," Expert Systems 3, 2 (Apr.), 86-96.

McCallum, J.C. (1985), "Maintenance of Expert Systems: Life-Cycle Validity," SIGART Newsletter, 94 (Oct.), 26-27.

Michaelsen, R.H., D. Michie, and A. Boulanger (1985), "The Technology of Expert Systems," BYTE 10, 4 (Apr.), 303-312.

Mili, A. (1985), An Introduction to Formal Program Verification, Van Nostrand Reinhold Company, N.Y.C.

Moskowitz, L. (1985) "Rule-Based Programming," BYTE 11, 12 (Nov.), 217-224.

Mostow, J. (1985), "What is AI? And What Does It Have to Do with Software Engineering?" IEEE Transactions on Software Engineering SE-11, 11 (Nov.), 1253-1255.

Meyers, G. (1979), The Art of Software Testing, John Wiley & Sons, N.Y.C.

Nguyen, T.A. (1987), "Verifying Consistency of Production Systems," In Proceedings of the Third Conference on Artificial Intelligence Applications (Kissimmee, Fla., Feb. 23-27), IEEE, Piscataway, N.J., pp. 4-8.

Neches, R., W. Swartout, and J. Moore (1985), "Enhanced Maintenance and Explanation of Expert Systems Through Explicit Models of Their Development," IEEE Transactions on Software Engineering SE-11, 11 (Nov.), 1337-1351.

O'Keefe, R.M., O. Balci, and E.P. Smith (1987), "Validating Expert System Performance," IEEE Expert 2, 4 (Winter), 81-90.

Overstreet, C.M., R.E. Nance, O. Balci, and L.F. Barger (1986), "Specification Languages: Understanding Their Role in Simulation Model Development," Technical Report SRC-87-001, Department of Computer Science, Virginia Tech, Blacksburg, Va., Dec.

Parnas, D.L. (1979), "Designing Software for Ease of Extension and Contradiction," IEEE Transactions on Software Engineering SE-5, 2 (Mar.), 128-137.

Parnas, D.L. (1985), "Software Aspects of Strategic Defense Systems," American Scientist 73, 5 (Sept.), 432-440.

Partridge, D. (1986), Artificial Intelligence: Applications in the Future of Software Engineering, Halstead Press, N.Y.C.

Perry, W.E. (1983), A Structured Approach to Systems Testing, Prentice-Hall, Inc., Englewood Cliffs, N.J.

Pohl, I. (1986), "SDI Software: AI is not the answer," ACM SIGSOFT Software Engineering Notes 11, 2 (Apr.) 18-19.

Quirk, W.J. (1985), Verification and Validation of Real-Time Software, Springer-Verlag, N.Y.C.

Ramamoorthy, C.V., S. Shekhar, and V. Garg (1987), "Software Development Support for AI Programs," Computer 20, 1(Jan.), 30-40.

Roach, J.W. (1988), Personal correspondence.

Samet, H. (1982), "Code Optimization Considerations in List Processing Systems," IEEE Transactions on Software Engineering SE-8, 2 (Mar.), 107-112.

Sell, P.S. (1985), Expert Systems - A Practical Introduction, Halstead Press, N.Y.C.

Slagle, J.R. (1971), Artificial Intelligence: The Heuristic Programming Approach, McGraw-Hill Book Company, N.Y.C.

Subrahmanyam, P. A. (1985) "The 'Software Engineering' of Expert Systems: Is Prolog Appropriate?" IEEE Transactions on Software Engineering SE-11, 11 (Nov.), 1391-1400.

Swartout, W. and R. Balzer (1982), "On the Inevitable Intertwining of Specification and Implementation," Communications of the ACM 25, 7 (July), 438-440.

Teitelman, W. and L. Masinter (1981), "The Interlisp Programming Environment," Computer 14, 4 (Apr.) 25-33.

Turing, A.M. (1950), "Computing Machinery and Intelligence," Mind 59, 236 (Oct.), 433-460.

Wada, E., editor (1985), Logic Programming '85, Springer-Verlag, N.Y.C.

Waite, W.M. (1973), Implementing Software for Non-Numeric Applications, Prentice-Hall, Inc. Englewood Cliffs, N.J.

Wasserman, A.I. and S. Gutz (1982), "The Future of Programming," Communications of the ACM 25, 3 (Mar.), 196-206.

Waterman, D.A. (1986), A Guide to Expert Systems, Addison-Wesley Publishing Company, Reading, Mass.

Waters, R.C. (1982), "The Programmer's Apprentice: Knowledge Based Program Editing," IEEE Transactions on Software Engineering SE-8, 1 (Jan.), 1-12.

Weiss, S. and C.A. Kulikowski (1984), A Practical Guide to Designing Expert Systems, Rowman & Allanheld, Totowa, N.J.

Winston, P.H. (1985) "The Lisp Revolution," BYTE 10, 4 (Apr.), 209-218.

VITA

Lee Anne Hite was born on July 15, 1962 in Wichita Falls, Texas. She began studying foreign languages at Virginia Tech in the Fall of 1980 and graduated Magna Cum Laude with a degree in French four years later. In September of 1984, she continued her education at Virginia Tech and began work on a Masters degree in computer science. She accepted a job as an instructional programmer for interactive video projects at Educational Technologies of Virginia Tech in June of 1987.