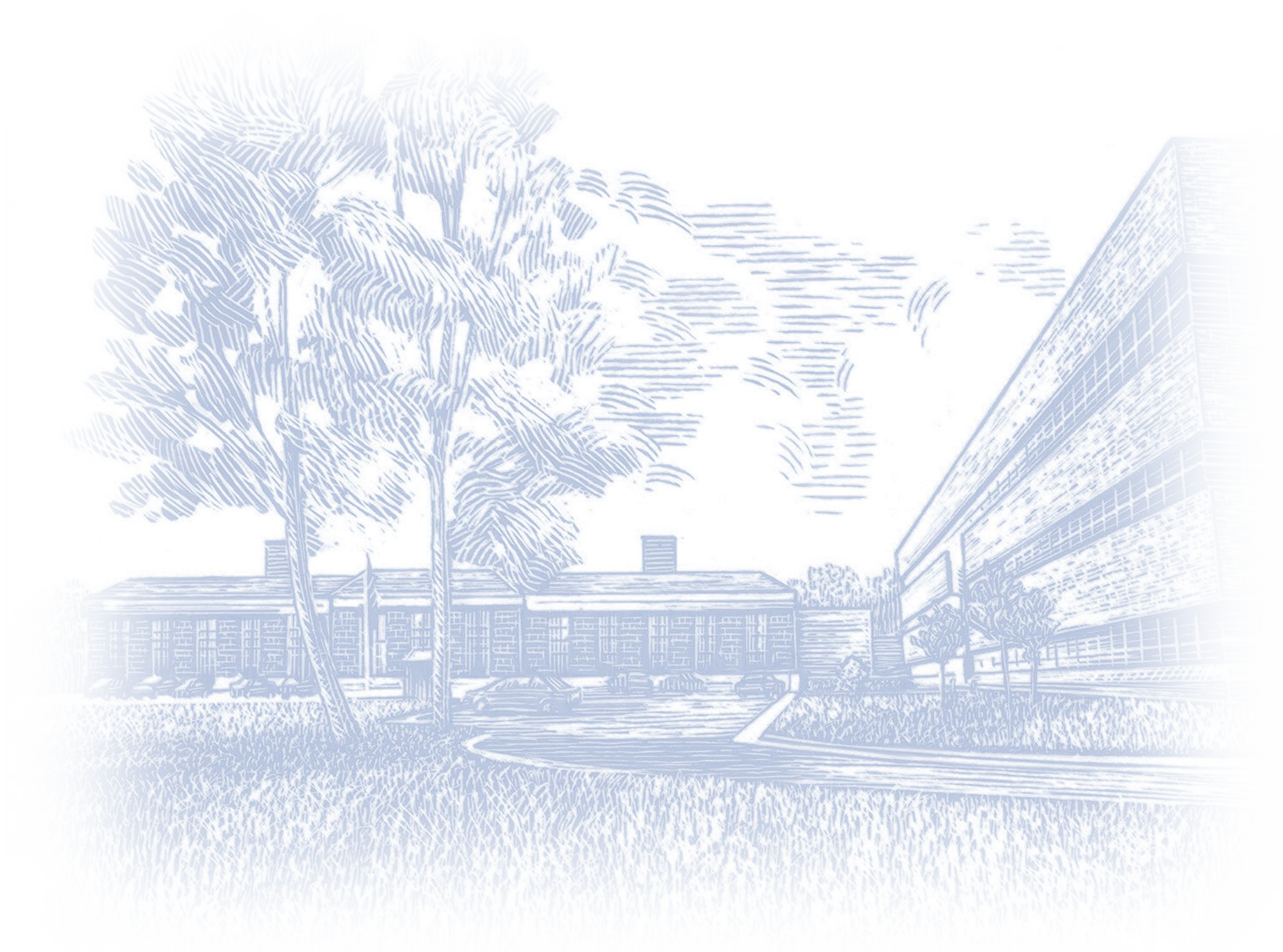
Verification, Validation and Evaluation of Expert Systems, Volume I, A FHWA Handbook

Publication No.: N/A **August 1995**



U.S. Department of Transportation
Federal Highway Administration

Federal Highway Administration
Turner-Fairbank Highway Research Center
6300 Georgetown Pike, McLean, VA 22101

The original format of this document was an active HTML page(s). The Federal Highway Administration converted the HTML page(s) into an Adobe® Acrobat® PDF file to preserve and support reuse of the information it contained.

The intellectual content of this PDF is an authentic capture of the original HTML file. Hyperlinks and other functions of the HTML webpage may have been lost, and this version of the content may not fully work with screen reading software.

Foreword

A draft Verification, Validation and Evaluation of Expert Systems,. Volume I, A FHWA Handbook, has been developed. The purpose of this communication is not to present an official document, but to share a work in process and to solicit advice. It is the intention of the Development Team to produce a quality product that will truly be of value to those developing and testing expert systems. Thus I encourage you to critically review this draft handbook and provide any suggestions for improvement. We want to hear the bad news as well as the good so that we can improve this handbook.

This draft handbook discusses how verification, validation and evaluation (VV&E) should be incorporated into the expert system life cycle, shows how to partition knowledge bases with or without expert domain knowledge, presents knowledge models, presents methods of validating domain (the experts') knowledge, and discusses management issues related to expert systems development and testing. Mathematical proofs for partitioning and consistency and visualization of concepts are also presented.

I would have considered the draft handbook to be of little use if we were unable to do a pen and paper analysis using its procedures on a real-world expert system of reasonable size and complexity (with computer support for matrix manipulation, solving differential equations, etc.). The expert system PAMEX: Expert System for Maintenance Management of Flexible Pavements, with 327 rules, 20 input variables and 59 qualifiers, was selected for the in-house pen and paper analysis. The results of this analysis provided insights into PAMEX and identified errors in programming that the developers and users were unaware of.

This draft handbook will also be field tested in a number of States using operational expert systems as test cases. At the end of this testing (probably in about one year), the handbook will be updated based on the results of the testing.

It will be impossible to respond directly to every comment, but be assured that every comment will be reviewed and given the consideration it deserves. Thank you in advance for your assistance.

Sincerely yours,

James A. Wentworth
Chief, Advanced Research Team
Office of Safety and Traffic Operations
Research and Development
Federal Highway Administration
E-Mail: tfhrc.webmaster@dot.gov

Technical Report Documentation Page

1. Report No. FHWA-RD	2. Government Accession No.	3. Recipient's Catalog No
4. Title and Subtitle VERIFICATION, VALIDATION AND EVALUATION OF EXPERT SYSTEMS Volume 1		5. Report Date 6. Performing Organization Code
7. Author(s) James A. Wentworth, Rodger Knauss, Hamid Aougab		8. Performing Organization Report No.
9. Performing Organization Name and Address MiTech Incorporated 9430 Key West Avenue, Suite 100 Rockville, Maryland 20860 Federal Highway Administration Washington, DC		10. Work Unit No. (TRAIS) 11. Contract or Grant No. DTFH61-93-Z-00043
12. Sponsoring Agency Name and Address Federal Highway Administration Office of Safety and Traffic Operations R&D 6300 Georgetown Pike McClean, Virginia 22101-2296		13. Type of Report and Period Covered Interim Report April 1993-August 1995 14. Sponsoring Agency Code

15. Supplementary Notes

Contracting Officer's Technical Representative (COTR): James A. Wentworth, HSR-1

This research was funded by the Federal Highway Administration

16. Abstract

The importance and difficulty of performing verification, validation and evaluation (VV&E) on expert systems cannot be overstated. This is one of the major factors slowing the development and acceptance of expert systems in the transportation community. There is little agreement among experts on how to accomplish the VV&E of expert systems. The complexity and uncertainty related to these tasks has lead to the situation where most expert systems are not adequately tested. In some cases testing is ignored until late in the development cycle, always with predictable disastrous results.

This guide discusses how VV&E should be incorporated into the expert system lifecycle, shows how to partition knowledge bases with and without expert domaine knowledge, presents knowledge models, presents methods for validating underlying the experts' knowledge, and presents management issues related to expert systems development and testing. Mathematical proofs for partitioning, consistency, and completeness and visualization of concepts are presented.

17. Key Words verification, validation, evaluation, expert systems, knowledge base, knowledge engineering, partitioning, consistence, completeness.		18. Distribution Statement No restriction. This document is available to the public through the National Technical Information Service, Springfield, VA 22161	
19. Security Classif. (Of this report) Unclassified	20. Security Classifi. (Of this page) Unclassified	21. No. Of Pages	22. Price

Form DOT F 1700.7 (8-72) Reproduction of completed page authorized(PF V2.1, 12/13/93)

Table of Contents

List of Figures

List of Tables

Introduction

Roadway engineering and construction pre-date Roman times. Over the centuries, standards in design and construction and the documentation of practice have been raised to very high levels. In the process of modernizing and improving design, construction, and maintenance, new approaches and technologies have been incorporated into civil engineering practice. Initially, many of the new technologies do not achieve the levels of reliability and standardization required by the civil engineering profession. Regrettably, many expert systems fall into this category, due partly to the lack of verification, validation, and evaluation standards.

The goals of expert systems are usually more ambitious than those of conventional or algorithmic programs. They frequently perform not only as problem solvers but also as intelligent assistants and training aids. Expert systems have great potential for capturing the knowledge and experience of current senior professionals (many of whom are approaching retirement age) and making it available to others in the form of training aids or technical support tools. Applications include design, operations, inspection, maintenance, training, and many others.

In traditional software engineering, testing [verification, validation and evaluation (VV&E)] is claimed to be an integral part of the design and development process. However, in the field of expert systems, there is little consensus on what testing is necessary or how to perform it. Further, many of the procedures that have been developed are so poorly documented that it is difficult, if not impossible, for the procedures to be reproduced by anyone other than the originator. Also, many procedures used for VV&E were designed to be specific to the particular domain in which they were introduced. The complexity and uncertainty related to these tasks has led to a situation where most expert systems are not adequately tested.

Impelled by the existing environment of lack of consensus among experts and inadequate procedures and tools, the FHWA developed this guideline for expert system verification, validation, and evaluation, complete with software to implement recommended techniques. The guideline is needed because knowledge engineers today do not often design and carry out rigorous test plans for expert systems. The software is necessary because real-world knowledge bases containing hundreds of rules and dozens of variables are difficult for humans to assimilate and evaluate. Computerized verification and validation (V&V) tools would also enable the knowledge engineer to use interim V&V reports to guide knowledge acquisition and coding, something that is too labor-intensive with hand methods.

Basic Definitions

This guide covers *verification, validation, and evaluation* of *expert systems*. An expert system is a computer program that includes a representation of the experience, knowledge, and reasoning processes of an expert. Figure 5.1 shows a six rule expert system that will be used as an example throughout this guide.

Verification of an expert system (or any computer system for that matter) is the task of determining that the system is built according to its specifications. *Validation* is the process of determining that the system actually fulfills the purpose for which it was intended. *Evaluation* reflects the acceptance of the system by the end users and its performance in the field. In other words *(Miskell et al, 1989)*,

- **verify** to show **the system is built right**
- **validate** to show **the right system was built**
- **evaluate** to show **the usefulness of the system**.

Verification

As stated above, verification asks the question "is the system built right?," i.e, verification is checking that the knowledge base is complete and that the inference engine can properly manipulate this information. Issues raised during verification include:

- Does the design reflect the requirements? Are all of the issues contained in the requirements addressed in the design?
- Does the detailed design reflect the design goals?
- Does the code accurately reflect the detailed design?
- Is the code correct with respect to the language syntax?

When the program has been verified, it is assured that there are no "bugs" or technical errors.

Validation

Validation answers the question "is it the right system?" "is the knowledge base correct?" or "is the program doing the job it was intended to do?" Thus, validation is the determination that the completed expert system performs the functions in the requirements specification and is usable for the intended purposes. It is impossible to have an absolute guarantee that a program satisfies its specification, only a degree of confidence that a program is valid can be obtained. Issues addressed during validation of an expert system include:

- How well do inferences made compare with knowledge and heuristics of experts in the field?
- How well do inferences made compare with historic (known) data?
- What fraction of pertinent empirical observations can be simulated by the system?
- What fraction of model predictions are empirically correct?
- What fraction of the system parameters does the model attempt to mimic?

Evaluation

Evaluation addresses the issue "is the system valuable?" This is reflected by the acceptance of the system by its end users and the performance of the system in its application. Pertinent issues in evaluation are:

- Is the system user friendly, and do the users accept the system?
- Does the expert system offer an improvement over the practices it is intended to supplement?
- Is the system useful as a training tool?

- Is the system maintainable by other than the developers?

To illustrate the difference, the task might be to build a system that computes the serviceability coefficient of pavement. The specifications for the system are contained in textbooks that define the coefficient. To validate the system, test the serviceability program on examples in the texts and other test cases, comparing the results of the program with independently computed coefficients for the same examples. It is important to use a test set that covers all the important cases and contains enough examples to make sure that correct results are not just anomalies. Once the system is validated, the next step is to verify it. This involves completeness and consistency checks and examining for technical correctness using techniques such as are described in this handbook. The final step is evaluation. For the serviceability program, this means giving the system to engineers to use in computing the coefficient. Although the system is known to produce the correct result, it could fail the evaluation because it is too cumbersome to use, requires data that is not readily available, does not really save any effort, does something that can be estimated accurately enough without a computer, solves a problem rarely needed in practice, or produces a result not universally accepted because different people define the coefficient in different ways.

Need for V&V

It is very important to verify and validate expert systems as well as all other software. When software is part of a machine or structure that can cause death or serious injury, V&V is especially critical. In fact, there have already been failures of expert systems and other software that have resulted in death. For example, a robotized overhead material mover struck an overhead crane at an Alcoa aluminum plant, killing the crane operator, because its narrow-field vision system saw only an interior region of the crane front, a blank field to the robot. In another case, a much-patched system for cancer radiation treatment gave a fatal dose to at least one patient, because the operator overrode the emergency stop; it had given repeated false alarms in past situations.

Expert systems use computational techniques that involve making guesses, just as human experts do. Like human experts, the expert system will be wrong some of the time, even if the expert system contains no errors. The knowledge on which the expert system is based, even if it's the best available, does not completely predict what will happen. For this reason alone, it is important for the human expert to validate that the advice being given by the expert system is sound, this is especially critical when the expert system will be used by persons with less expertise than the expert, who can not themselves judge the accuracy of the advice from the expert system.

In addition to mistakes which an expert system will make because the available knowledge is not sufficient for prediction in every case, expert systems contain only a limited amount of knowledge concentrated in carefully defined knowledge areas. Today's expert systems have no common sense knowledge. They only "know" exactly what has been input into their knowledge bases. There is no underlying truth or fact structure to which it can turn in cases of ambiguity. This means that an expert system containing some errors in its knowledge base can make mistakes that would seem ridiculous to a human, and not realize that a mistake had occurred. [On the other hand, expert systems do not get tired or sick or bored or fall in love, and therefore avoid some of the "careless" mistakes that a person might make, particularly on repetitive problems.] If the expert system does not realize its mistake, and it is being used by a person with limited expertise, there is nobody to detect the error. Therefore, where the expert system is going to be used by someone without expertise, and the decisions made have the potential for harm if made badly, the very best effort at verification and validation is required.

Problems in Implementing Verification, Validation, and Evaluation for Expert Systems

One of the impediments to a successful V&V effort for expert systems is the nature of expert systems themselves. Expert systems are often employed for working with incomplete or uncertain information or "ill structured" situations (Giarratano and Riley, 1989). These are cases where, as in a diagnostic expert system, not all symptoms for all malfunctions are known in advance. In these situations, reasoning offers the only hope for a good solution. Since expert system specifications often do not provide a precise criteria against which to test, there is a problem in verifying, validating, and evaluating expert systems according to the definitions in Section 1. For example, specifying that a speech recognition system should understand speech does not define a testable standard for the system. Some vagueness in the specifications for expert systems is unavoidable; if there are precise enough specifications for a system, it may be more effective to design the system using conventional programming languages.

Another problem in VV&E for expert systems is that expert system languages are unstructured to accommodate the relatively unstructured applications. However, rigid structure in implementing code is a key technique used in writing verifiable code, such as the Cleanroom approach. Cleanroom software specification (Linger, 1993) begins with a specification of required system behavior and architecture. Many expert systems cannot conform to the rigidity required by this quality control method used principally for conventional programming.

Intended Audiences for the Handbook

The following table describes the intended audiences for the handbook, and the parts of the handbook that will be most useful to these audiences:

Table 1-1: Intended Audiences for the Handbook

Audience	Task to be Performed	Part of Handbook
Managers	Manage expert system project	Introduction
Knowledge Engineers	Build new expert systems	Techniques VV&E on New Systems
Knowledge Engineers	Perform VV&E on existing systems	Techniques VV&E on Existing Systems
Highway Engineers	Insure that a correct new expert system is built	VV&E on New Systems
Highway Engineers	Insure that an existing expert system has been validated	VV&E on Existing Systems
Software Researchers	Critique and extend VV&E methods	Techniques VV&E on Existing Systems VV&E on New Systems

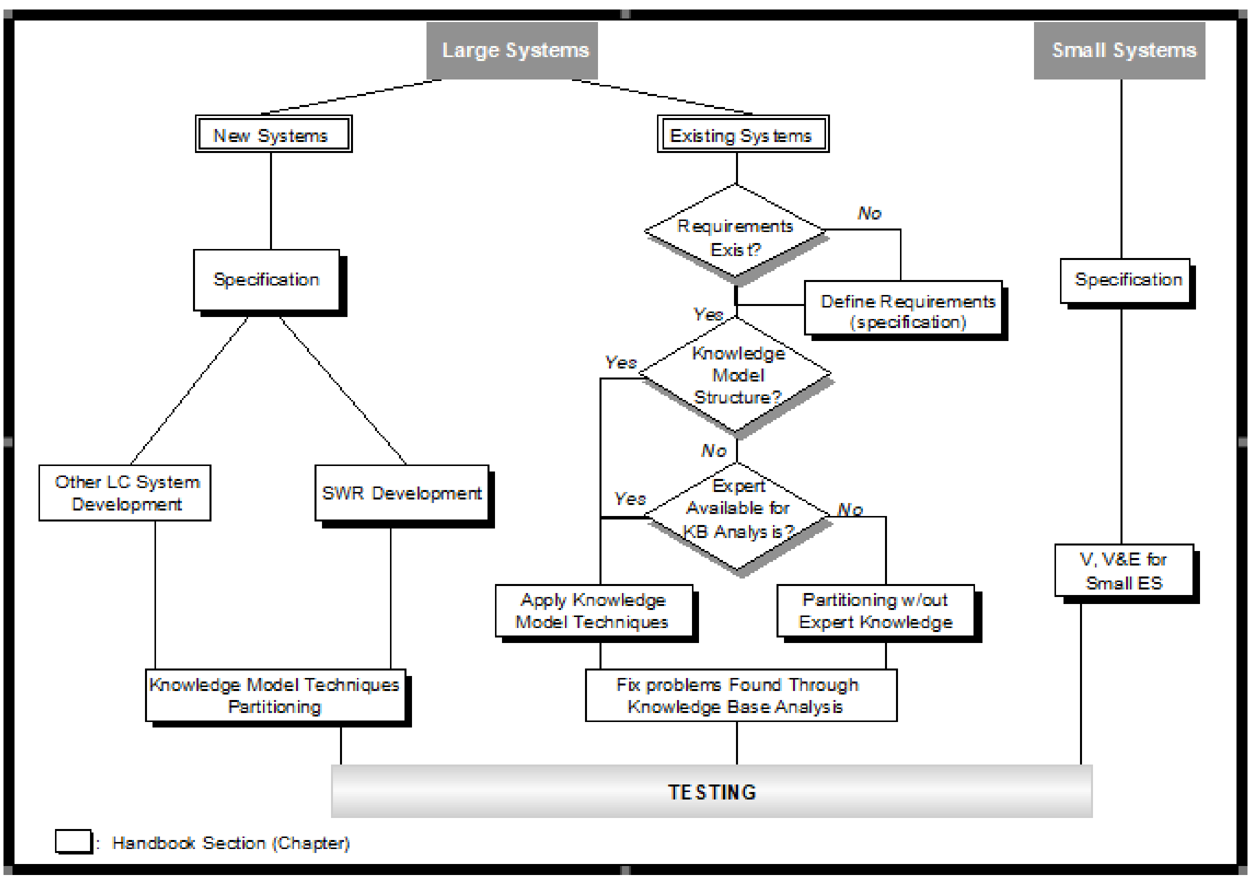


Figure 1.1: The V&V Process

Verification and Validation: Past Practices

The purpose of this chapter is to give an overview of past practices in VV&E. It includes a summary of validation and verification methods as well as a listing of V7V software and its corresponding references.

Knowledge-based verification and validation systems first appeared in the literature of the early 1980s. Most authors who have written about or attempted the verification and/or validation of knowledge-based expert systems have their own definition of the concepts. The method that they use or system that they design to accomplish the task(s) is a reflection of that particular definition. Only a few authors have asserted that verification and validation are the same concept.

The following tables summarize past work in verification and validation. Complete references appear in the bibliography.

Selected for the FHWA Handbook are those techniques which seem the most straightforward, precise and powerful in practice. Included are particular variations of partitioning, incidence matrices, and the use of metaknowledge (i.e., knowledge models).

VALIDATION METHODS THAT HAVE BEEN USED:

Table 2-1: Validation Methods

METHOD	EXPERT SYSTEM	REFERENCE
Turing Test Variation	Mycin KBSCD	Yu, et al., 1979 Agarwal, Kannan,Tanniru,1993
Simple Comparison with Expert	Diabetes Mellitus Tegument Hemody. Monitoring	Lehmann, et. al., 1993 Potter & Ronan, 1987 Koski, et. al., 1991/92
Comparison w/Expert Using Sensitivity Analysis	ESPE (Tool Set) Prospector	Franklin, et. al., 1988 Gaschnig, 1979
Comparison w/Expert Using Freq. Anal. & Distance Anal.	PNEUMON - IA	Verdaguer, et. al., 1992

VERIFICATION METHODS THAT HAVE BEEN USED:

Table 2-2: Verification Methods

METHOD	**TOOL (If Exists)**	**REFERENCE**
Tables & Pairwise Rule Comparisons	Rule-Checker Check	Suwa, Scott, Shortliffe, 1982 Nguyen, et al.,1987
Decision Tables of 'Contexts'	ESC GRAFCET	Cragun & Steudel, 1987 Renard, Sterling, Brosilow, 1993
Meta-Knowledge	EVA Valid	Stachowitz, Combs, 1987 Laurent (ESPIRIT-II)
Analytical Hierarchy Process		Bahill, Jafar, Moller, 1987
Graphs: Constraint Connection Flowgraph Parameter Dependency Network		 Freeman, 1985 Fenton, Kaposi, 1987 Agarwal, Tanniru, 1992
Petri - Nets		Agarwal & Tanniru, 1992 Liu & Dillon, 1991
Partitioning: Graph-Based Clustering Clustering Algorithm Category Partition Method Testing		 Jacob & Froscher, 1986 Cheng & Fu, 1985 Jacob & Froscher, 1990 Amla & Ammann, 1992
Incidence Matrix Technique	IMVER	Coenen, Bench-Capon, Kent, 1994
Ripple-Down Rules		Kang, Gambetta, Compton, 1994

Table 2-3: V&V Software

TOOL	PURPOSE	METHOD USED	REFERENCE
RITCaG	Validation	Test Case Generator	Gupta, Biegel, 1990
un-named	Validation	Runs Test Cases	Kang & Bahill, 1990
ESPE	Validation	Sensitivity Analysis	Franklin, et al., 1988
Check	Verification	Tables	Nguyen et al., 1987
ESC	Verification	Decision Tables	Cragun, Steudel, 1987
GRAFCET	Verification	Graphical Design Lang./Dec. Tables	Renard, Sterling, Brosilow, 1993
un-named	Verification	Decision Tables	Vanthienen,Dries, 1993
EVA	Verification	Meta-language	Stachowitz,Combs,1987
Valid	Verification	Meta-language	Jean-Pierre Laurent (ESPIRIT-II project) - Europe
BEACON	Verification	Graphs	Freeman, 1985
un-named	Verification	Layered Support Graphs	Valiente, 1992
VALIDATOR	Ver. & Valid.	Syntax & Semantics Checks	Jafar & Bahill
COVER	Verification	First-Order Logic	Preece, et al. 1992
Expert Choice	Verification	Analytical-Hierarchy Process	Bahill,Jafar, Moller, 1987
Spot	Verification	Prolog Rule Base	Lane, 1989
KB-Reducer	Verification	KB reduction	Ginsberg, 1988
IMVER	Verification	Incidence matrices	Coenen, Bench-Capon, Kent, 1994
un-named	Verification	Clustering Algorithm	Jabob & Froscher, 1990
in-progress	Ver. & Valid.	Meta-language, GUI, Visual Guide to Rule in Flow-Graphs	Traylor, Schwuttke, Quan, 1994 (JPL-NASA)

Planning and Management

The purpose of this chapter is to provide guidance on planning and decision making early in an expert systems project. This concept applies not only to new developments, but to thinking/improved decision making at any stage from development through implementation, this includes planning the verification, validation, and evaluation of an already developed system. The advice given here should aid in developing clear problem definition and thorough system requirements, reflecting realism from both technical and organizational viewpoints. Risk identification information is also provided..

Introduction

The development, testing and evaluation of an expert system is a demanding process. It is critical in the planning stages that the necessary resources are secured and that the proper development team is assembled. Both the successes and failures in the development of expert systems can usually be traced back to the planning phase of development. The following are important elements of a successful expert system development program:

- Management support in the institution sponsoring the development of the expert system is necessary. This support must include the commitment of both staff and financial resources needed to successfully develop and implement the system.
- The goal of the expert system and the exact uses of the end product must be clearly defined and understood by all involved from executive management to end users. Full knowledge and understanding of the costs and risks involved are also essential.
- Recognized experts in the appropriate technical fields must be available and have sufficient time committed to the expert system development project.
- Influential advocate(s) of the system are essential. Ideally, there should be advocates from both the technical development area and the end user community.
- The end users are pivotal to the development of expert systems and must be involved from the planning through the field evaluation stages. The end users provide definition of the skill level of the user community, information on how problems are addressed in practice versus the pre-scribed solutions, advice on how the system must function (interact with the user) to be accepted by the user community, and a cadre of supporters to test and promote the system once it is completed.
- Structured planning is recommended for the successful development of a system. This should include the problem/need to be addressed and the system benefits, organizational risk factors,

technical risk factors, and user risk factors. Development milestones must be identified and the system demonstrated at each milestone.

– Knowledge elicitation from the experts is vital throughout the duration of the expert system development. It is vital both in terms of building the system and for maintaining interest and continuity throughout the project.

– The verification, validation, and evaluation must be considered in all phases of the system development. Since some aspects of the verification, validation, and evaluation may not be performed by the developers, it is critical that VV&E plans be clearly identified and documented.

– Maintainability must be considered in all phases of the system development. Since the maintenance will probably not be performed by the developers, the system structure must be clear and straightforward. Logical and understandable names should be used for objects and knowledge structures within the system. Clear and complete system documentation is required for effective maintenance.

– The selection of the development tool for an expert system project should be performed by a qualified knowledge engineer or expert systems developer. This is critical because there are significant differences among the development tools. These differences are not explained in available literature and the application should dictate the selection of the development tool with its specific knowledge handling and operational characteristics.

Figure 3-1 shows the initial project planning process. This process can be applied to either a new development of the VV&E for an existing (but not adequately tested) system, or an existing system.

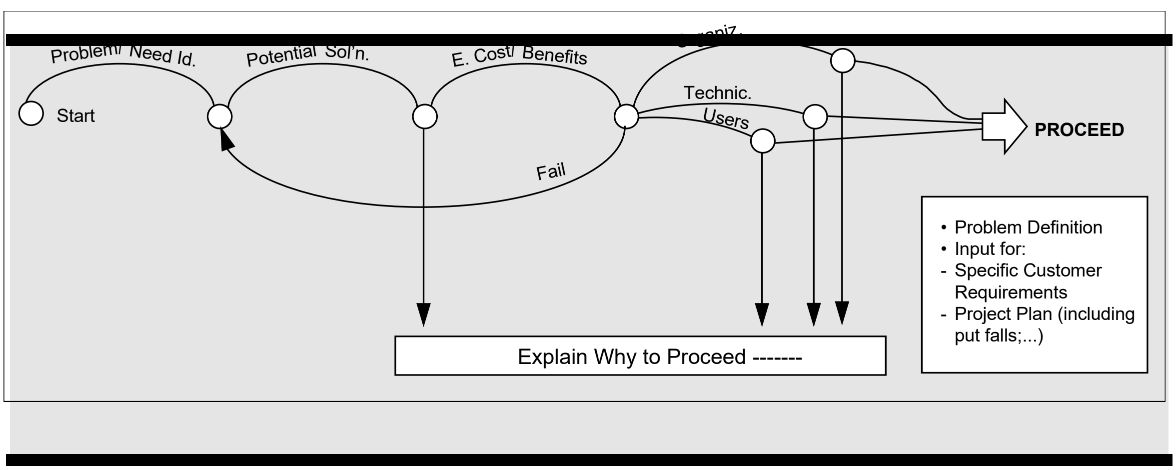


Figure 3-1: Initial Project Planning

Identify the Need for an Expert System

Before an expert system can be developed, the need has to be established and the problem to be addressed clearly identified and defined. It is strongly recommended that this be done in a structured study to include the following issues (Wentworth 1989):

- The problem/need to be addressed and the system benefits
- Organizational risk factors
- Technical risk factors
- User risk factors

NOTE: The term risk factors is used in deference to the old adage "if it can go wrong, it will go wrong." The risk factors represent areas where it "will go wrong" if there is any deficiency in planning and common sense.

Once a suitable problem domain has been defined for the expert system, the next task is to narrow the scope of the development effort by clearly defining the set of problems that the system will be expected to solve. The narrower the scope, the better are the chances that the expert system can be successfully built. However, if the scope is too narrow, the application becomes trivial. Judgment must be used in establishing the scope of the system as deterministic methods are not available. In general, it is better to err on the side of too narrow a scope rather than on too broad a scope. If the scope ultimately turns out to be too narrow, it may be relatively easy to broaden the scope by adding more knowledge to the knowledge base. However, if the development tool is too limited, it will be impossible to broaden the scope of the expert system by expanding the knowledge base. This highlights the importance of selecting the proper development tool to fit the particular problem.

Prior to embarking upon a expert system development effort, the expected benefits of such an effort must be clearly defined. There are two categories of benefits that are typically cited as reasons for developing an expert system. One category consists of concrete, quantifiable reasons such as savings of time and money, utility as a training tool, etc. The other category of benefits consists of tangible but not quantifiable reasons. Specifically, the process of developing an expert system will formalize and document the knowledge in a given problem domain, or combine and formalize the expertise from many experts in a given domain. This will result in expanded knowledge and better problem solving techniques in the domain, and provide a mechanism for giving this knowledge wide distribution to the users.

Under the heading of the **problem/need** to be addressed and system benefits, the following should be accomplished:

- The problem or need must be clearly identified and documented.
- The probability of the expert system resolving this problem or need must be described and quantified.
- The application or the output and the use of the output must be clearly defined .
- If standardization of results is desirable, the degree to which the expert system will improve standardization must be estimated.
- The use of the expert system to improve conditions by improving quality of results must be estimated.
- The expected utility of the expert system as a training tool must be described.
- End user involvement for the duration of the development process must be assured.
- Time and money savings based on the projected use of the expert system must be estimated.

Under **organizational risk factors**, suggested requirements and considerations are:

- There must be a dedicated and influential advocate who wants the system to be a success.
- There must be management support for the financial support, staff and time required to build the expert system.
- Management must have realistic expectations regarding the difficulty in developing the expert system.
- Management must have realistic expectations regarding the performance of the developed system.
- The results of the expert system must be applied without excessive management approvals being required.

Once a problem domain has been identified and the initial effort at narrowing the scope of the expert system application completed, the expert(s) whose expertise will be modeled must be selected.

The two main criteria that should be used to identify the expert(s) are:

1. The candidate(s) must be an expert in solving problems in the problem domain of interest and must be recognized as such by the potential user community. The need for the candidate to be an expert in the field is essential for the development of the expert system. The need for the expert to be recognized as such by the potential user community is primarily useful in selling the potential users on the viability of the given system as a useful problem solving tool for them.
2. The expert(s) must be dedicated to the successful development, testing, evaluation, and implementation of the system and be available and willing to spend the time (perhaps months) that will be required to accomplish this. The failure to identify such a person or persons and obtain a firm commitment means that the development project should not be undertaken.

Other useful characteristics for the domain expert(s) to have include the ability to communicate effectively, an orderly mind, patience and the willingness to teach.
In evaluating **technical risk factors**, the following should be included:

- There must be recognized experts in the field and there must be general agreement among these experts on the knowledge required to solve the problem the expert system is being developed to address.
- The development team must be identified and arrangements made to assure their dedication to the development and follow-up processes. The availability and personal commitment of all team members must be assured.
- The availability of a manual or automated procedure to be used as a model for the development of the expert system should be considered.
- The required performance of the expert system must be defined (in terms of finding the best solution as compared to senior experts). Unrealistic expectations must be avoided.
- Ambiguity in specifications must be avoided, or if ambiguity does exist, the specifications must be modified to avoid it.
- The scope and range of problems to be addressed by the expert system must be clearly identified.
- Interaction with external programs to run algorithmic routines or for data entry, etc., must be identified.

User risk factors must be considered and resolved in the initial planning phases of the expert system development. If representative end users are not involved in the planning and development stages, the system probably will not be accepted by the user community. Issues include:

- The end users must want the system and have a vested interest in its success.
- The computer proficiency and other skills and interests of the end users must be accommodated.
- The environment or conditions under which the system will be operated must be accounted for.

The Development Team

There are four categories of participants involved in the expert system building process. These are the advocate who champions the building of the expert system, the end users of the expert system, the domain expert(s) whose problem-solving expertise is to be modeled, and the knowledge engineer who actually builds the system. Although in the process of building a given expert system the same person may at various stages of development take on different roles, it is important to recognize that these roles are distinct.
The role of the advocate who champions the development of the expert system is to:

- Identify the need for the system.
- Define the problem domain.
- Identify the intended user community.

- Define the expected benefits that will accrue to the intended audience using the expert system.
- Identify the expert(s) whose expertise will be modeled.
- Choose the knowledge engineer who will develop the system.
- Maintain (or plan for the maintenance of) the finished product.
- Plan and chaperon the entire development process.

The end user is critical in the development of an expert system and must be involved in the entire development process. The end user provides:

- Definition of the skill level of the user community.
- Information on how problems are addressed in the field versus the prescribed solutions.
- Advice on how the system must function (interact with the user) to be accepted by the intended users.
- A cadre of supporters to test and promote the expert system once it is completed.

The domain expert has a dual role in the expert system development process. First, the expert's problem-solving ability serves as the model for the expert system. Second, the expert must assist in quality control on the project and make certain that the expert system faithfully represents a useful portion of the expert's knowledge. In essence, the expert must take some responsibility for insuring that the expert system faithfully models his expertise. The expert's major task in fulfilling this responsibility is to assist in the design of a comprehensive set of test problems for use in verifying that the expert system actually works.

The knowledge engineer has the task of developing a faithful model of the expert's problem solving ability in the domain of interest. Other tasks which the knowledge engineer must perform are:

- Implement the model of the expert's knowledge.
- Insure that the implementation is as transparent as possible.
- Document the expert system.
- Test the expert system.

One individual may perform more than one of these functions; however, the function of the end users should remain autonomous. If the roles of the domain expert and the knowledge engineer are combined, a second domain expert should review and confirm the technical findings.

The T / E Team

The same four categories of participants involved in expert system verification, validation, and evaluation are involved in the building of the system. However, their roles have changed in some aspects. These are the advocates who champion the building of the expert system, the end users of the expert system, the domain expert(s) whose problem-solving expertise is to be modeled, and the knowledge engineer who actually builds the system. Although in the process of building a given expert system the same person may at various stages of development take on different roles, it is important to recognize that these roles are distinct.

The role of the advocate who champions the expert system is to:

- Identify the need for system robustness and usefulness.
- Define the problem domain for testing.
- Identify the intended user community.
- Define the expected benefits that will accrue to the testers of the expert system.
- Identify the sites where testing will be conducted.

The end user is critical and must be involved in the entire process from development through implementation. The end user provides:

- Access to a cadre of supporters to test and promote the expert system.
- Information on how problems are addressed in the field versus the prescribed solutions and knowledge on how to "fix" problems on the fly.
- Advice on how the system must function (interact with the user) to be accepted by the intended users.

The domain expert has a dual role in the expert system development process. First, the expert's problem-solving ability serves as the model for the expert system. Second, the expert must assist in quality control on the project and make certain that the expert system faithfully represents a useful portion of the expert's knowledge. In essence, the expert must take some responsibility for insuring that the system faithfully models his expertise. The expert's major task in fulfilling this responsibility is to assist in the design of a comprehensive set of test problems for use in verifying that the system actually works.

The knowledge engineer has the task of developing a faithful model of the expert's problem solving ability in the domain of interest. Other tasks which the knowledge engineer must perform are:

- Implement the model of the expert's knowledge.
- Insure that the implementation is as transparent as possible.
- Document the expert system.
- Test the expert system.

One individual may perform more than one of these functions; however, the function of the end users should remain autonomous. If the roles of the domain expert and the knowledge engineer are combined, a second domain expert should review and confirm the technical findings.

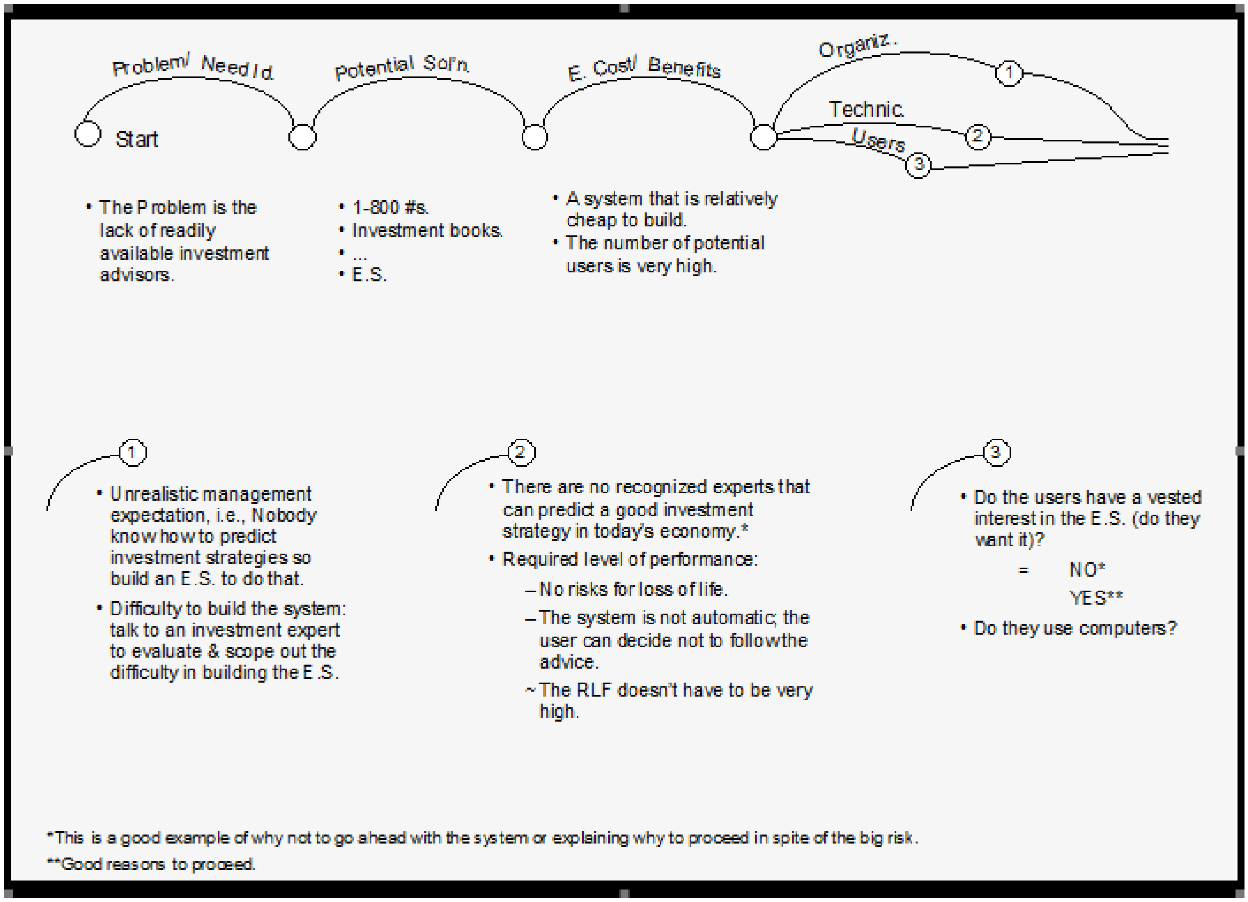


Figure 3.1.1: KB1 Initial Project Planning

Developing a Verifiable System

This chapter discusses how VV&E should be incorporated into the expert system lifecycle. Although some ideas may be used for revising and/or reengineering existing systems, this chapter is aimed mainly at designing new systems and ensuring that enough VV&E operations are done during the lifecycle so that this systems are verifiable. This includes decisions that should be made during system specification and verification/validation during stepwise development of an expert system.

Introduction

The proposed lifecycle for the development of expert (rule based and other) systems is a compilation of concepts taken from many sources (including lifecycle, cleanroom, ect.). The compiled system was organized and enhanced based on the experience of its developers to generate a basis for the development of "verifiable" systems. Even though the system allows for some flexibility in the degree of application of each of the system's components, the general outline has to be followed rigorously in order to achieve the objective outlined above.

The Concept: Figure 4.1. outlines the general concept for the development of a verifiable system. It includes the following stages:

Specification: This step is indispensable in the VV&E process.

Stepwise Development Process: This is one of the methods for the development process; other software development methods can be used as long as they include enough structure and verification steps.

Design (1): Start by designing the main parts of the expert system.

Verify (1): Verify that the design complies with the specification.

Implement (1): Implement (code) the first increment.

Verify (1): Verify that the implemented code complies with the design.

Design - Verify - Implement - Verify (2 to n): Loop through the entire process for the 2nd, 3rd, ... nth level until the entire system is complete.

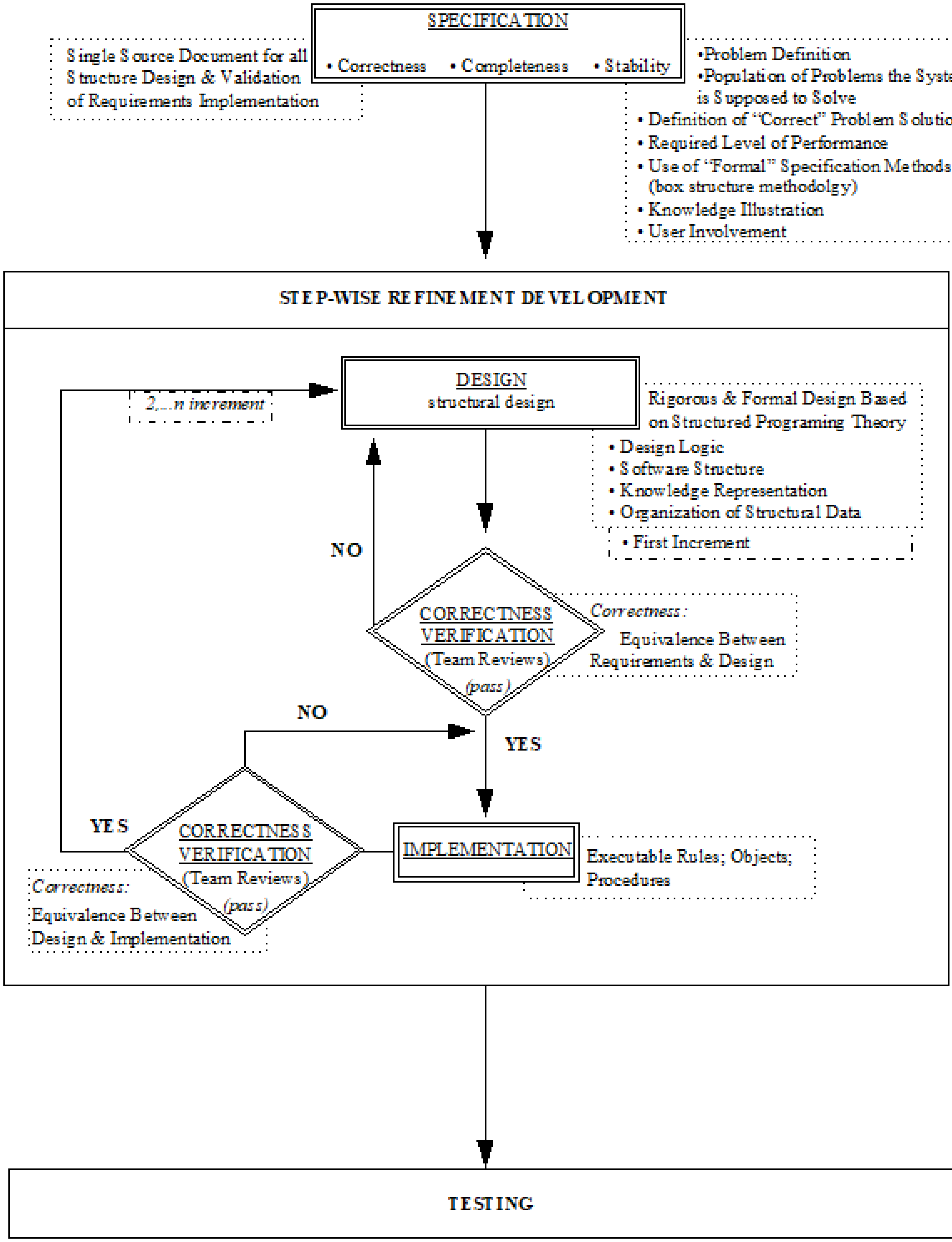


Figure 4.1: Developing a Verifiable System

Specification

The goal of this stage is to develop the system's specification.

input: software specific customer requirements.

output: software functional and performance requirements.

The Importance of Specifications

Specifications are important for VV&E. As noted in the introduction, *verification* determines if a system meets its specifications; this is meaningless if there are no specifications. *Validation* determines if a system does what is needed; this is only possible if it has been decided what a system is supposed to do. The results of these decisions are *specifications*.

At the specification stage the emphasis is on producing a clear identification of:

What is to be produced?

When to produce it?; and

What are the resources required?

The issue is to find a trade-off between the requirements specification (client) and the resources (time and money). The use of formal approaches (formal notation i.e., the Structured Analysis [SA; De Marco 1978], the Software Requirements Engineering Methodology [SREM; Alford 1978], the Structured Analysis and Design Technique [SADT; a trademark of SofTech],) proved to be very useful in this process. This is especially important to the V&V task because of the clarification provided by the use of these methods.

Functional Specification (FS): Specification of functions to be performed by the system and the constraints within which it must work.

Acceptance Test Specification: Test definition:

Who will perform the test(s)?

When (at what point)?

How do we insure that the system behaves according to the FS ? and

Include V&V Techniques to be used and when (at which time).

In addition to the above mentioned items, the following items should also be addressed:

A clear definition of the population of problems the expert system is supposed to solve.

A provision of test and development samples.

The required level of performance.

A clear definition of what constitutes a correct problem solution verification:

Is it possible to collect inputs that could possibly solve the problem?

Is it possible to compute the proposed output from the input validation?

Can the experts certify that the specifications, if properly implemented, would solve the problem evaluation?

Can experts judge that the system is worth the probable cost?

Can experts judge that the system would be useful in practice?

Is it possible to build a system that could be integrated with other components as necessary?

The General Form of Specifications

For the formal proof techniques presented in the following chapters, it is useful to have a general representation of a specification. Most specifications are based on the following form:

> For some subset S of the input space of an expert system, and for all X in S, the output of the system satisfies some proposition P.

Defining Specifications

It is particularly important to define specifications for the critical cases the expert system may encounter. A *critical case* for an expert system is a set or range of input data on which failure of the expert system to perform correctly causes an unacceptable, perhaps catastrophic, failure of the system of which the expert system is a part.

There are several steps in defining and verifying specifications for an expert system:

- Gather informal requirements from experts, with particular attention to defining the critical cases.
- Obtain expert certification of the specifications.
- Validate informal descriptions of the specifications with experts.
- Validate the translation of informal specifications into the formal notation used in the knowledge base.
- Validate the formal statement of the requirements using symbolic evaluation.

Each step is detailed in a section below, with particular attention to critical cases.

Gather Informal Descriptions of Specifications

The first step in verifying specifications is to gather a complete set of requirements. Only the domain expert(s) can provide this list. Ideally, during the original knowledge acquisition phase for the expert system, the knowledge engineer gathered, documented, and validated the critical cases. If the informally stated requirements are not available, however, gathering them is the first necessary step in verifying the correctness of an expert system.

Typically, to gather the critical cases, the knowledge engineer should ask the domain expert(s) to list critical cases, and to keep a careful record of them. As with most knowledge acquisition tasks, it is important to ask for the following information:

- General principles, e.g. "What are the critical performance requirements for this expert system?"
- Specific projects, and the critical performance requirements found in those projects. To get this information, the knowledge engineer should ask the expert(s) to tell him about their projects and experiences that are within the scope of the knowledge base. The purpose of this is that by reviewing the specific projects the expert's memory will be spur. This process will help the engineer to decide what the critical cases really are.

In gathering a set of critical cases, it is important to let the domain expert describe critical cases in their own words and notation, not in the notation of the expert system. This is because the expert system may have missed a critical variable that may be needed to recognize a critical case. If the knowledge engineer asks the expert to verify knowledge base gobbledygook, the expert may become too distracted to think of a critical case not described with the incomplete set of variables used in the incomplete knowledge base.

Obtain Expert Certification of the Specifications

It is important that the knowledge engineer elicit the expert(s) to certify the specifications,especially those concerning the critical cases. This is important, because the expert system will be built to meet and tested against the specifications. If the specifications are in error, the expert system will almost surely fail to perform properly.

In order to obtain meaningful certification of the specifications, the knowledge engineer must employ some means to focuses the expert on a careful review of the specifications. Among the ways to obtain this focus are:

- Have a group of experts reach consensus on the specifications. In this technique, the knowledge engineer functions as a moderator. In this role, the engineer will:
  - Be familiar with the ongoing discussion, and in addition, will be in a position to solicits important issues that must be resolved.
  - Insures that the experts address those issues and reach an agreement.
- Have the expert(s) sign off on the specifications.

Validating Informal Descriptions of Specifications

For systems where correct performance is critical, the next step in validating specifications of the expert system is to validate the informal descriptions of critical cases. The basic method for validation is that of cultural consensus, described in the chapter, "Validating Expert Knowledge". In this method, experts, ideally ones who have not provided the specifications, are used to validate the correctness of those specifications.

There are two questions that should be asked concerning the informal list of critical cases to validate: is the set of critical cases complete, and are the critical cases correct? To validate completeness, the knowledge engineer should conduct interviews with experts who have not contributed to the critical case list. This interview is like the one used to gather the list of critical cases, with one additional step: at the end of the interview, ask the expert to certify not just the critical cases the expert proposed, but the entire list of critical cases gathered so far (including those that were added during the interview). After additional experts no longer provide new critical cases, the list of critical cases has been validated to a confidence level depending on the number of experts who certify the list. Chapter 9, "Validating Underlying Knowledge", discusses these confidence levels in more detail.

Validating the Translation of Informal Descriptions

To validate the critical cases, the informal descriptions must be translated into formal statements in the language of the knowledge base. The goal of this translation is to produce statements of the form

if H1 and H2 ... and Hn then C1 and C2...and Cn.

The Hs should be stated in terms of input variables of the expert system, and the Cs should be possible conclusions of the expert system.

The translation into a knowledge base language is a process that can introduce errors. For example, for Knowledge Base 1 a critical case in the informal language of an expert might be, "If the client doesn't have a lot of money, they should first build a savings account". The closest that one can come to expressing this in the language of Knowledge Base 1 is

If "Discretionary income exists" = no

then investment = "bank account".

A financial planner would probably consider "Discretionary income exists" an inadequate translation of "the client doesn't have a lot of money"; Knowledge Base 1 does not even ask about existing savings or most other assets, for example.

As this example illustrates, the translation of expert knowledge into the formal knowledge language of an expert system is one of the tasks where errors can creep into the expert system. To have a truly validated expert system, the translation has to be validated. Although this is rarely done, items can be created for validation as follows:

Is <expert's statement of a critical case>

equivalent to <the same critical case in the knowledge language>

These items form the basis for a cultural consensus test for a set of knowledge engineers (see chapter 9 "Validating Underlying Knowledge"). When asking knowledge engineers to validate the translation of critical cases, it is important to:

- Use knowledge engineers who have not built the knowledge base.
- Give the validating knowledge engineers the opportunity to familiarize themselves with the knowledge language before examining the individual items.

In translating the informal requirements into formal knowledge base statements, there some typical kinds of errors; these are discussed below:

- False negatives in the input variables: One problem in knowledge translation results from the fact that a symptom is often used in a knowledge base to stand for an underlying condition; in the above example, for example, "no discretionary income" stands for "has no money". However, few observations are 100% reliable. If a single symptom is used to test for a condition in a knowledge base, a false negative of that symptom will produce an error in what the expert system does.

The solution to the false negative problem is to separate symptoms and underlying conditions in the knowledge base. If C is a condition, the knowledge base should contain a rule of the form

if S1 or S2 or ... Sn then C (Rule C)

where S1 through Sn are a set of symptoms such that the probability of false negatives in all the Ss is less than some agreed-on threshold. Outside of Rule C, and similar condition-inferring rules, the Ss should not appear when a condition (i.e., C) is intended. Therefore, every occurrence of an S outside of a condition-inferring rule should be validated by expert(s).
In the case where a single symptom has so low false negatives that it identifies C by itself below the acceptable error threshold, it is unnecessary to separate the symptom and condition in the knowledge base:

- Missing input variables: An expert learns to observe many symptoms of possible problems. An expert system may use only a small number of variables. Whether the small number of variables is adequate is a matter that experts must validate. It is important to ask experts what data they gather in looking at problems covered by the knowledge base. If the expert looks at more than the expert system, for example variable X, then:

Can the expert get along without <variable X>

is a knowledge item that should be validated (see chapter 9).

Validation of Formalized Requirements

At this point, the critical cases have been transformed into a set of statements of the form

if H1 and H2 and ... and Hn then C1 and ... Cm (name: f1)

Formal verification methods for specifications in this form are discussed in the chapters on knowledge modeling and verification techniques for small systems.

Figure 4.2 outlines the steps to be considered at the specification stage and figure 4.2.1 shows their implementation to knowledge base 1.

Other Issues to be addressed at this stage:

Project Plan: Breakdown of the work; manpower figures, milestones, ect.

Quality Management Plan: Quality Control.

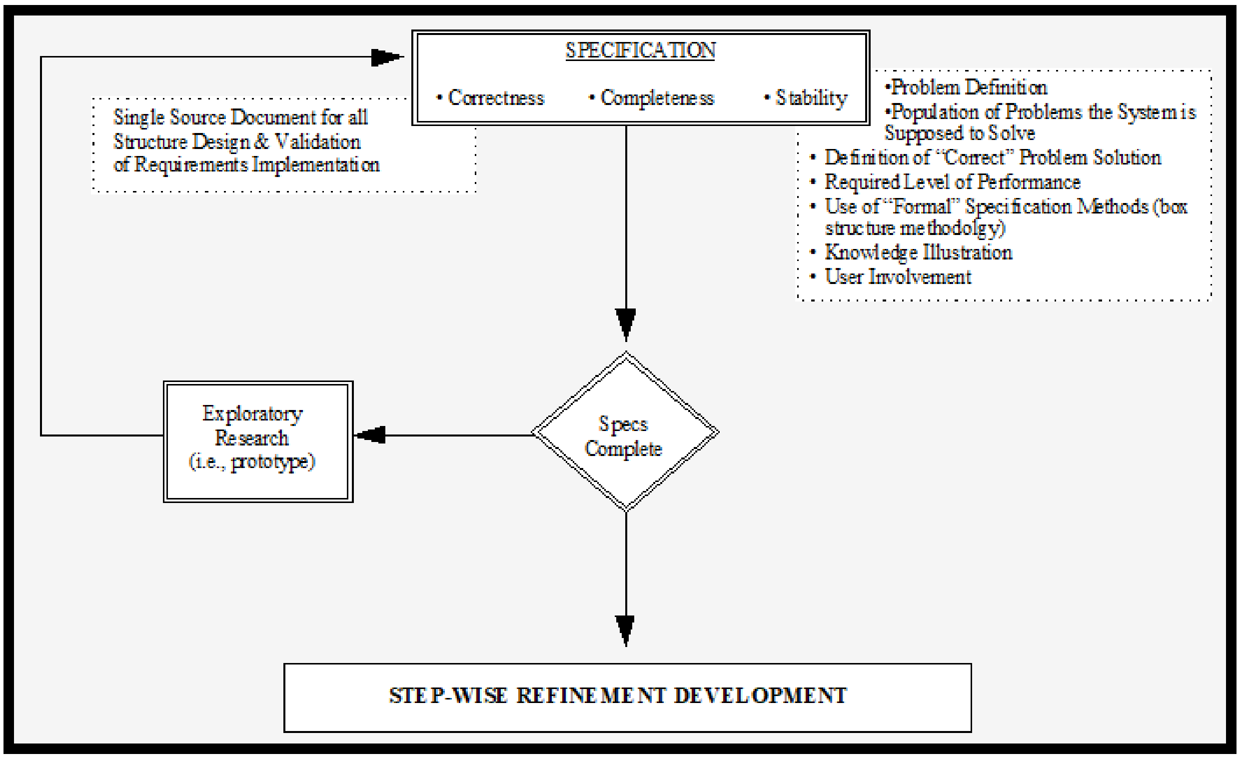


Figure 4.2: Specification

Problem Definition:

- The lack of readily available investment advise
- Develop a system that will advise the user on investment strategies.

Population of Problems the System is supposed to solve:

- Investment advise to people with less than $ 1 Million to invest.

Definition of Correct Problem Solution:

- An investment strategy is always suggested.
- Proposed solution should be affordable
- The investor is comfortable with the advise.

Required level of performance:

- As good as 70% of the expert(s) [Define: good 70% of the time or 70% as good all the time].
- The system should always recommend an affordable investment even if it has to be a conservative one.

Note: Knowledge Acquisition & User Involvement:

Figure 4.2.1: KB1 Specification

Step-Wise Refinement Development

At this stage, a mapping of the system functions (from FS) into software components will occur. This is also where the overall System Structure (Architecture) must be defined. The use of the following Box Structure Methodology will help tremendously in this process.

Box Structure Methodology:

Black Box: External view of the system. This provides a system description of the user visible system inputs and responses. No details on the internal structure and operations are provided.

State Machine: Intermediate system view. This decomposes the internal state structure from the BB description of the system.

Clear Box: Internal view of the system operations on inputs and internal state data.

Design

[(First level; First increment)]:

Input: System specification.

Output: System design document.

The general software architecture should consist of:

Software Components (for each software component, determine its purpose, functionality, interface, and data requirements),

- Structure & Flow.

If the box structure methodology, outlined above, is to be used, the first level/increment should consist of the overall design taken as deep as possible (using black boxes for functions and sub-systems). At every subsequent increment the design should be taken deeper (two to three level down) until all the boxes are replaced by their respective functions/subsystems.

From Specifications:
1. An Investment strategy is always suggested:
List of possible investments strategy for KB1:

- Stocks
- Saving Accounts
- Do Nothing (Although this is a good choice for many instances, it is not considered for the example)

Note: The list of output might be incomplete at this stage (i.e., may discover other possible strategies down the line).
Define the specifications in terms of these newly defined list of output.

2. Proposed solution should be affordable:

- When is stock affordable?
- When is Saving Account affordable?

Interaction with the expert(s)

Depending on the complex nature of the questions to be answered, we may find out that other things might be needed:

- Interaction with data bases
- Algorithmic routines
- Sub Expert systems

For KB1:
The expert determined that stocks are affordable if "Discretionary Income" exists.

We have to define "Discretionary Income" in a measurable manner.

From the interaction with the expert, we introduce the concept that in order to have "DI", the investor has to have:

- Some savings (> $ 3000.).
- A luxury item (Boat/ Luxury Car)

n.b.: 1. Keep careful records of interaction with the expert(s).
2. One of the products of these steps are expert(s) verifiable statements about the knowledge domain.

i.e., Stocks are affordable if there is savings and a luxury item.
These will be used for carrying out formal proof procedures. In a high risks situation (see table 4.1) these statements should be verified by enough experts to get the required level of confidence (see Chapter 9).
We have preliminary design information that consists of:

1. An expert sub-system to determine affordability
2. An expert sub-system for risk tolerance
3. An expert sub-system which makes an investment category decision using 1 & 2

n.b.: This is very useful for designing a well structured system.

Refer to Chapter 7, "Knowledge Modeling", and pick a knowledge model that fits the preliminary design information.

Figure 4.2.2: KB1 Design

Implementation

At the Implementation stage, the main objective is the creation of a complete executable system, including software to carry out all processes specified in clear or black boxes, according to constraints on those parts of the system. The system is comprised of executable rules, objects, procedures, etc., that:

- Satisfy requirements of the system as a whole .
- Are executable functions that are equivalent to abstract functions specified in the design. For example, the design may specify a function that determines that the user is rich. The implementation may check the bank account, kind of car owned, etc. However, it may not catch certain rich people because it does not check art owned. In this case, the implementation fails to carry out the abstract function required of it. In general, the computer bases a conclusion on less observed data than an expert, and simplifies the inference an expert makes to one that is just based on the small set of data the computer looks at.

The implementation stage should consist of the following steps:

1. Determine the high level structure of the system to be implemented.
2. Define communication between subsystems Implementation.
3. Detailed definition of subsystems.
4. Selection of implementation tool.
5. Implementation in the tool.

Correctness Verification

a. Design vs. Specification:

[if Design = Specification

then]

The overall result of this is a proof that any system that satisfies all the design documents is correct (i.e., complete, consistent, stable, satisfies requirements imposed by subject) provided that the parts not yet designed or implemented have properties as required by clear box theorems and the models of knowledge, or specified by the expert.

b. Code vs. Design:

[if Code = Design]

What has to be proved is the equivalence between requirements and implementation. (May use previous results together with proof of equivalence of design and implementation.) This may take the form of a cleanroom-type layered correctness proof, in which all boxes are clear and implemented, the top part of which was the previous proof of the equivalence of requirements and design.

then

Design

[(Second Level; Second increment; ... nth level; nth increment)]

else (re)Implement [(First level; First increment)].

Depending on the complexity of the problem and the consequence of failure, this process is to be accomplished by the developer(s) (Level I), the developer(s) and two members of the organization (Level II), or a separate verification team (level III). Table 4.1 is to be used as a guide in determining the level of the project. Figure 4.3 shows the process and figure 4.3.1 is the implementation to knowledge base 1.

Table 4.1: Level of Effort for the Correctness Verification Stage

	Consequence of Failure				
Complexity	**Loss of Life**	**Injury**	**High $$$**	**Inconvenience**	**Other(IC,...)**
Very Complex	III	III	III	II	I
Medium	III	III	II	I	I
Simple	III	III	II	I	I

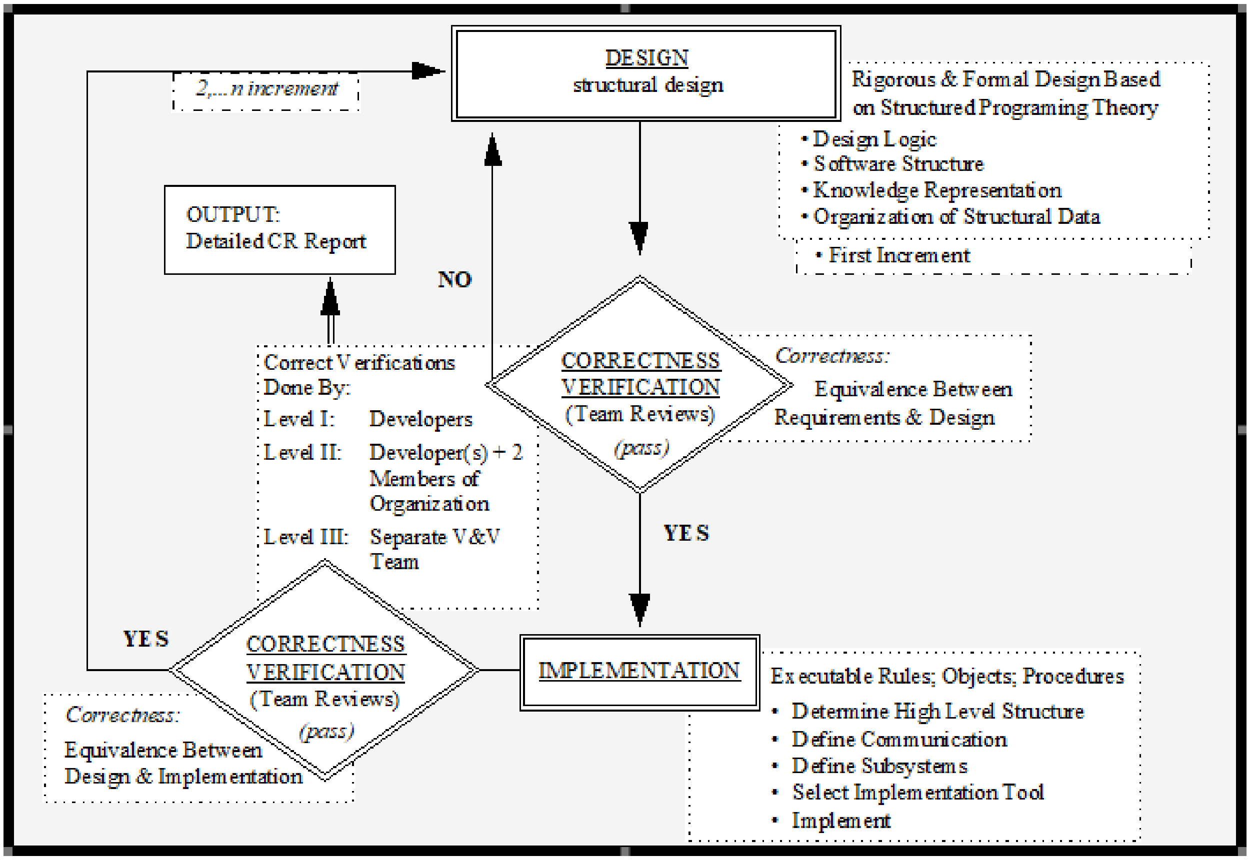


Figure 4.3: Correctness Verification

Step 1 -- Determine the high level structure of the system to be implemented: From the design stage, it was determined that the expert system consists of 3 subsystems, discretionary income (DI), risk tolerance (RT) and type of investment (INV). The structure of the system can be expressed by the function

Investment = INV(DI(boat, "luxury car", "savings account"),

RT(stocks, "lottery tickets"))

This expresses the fact that the output of DI and RT are inputs to INV.

Step 2 -- Define communication between subsystems: The output of DI and RT must be sufficiently fine-grained to distinguish cases where different investments are indicated. Since there are only 2 investments in this example system, only 2 values are required as output for each of these subsystems; we will use *high* and *low* for risk tolerance, and *yes* and *no* for discretionary income. At this point, the inputs, outputs and communication between subsystems have all been defined.

Step 3 -- Detailed definition of subsystems: In this stage, the expert information collected in the design step will be converted into precise logical statements; this process will be illustrated on the DI subsystem.

The condition that must be true to have discretionary income is

A = (Savings > $3000)(1)

AND ("Own Boat" = yes OR "Own Car" = yes)

The expert information about discretionary income can be formalized as

A IMPLIES ("discretionary income" = yes) (2)

NOT A IMPLIES ("discretionary income" = yes) (3)

Step 4 -- Selection of implementation tool: At this point, there is enough information to choose a tool in which to implement the expert system. The requirements on the tool are:

- Provide for communication between subsystems
- Express rules such as (2) and (3)

Most rule-based expert system shells meet these requirements. Although the order of information in the knowledge base must be slightly different in forward and backward chaining implementations, either form of inference engine can be used to implement this knowledge base.

Step 5 -- Implementation in the tool: The rule-based-shell implementation will be written in two steps: first as a generic rule-based implementation, finally as an implementation in CLIPS.

Step 5.1 -- A generic rule-based implementation: Rule-based shells typically allow menu, fill-in and yes-no questions. The following questions will gather the necessary information for discretionary income:

QUESTION TEXT	TYPE
What is your savings balance?	fill-in
Do you own a boat	yes-no
Do you own a luxury car	yes-no

The inputs and outputs can be represented inside the expert system by the following variables:

VARIABLE	TYPE	VALUE
savings	numerical	>= 0
"Do you own a boat"	boolean	yes or no
"Do you own a luxury car"	boolean	yes or no
"discretionary income"	enumerated values	high or low

Now we will put the knowledge in statements (2) and (3) into the rule form of rule-based shells. Rule-based shells encode information in the following form:

• Rules are of the form

IF <conditions> then <inferences> and <actions>

• <conditions> are built from simple requirements with the logical operations AND, OR and NOT.

• many of the simple requirements can be written in the forms such as VARIABLE = VALUE, or more generally

VARIABLE REL VALUE, where REL is one of the relations

=, >, <, >=. <=

• inferences can also be written in the form

VARIABLE = VALUE, i.e. VARIABLE is set equal to VALUE.

Actions are dependent on particular shells, and will be deferred at this time.

Using the above notation, (2) can be written as

IF (Savings > $3000) (4)

AND ("Do you own a boat" = yes

OR "Do you own a luxury car" = yes)

THEN "Discretionary income" = yes

(3) can be put into rule form as

If NOT <if part of (4) THEN "Discretionary income" = no (5)

Alternatively and more usually, a rule implementing (3) is written in a form in which the NOT is applied individually to the simple requirements contained in the "IF" part, rather than to a complicated expression built up from requirements. *DeMorgan's Laws* in mathematical logic,

NOT (A OR B) = NOT A AND NOT B (6)

NOT (A AND B) = NOT A OR NOT B

Using (6) repeatedly transforms (5) to

IF (NOT Savings > $3000) (7)

OR

(NOT "Do you own a boat" = yes

AND NOT "Do you own a luxury car" = yes)

THEN "Discretionary income" = no

Simplifying the simple conditions using the following relations,

(NOT Savings > $3000) = (Savings <= $3000)   (8)

(NOT "Do you own a boat" = yes)

= ("Do you own a boat" = no)

(NOT "Do you own a luxury car" = yes)

= ("Do you own a luxury car" = no)

Substituting (8) into (7) gives

```
IF (Savings <= $3000) (9)
      OR  ( "Do you own a boat" = no
      AND "Do you own a luxury car" = no )
THEN "Discretionary income" = no
```

Figure 5.1 shows an expert system in a generic rule-based shell language that implements the discretionary income, risk tolerance and investment subsystems. The result is a small knowledge base (called Knowledge Base 1) that implements the investment expert system. [Note: Knowledge Base 1 leaves out the savings requirement, to further simplify the example when it is used to illustrate verification and validation.]

Step 5.2 -- Implementation in CLIPS

Once a generic knowledge base has been written, it must be translated into the language of a particular shell. Shown below is an implementation of the generic knowledge base in CLIPS. The CLIPS is fairly close to the generic rule-based KB. The main differences are:

rule syntax: Rules in CLIPS have the following syntax:

```
(defrule <RULE NAME> <COMMENT>
 <LIST OF CONDITIONS>
 =>
 <LIST OF ACTIONS AND INFERENCES>
)
```

implementation of the AND operation: The AND operation can be implemented in two ways:

- a list of the conjuncts in the AND
- an explicit AND operation.

These alternative ways of writing AND are illustrated by the following two equivalent rules:

```
(defrule rule1  "stock"
 (risk_tolerance high)
 (discretionary_income TRUE)
 =>
 (assert (investment stocks))
 (printout t "We recommend stocks." crlf)
```

```
)
(defrule rule1  "stock"
 (and (risk_tolerance high) (discretionary_income TRUE))
 =>
 (assert (investment stocks))
 (printout t "We recommend stocks." crlf)
)
```

implementation of the OR operation: The OR operation can be implemented by an explicit OR operation, i.e.,

```
(defrule rule3c  "high risk tolerance"
 (or  (now_own_stocks TRUE )(lottery_tickets TRUE ))
 =>
 (assert(risk_tolerance high))
)
```

Equivalently, one can write a separate rule for each disjunct in the OR:

```
(defrule rule3c1  "high risk tolerance 1"
 (now_own_stocks TRUE )
 =>
 (assert(risk_tolerance high))
)
(defrule rule3c2  "high risk tolerance 2"
 (lottery_tickets TRUE )
 =>
 (assert(risk_tolerance high))
)
```

Here is an actual CLIPS implementation. This implementation is a fairly straightforward translation of the generic KB1. More sophisticated implementations of KB1 would structure the knowledge base so that when sufficient information for a conclusion had occurred, the user would be spared extra questions.

```
;
; KB1 in CLIPS,  a demo rule based system
;
;
; Note: In the following knowledge base,
; we will use certain user interface functions
; which can be defined in CLIPS:
;
; yes-or-no-p asks a yes-no question
```

```
; ask-parm asks a fill-in question
; ask-parm asks a menu question
;
; To run this CLIPS knowledge base, you need these functions
; which are not shown here.
;
;
; INVESTMENT TYPE SUBSYSTEM
;
;    Rule 1: If "Risk tolerance" = high
;      AND "Discretionary income exists" = yes
;         then investment = stocks.
;
(defrule rule1  "stock"
 (risk_tolerance high)
 (discretionary_income TRUE)
 =>
 (assert (investment stocks))
 (printout t "We recommend stocks." crlf)
)
;
;    Rule 2: If "Risk tolerance" = low
;      OR "Discretionary income exists" = no
;         then investment = savings account.
;
(defrule rule2a  "savings account 1"
 (risk_tolerance low)
 =>
 (assert (investment "savings account"))
 (printout t "We recommend a savings account." crlf) )

(defrule rule2b  "savings account 2"
 (discretionary_income FALSE)
 =>
 (assert (investment "savings account"))
 (printout t "We recommend a savings account." crlf) )
;
; DISCRETIONARY INCOME SUBSYSTEM
;
;    Rule 5: If
;         ( Savings > $3000)
;         AND ("Do you own a boat" = yes
;           OR "Do you own a luxury car" = yes)
;         then "Discretionary income exists" = yes.
;
; First we will gather the information
;
(defrule rule5a  "boat"
 (not (has_boat ? ))
 =>
 (bind ?boat ( yes-or-no-p "Do you own a boat? " ))
 (assert (has_boat ?boat  ))
)
;
(defrule rule5b  "luxury car"
 (not (has_lux_car ? ))
```

```
 =>
 (bind ?lc ( yes-or-no-p "Do you own a luxury car? " ))
 (assert (has_lux_car ?lc  ))
)
;
(defrule rule5c  "savings balance"
 (not (savings_balance ? ))
 =>
 (bind ?sb ( ask-parm "What is your savings balance? " ))
 (assert (savings_balance ?sb))
)
;
; Now we will use the information determining discretionary income
;
(defrule rule5d  "has discretionary income"
 (savings_balance ?sb)
 (test( > ?sb 3000))
 (or (has_lux_car TRUE  ) (has_boat TRUE))
 =>
 (assert (discretionary_income TRUE))
)
;
;    Rule 6: If Savings <= $3000
;             OR
;             ("Do you own a boat" = no
;               AND "Do you own a luxury car" = no)
;             then "Discretionary income exists" = no.
;
(defrule rule6  "has no discretionary income"
 (savings_balance ?sb)
 (test( <= ?sb 3000))
 (and (has_lux_car FALSE  ) (has_boat FALSE))
 =>
 (assert (discretionary_income FALSE))
)
;
; RISK TOLERANCE SUBSYSTEM
;
;    Rule 3: If "Do you buy lottery tickets" = yes
;      OR "Do you currently own stocks" = yes
;        then "Risk tolerance" = high.
;
(defrule rule3a  "lottery tickets"
 (not (lottery_tickets ? ))
 =>
 (bind ?Lt ( yes-or-no-p "Do you purchase lottery tickets ? "))
 (assert (lottery_tickets ?Lt ))
)
;
(defrule rule3b  "currently own stocks"
 (not (now_own_stocks ? ))
 =>
 (bind ?s ( yes-or-no-p "Do you currently own stocks ? "))
 (assert (now_own_stocks ?s ))
)
;
```

```
(defrule rule3c  "high risk tolerance"
 (or  (now_own_stocks TRUE )(lottery_tickets TRUE ))
 =>
 (assert(risk_tolerance high))
)
;
;    Rule 4: If "Do you buy lottery tickets" = no
;      AND "Do you currently own stocks" = no
;        then "Risk tolerance" = low.
;
(defrule rule4  "low risk tolerance"
 (and (now_own_stocks FALSE)(lottery_tickets FALSE))
 =>
 (assert(risk_tolerance low ))
)
```

Figure 4.3.1: KB1 Implementation

The Basic Proof Method

This chapter provides an overview of the basic method for formal proofs:

- Prove correctness on small systems by non-recursive means.
- Partition larger systems into small systems and the component systems are proved correct relations as required by partitioning theorems are proved among the components.

Introduction

An expert system is correct when it is complete, consistent, and satisfies the requirements that express expert knowledge about how the system should behave (see introduction for definitions). For real-world knowledge bases containing hundreds of rules, however, these aspects of correctness are hard to establish. There may be millions of distinct computational paths through an expert system, and each must be dealt with through testing or formal proof to establish correctness.

To reduce the size of the tests and proofs, one useful approach for some knowledge bases is to *partition* a knowledge base into two or more interrelated knowledge bases. For many, knowledge bases, this reduces the size of the VV&E problem.

Overview of Proofs Using Partitions

The basic method of proving that each of theswe aspects of correctness is basically the same. If the system is small, use a technique designed for proving correctness of small systems. If the system is large, apply a technique for partitioning the expert system and prove the required conditions for applying the partition to the system as a whole. In addition, insure that the correctness of any subsystem required by the partition. Once this has been accomplished apply this basic proof method recursively to the subexpert systems.

To carry out a partitioning of an expert system, one generally requires expert knowledge to define the top level problem-solving strategy of the expert system. In Chapter 7, "Knowledge Modeling", are presented a number of knowledge representations that may be useful in formalizing the top level structure of the knowledge base. Through knowledge acquisition with one or more expert, the top level structure of the knowledge base should be represented in a knowledge model. The correctness of this knowledge model should be validated with other experts or with standard reference materials in the target domain (the section in Chapter 9, on Validating the Semantic Consistency of Underlying Knowledge Items, addresses the problem of validating expert knowledge). When the formalization of the top level knowledge base has been so validated, the fact that the knowledge base has the validated structure can, from the standpoint of a formal proof, be assumed.

Once the top level structure of the knowledge base has been validated, to show the correctness of the expert system, the following criteria must be accomplished:

- Show that the knowledge base and inference engine implement the top level structure.
- Prove any required relationships among sub-expert systems or parts of the top level knowledge representation.

- Prove any required properties of the sub-knowledge bases.

Chapter 7, "Knowledge Modeling", discusses what exactly must be proved for various knowledge models and for various aspects of the correctness problem.

A Simple Example

To illustrate the basic proof method, Knowledge Base 1 will be proved correct in Figure 5.1. Although this knowledge base is small enough to verify by inspection, the proof will be carried out in detail to illustrate the proof method.

Knowledge Base 1

Rule 1: If "Risk tolerance" = high
AND "Discretionary income exists" = yes
then investment = stocks.

Rule 2: If "Risk tolerance" = low
OR "Discretionary income exists" = no
then investment = "bank account".

Rule 3: If "Do you buy lottery tickets" = yes
OR "Do you currently own stocks" = yes
then "Risk tolerance" = high.

Rule 4: If "Do you buy lottery tickets" = no
AND "Do you currently own stocks" = no
then "Risk tolerance" = low.

Rule 5: If "Do you own a boat" = yes
OR "Do you own a luxury car" = yes
then "Discretionary income exists" = yes.

Rule 6: If "Do you own a boat" = no
AND "Do you own a luxury car" = no
then "Discretionary income exists" = no.

Figure 5.1: Knowledge Base 1

Step 1 -- Determine Knowledge Base Structure

To prove the correctness of Knowledge Base 1 (KB1), the expert knowledge can determine that the system represents a 2-step process:

1. Find the values of some important intermediate variables, such as risk tolerance and discretionary income.
2. Use these values to assign a type of investment.

KB1 was built using this knowledge; therefore, it can be partitioned into the following pieces:

- A subsystem to find risk tolerance (part of Step 1).
- A subsystem to find discretionary income (part of Step 1).
- A subsystem to find type of investment given this information (part of Step 2).

To prove the correctness of a multi-step system, it must be proved that Step 1 satisfies the following criterias:

- For each set of inputs, all the outputs required by Step 2 are always produced by Step 1.
- For each set of inputs, all the outputs of Step 1 are single-valued.
- The correct outputs of Step 1 are assigned to each possible set of inputs.

It must also be proved for Step 2 that:

- For each set of inputs and computed Step 1 outputs, Step 2 produces some output.
- For each set of inputs and Step 1 outputs, all the outputs of Step 2 are single-valued.
- The correct outputs of Step 2 are assigned to each possible set of inputs and computed Step 2 outputs.

Step 2 -- Find Knowledge Base Partitions

To find each of the 3 subsystems of KB1, an iterative procedure can be followed:

1. Start with the variables that are goals for the subsystem, e.g., risk tolerance for the risk tolerance subsystem.
2. Include all the rules that set subsystem variables in their conclusions. For the risk tolerance subsystem, Rules 3 and 4 are included.
3. Include all variables that appeared in rules already in the subsystem and are not goals of another subsystem.

For the risk tolerance subsystem, include "Do you buy lottery tickets" and "Do you currently own stocks".

4. If all rules setting subsystem variables are in the subsystem, quit, else go to Step 2. For the risk tolerance subsystem, there are no more rules to be added.

Figure 5.2 below shows the partitioning of KB1 using this method.

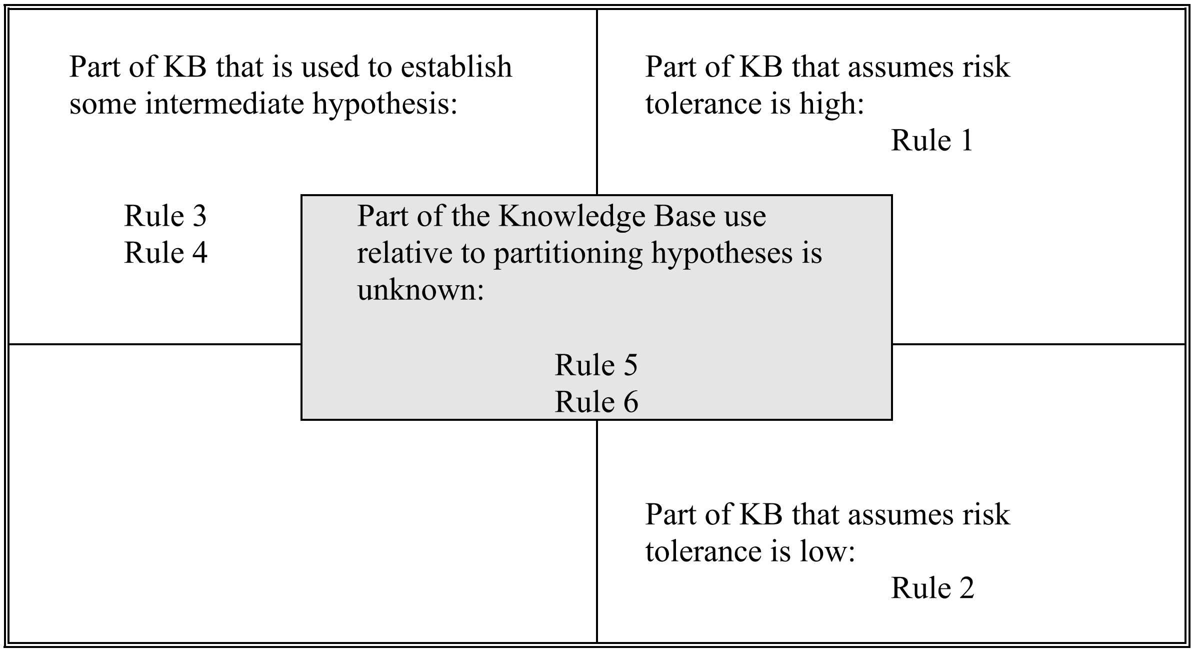


Figure 5.2: An Example of Knowledge Base Partitioning

Step 3 -- Completeness of Expert Systems

Completeness Step 1 -- Completeness of Subsystems

The first step in proving the completeness of the entire expert system is to prove the completeness of each subsystem. To do this, it must be shown that for all possible inputs, there is an output, i.e., the goal variables of the subsystem are set. This can be done by showing that the OR of the hypotheses of the rules that assign to a goal variable is true.

For example, the discretionary subsystem of KB1 will be shown to be complete. The discretionary subsystem consists of these rules:

Rule 5: If "Do you own a boat" = yes

OR "Do you own a luxury car" = yes

then "Discretionary income exists" = yes.

Rule 6: If "Do you own a boat" = no

AND "Do you own a luxury car" = no

then "Discretionary income exists" = no.

Step 3.1: The first step is to form the OR of the possible outputs of the system,

"Discretionary income exists" = yes (1)

OR "Discretionary income exists" = no

(1) expresses the condition under which some conclusion is reached.

Step 3.2: For each output condition in (1), the user substitutes the OR of rule hypotheses for rules that infer that condition. For example, for

"Discretionary income exists" = yes (2)

the only rule inferring (2) is Rule 5; its hypothesis is

"Do you own a boat" = yes (3)

OR "Do you own a luxury car" = yes

Since this is the only rule concluding (2), (3) is the OR of rule hypotheses inferring (2). Making the substitution of (3) for (2) in (1), and a similar substitution for

"Discretionary income exists" = no (4)

the result is

("Do you own a boat" = yes (5)

OR "Do you own a luxury car" = yes)

OR

("Do you own a boat" = no

AND "Do you own a luxury car" = no)

Step 3.3: Continue substitutions of the OR of rule hypotheses for inferred propositions (5) until the user obtains an expression in which only input variables appear. In fact, (5) already contains only input variables, and no further substitutions are needed.
Step 3.4: Apply Boolean algebra to simplify the expression from Step 3; the goal is to show that the Step 3 expression always has the truth value TRUE.
Letting

A = "Do you own a boat" = yes

B = "Do you own a luxury car" = yes

(5) can be rewritten as

(A or B) or (Not A and Not B) (6)

Simplifying this gives

(A or B) or (Not A and Not B)

= (A or B or Not A) and (A or B or Not B)

= true and true

= true

This means that the OR of conditions that infer *some* conclusion is true.

Completeness Step 2 -- Completeness of the Entire System

The results of subsystem completeness are used to establish the completeness of the entire system. The basic argument is to use results on subsystems to prove that successively larger subsystems are complete. At each stage of the proof, there are some subsystem known to be complete; initially, this is the subsystem that concludes goals of the expert system as a whole. At each stage of the proof, a subsystem that concludes some of the input variables of the currently-proved-complete subsystem is added to the currently complete subsystem. After a number of steps equal to the number of subsystems, the entire system can be shown to be complete.
When a complete subsystem that sets input variables of the currently complete subsystem is added to the currently complete subsystem, the augmented subsystem is complete. Any input to the augmented subsystem can be divided into a set V1 of input variables for the unaugmented system and a set V2 for the newly added subsystem. Note that some variables may be in both of

these sets. Since the newly added subset is complete, given V1, that subsystem produces output O1. However, O1 union V2 is an input for the unaugmented system, which, because of its completeness, produces an output. This shows that the augmented system is complete.

Since the number of subsystems is finite, this process of augmentation ceases after a finite number of steps. By mathematical induction, using an argument like that of the previous paragraph, it follows that the entire system is complete.

For KB1, this result can be applied, or alternatively make the following specific argument: Inputs to the system as a whole can be partitioned into inputs for the risk tolerance and the discretionary income subsystems. Each of these is complete, and so produces a risk tolerance and discretionary income, respectively. These are inputs to the investment subsystem, and are that system's only inputs. Since the investment subsystem is complete, it produces an investment. So an output for the entire system exists for each input, and the system as a whole is complete.

Step 4 -- Consistency of the entire system

The first step in proving the consistency of the entire expert system is to prove the consistency of each subsystem. To do this, the user must show that for all possible inputs, the outputs are consistent, i.e., that the AND of the conclusions can be satisfied.

For example, if an expert system concludes "temperature > 0" and "temperature < 100", the AND of these conclusions can be satisfied. However, if the system concludes, "temperature < 0" and "temperature > 100", the AND of these two conclusions has to be false. As a result, for the inputs that produced these two conclusions, it is not possible for all of the system's conclusions to be true at one time, and the system producing these conclusions is inconsistent.

Completeness Step 1 -- Find the Mutually Inconsistent Conclusions

The first step in proving consistency is to identify those sets of mutually inconsistent conclusions for each of the subsystems identified in the "Find partitions" step above.

Some sets of conclusions are mathematically inconsistent. For example, if a system describes temperature, the set

{"temperature < 0", "temperature > 100"}

is mathematically inconsistent.

However, other conclusion sets that are not mathematically inconsistent may be inconsistent based on domain expertise. For example, one investment advisor expert system could be designed to recommend several types of investments to each investor (probably not a bad idea). For such a system, "investment = stocks" AND "investment = bank account" are not inconsistent; stocks and bank accounts are just two of the investments recommended for some investor. However, if the system were designed to recommend only one investment per investor, "investment = stocks" AND "investment = bank account" would be interpreted as a contradiction, and the system recommending this would be inconsistent.

Because some sets of conclusions are inconsistent because of domain expertise, finding all sets of inconsistent conclusions generally requires expert knowledge.

Note that if there are no mutually inconsistent conclusions in the expert system as a whole, then consistency is true by default, and no further consistency proof is necessary.

Completeness Step 2 -- Prove Consistency of Subsystems

If there are inconsistent conclusions in the knowledge base as a whole, then the next step in proving consistency is to prove the subsystems consistent. This can be done by showing that no set of inputs to a subsystem can result in any of the sets of inconsistent conclusions. For each set of inconsistent conclusions, the user can construct, as detailed below, a Boolean expression B that represents all the conditions under which that set of inconsistent conclusions would be proved by the subsystem. If that Boolean expression can be shown to be FALSE, there are no such conditions.

Now the construction of the Boolean expression B to be proved false will be described. Let

S = {C1, ..., Cn}

be a set of potentially inconsistent conclusions for one of the subsystems.

B will be constructed by a backward chaining process, starting with

B0 = C1 AND ... AND Cn

Let Ci be one of the Cs. For all rules that conclude Ci, construct the OR of these rules initial conditions. Then substitute the resulting expression into B0.

Continue these substitutions until an expression results that has only the inputs to the expert subsystem. For each atomic Boolean expression A that is the conclusion of a rule in the subsystem, substitute the OR of the rule if parts of rules that conclude A. After at most a finite number of such substitutions, the user obtains an expression that expresses when all the Cs would be true in terms of the input variables of the subsystem.

For the risk subsystem, the only inconsistent set of rule conclusions is

S = { "Risk tolerance" = high and "Risk tolerance" = low }

The only initial conditions for "Risk tolerance" = high is from Rule 3,

"Do you buy lottery tickets" = yes

OR "Do you currently own stocks" = yes

and the only initial conditions for "Risk tolerance" = low is from Rule 4,

"Do you buy lottery tickets" = no

AND "Do you currently own stocks" = no

Let

A0 = ("Do you buy lottery tickets" = yes)

A1 = ("Do you currently own stocks" = yes).

This means

not A0 = ("Do you buy lottery tickets" = no)

not A1 = ("Do you currently own stocks" = no).

Using this notation,

B0 = (A0 OR A1) AND (NOT A0 AND NOT A1)

For this small subsystem, B0 is actually expressed in terms of inputs to the subsystem (i.e., B0 is actually B).

Distributing the top level AND over the OR,

B0 = (A0 AND (NOT A0 AND NOT A1))

OR

(A1 AND (NOT A0 AND NOT A1))

The first subexpression is FALSE because it contains A0 AND NOT A0. Likewise, the second is FALSE because it contains A1 AND NOT A1. Therefore, B0 is FALSE because it is the OR of only FALSE expressions.

Consistency Step 2 -- Consistency of the Entire System
The results of subsystem consistency are used to establish the consistency of the entire system. The basic argument is to use results on subsystems to prove that successively larger subsystems are consistent. At each stage of the proof, there are some subsystem known to be consistent; initially, this is the subsystem that concludes goals of the expert system as a whole. At each stage of the proof, a subsystem that concludes some of the input variables of the currently-proved-consistent subsystem is added to the currently consistent subsystem. After a number of steps equal to the number of subsystems, the entire system can be shown to be consistent.
When a consistent subsystem that sets input variables of the currently consistent subsystem is added to the currently consistent subsystem, the augmented subsystem is consistent. Any input to the augmented subsystem can be divided into a set V1 of input variables for the unaugmented system and a set V2 for the newly added subsystem. Note that some variables may be in both of these sets. Since the newly added subset is consistent, given V1, that subsystem produces and output O1. However, O1 union V2 is an input for the unaugmented system, which because of its consistency produces output. This shows that the augmented system is consistent.
Since the number of subsystems is finite, this process of augmentation ceases after a finite number of steps. By mathematical induction, using an argument like that of the previous paragraph, it follows that the entire system is consistent.
For KB1, this result can be applied, or alternatively make the following specific argument: Inputs to the system as a whole can be partitioned into inputs for the risk tolerance and the discretionary income subsystems. Each of these is consistent, and so produces a consistent set of risk tolerance and discretionary incomes, respectively. These are inputs to the investment subsystem, and are that system's only inputs. Since the investment subsystem is consistent, it produces a consistent investment. So an output for the entire system exists for each input, and the system as a whole is consistent.

The other subsystems of KB1 can be proved consistent in the same way.

Step 5 -- Specification Satisfaction
In order to prove that KB1 satisfies its specifications, the user must actually know what its specifications are. This is a special case of the general truth that in order to verify and validate, the user must know what a system is supposed to do. Specifications should be defined in the planning stage of an expert system project.
To illustrate the proof of specifications, it will be assumed that KB1 is supposed to satisfy:

A financial advisor should only recommend investments that an investor can afford.

As with many other aspects of verification and validation, expert knowledge must be brought to bear on the proof process. For KB1, an expert might say that anyone can afford a savings account. Therefore, the user only has to look at the conditions under which stocks are recommended. However, that same expert would probably say that just having discretionary income does not mean that the user can afford stocks; that judgment should be made on more than one variable. Therefore, it would be reasonable to conclude that KB1 does not satisfy the above specification.
However, if the expert does agree that the expert system observes all necessary inputs, one must use inputs to the expert system to express a specification. For KB1, this means that the specification is reexpressed as

KB1 recommends stocks only when there is discretionary income.
The user can prove this for the investment subsystem by assuming

NOT discretionary income

and proving

NOT stocks

The only rule that concludes stocks has "discretionary income" = yes in an AND in its "if" part. Therefore, the investment system satisfies the specification.

To prove the entire system satisfies the specifications, the user must look at the conditions under which "discretionary income" = yes is concluded from inputs for the system as a whole. A financial expert would surely say that owning a luxury car or boat does not mean that discretionary income actually exists, so the system as a whole fails the specification , what one would expect for a small example system tackling a complex subject.

Finding Partitions without Expert Knowledge

This chapter presents techniques for partitioning large expert systems when expert knowledge is unavailable.

Introduction

Generally, it is best to partition a knowledge base using expert knowledge. This results in a knowledge base that reflects the expert's conception of the knowledge domain. This, in turn, facilitates communication with the expert, and later maintenance of the knowledge base. Chapter 7, "Knowledge Modeling", presents techniques for partitioning using expert knowledge. Sometimes, however, it is not possible to obtain expert insight into a knowledge base. In this case functions and incidence matrices can be extracted from the knowledge base, and the information contained therein used to partition the knowledge base.

Functions

Expert Systems are Mathematical Functions

Expert systems are, among other things, complicated functions, in the mathematical sense of function. [By definition, a function is a set F of ordered pairs, such that if (a,b) and (c,d) are in F, and a = c, then b = d.] Less formally, a function is a single-valued mapping from an input space (called the domain) to an output space (called the range); i.e., there is only one value of the function for each point in the input space. For example, KB1 is a function that for each set of user data (i.e., amount of savings, personal property, etc.) assigns a type of investment.
The input variables to an expert system viewed as a function are the variables that are not computed inside the expert system, but are asked the user or looked up in a database. Variables that are inferred by rules or computed by functions in the knowledge base are not input variables. In KB1, for example, purchase of lottery tickets and ownership of boats and luxury cars are input variables, while risk tolerance and discretionary income are not. Tolerance and discretionary income, however, are inputs to the investment subsystem of KB1.

Propositions that are possible conclusions of the expert system are Boolean output variables of the expert system. Numerical or enumerated variables that are considered outputs of the expert system are also output variables. When viewed as a function, the value of an expert system is a vector of these individual output variables.

Partitioning Functions into Compositions of Simpler Functions

Functions can be written as compositions of simpler functions. For expert systems, two of the important relations that build more complex functions from simpler ones are cartesian product and function composition.

Cartesian Product

Suppose that an expert system made two different kinds of recommendations, e.g., a traffic management system that both set the timing of lights and controlled access to exit ramps. This expert system could be considered as a function E that computed light timing and onramp access from certain inputs, e.g.:

E(inputs) = (timings, access).

E could be split into two expert systems that computed these results separately:

E = (timings(inputs), access(inputs)) (6.2.2.1).

While some of the inputs and intermediate conclusions might appear in both subsystems, (6.2.2.1) decomposes E into two subsystems using the cartesian product operation. The cartesian product operation in this case takes the two separate conclusions, timings(inputs) and access(inputs) and builds the conclusions of E

(timings(inputs), access(inputs))

by putting the separate conclusions of the subsystems together in a fixed, predetermined order. More generally, if

(y1,...,ym) = f(w1,...,wk),

(z1...,zq) = g(x1,...,xn,),

then

(y1,...,ym, z1...,zq) = f(w1,...,wk) X g(x1,...,xn,),

where X is the cartesian product operator.

Applied to expert systems, this result means that if there is an expert system where input Ws are used to compute the conclusion Ys, and the Xs are used to compute the Zs, the system can be partitioned into subsystems

(y1,...,ym) = f(w1,...,wk),

(z1...,zq) = g(x1,...,xn,),

and the results concatenated together.

Function Composition

Function composition uses the results of an earlier function A as the inputs to a later function B to compute a single overall function C. This overall function is the result of :

1. Starting with the inputs to A.
2. Applying the function A to these inputs.

3. Applying B to the results of Step 2.

4. Using the results of step 3 as the value of C.

In the Pavement Maintenance Expert System (PAMEX), for example, various data items are used to compute the "Pavement Serviceability Index" and other measures of pavement life. The PSI and other similar parameters are then fed into a follow-up set of rules that choose appropriate maintenance procedures. PAMEX can be considered as a composition of the subsystem that computes indices with the subsystem that uses indices to compute appropriate maintenance procedures.

In mathematical notation, suppose the output of an expert system depends on a set of variables,

y1,...ym, i.e.

E = f(y1,...,ym)

In addition, suppose each of the y's is a function of some other variables, i.e.,

yi = gi(x1,...,xmi)

Then E = f(g1(x11,...,x1m),
g2(x21,...,x2m),
....
gn(xm1,...,xnm))

i.e., the expert system E is the result of applying the function f to the result of applying Gs to the input variables.

Note that which variables are functions of which others are properties of the expert system. This means that a function implemented by an expert system can not be arbitrarily rewritten as the composition of simpler functions. Instead, the choice of simpler functions is motivated by:

- Which variables are functions of which other ones in the expert system knowledge base.
- Which rewriting of the function computed by an expert system as the composition of functions reduces the size of the VV&E problem.

For KB1, investment is a composition of an investment function with risk tolerance and discretionary income functions:

investment(risk_tolerance("lottery tickets", "stock ownership"),
discretionary_income("boat", "luxury car"))

Dependency Relations

To find the functions embedded in a knowledge base, it is helpful to compute the dependency relation among variables.

Immediate Dependency Relation

The first step is to compute the immediate dependency relation. If X1 and X2 are variables in the knowledge base, X2 is immediately dependent on X1, if and only if, the following are true:

- X1 appears in an expression that computes X2.
- X1 appears in the if part of a rule that sets or concludes X2.
- X1 is an input to a function that computes X2.

The table below shows the immediate dependency relation for Knowledge Base 1. A1 appears in cell (I,Jj), if and only if, variable J is immediately dependent on variable I.

The immediate dependency relation for Knowledge Base 1 is shown in table 6.1.

Table 6.1: Immediate Dependency Relation for KB1

immediate dependency	LC	B	S	LT	DI	RT	INV
luxury car (LC)	0	0	0	0	1	0	0
boat (B)	0	0	0	0	1	0	0
stocks (S)	0	0	0	0	0	1	0
lottery tickets (LT)	0	0	0	0	0	1	0
discretionary income (DI)	0	0	0	0	0	0	1
risk tolerance (RT)	0	0	0	0	0	0	1
investment (INV)	0	0	0	0	0	0	0

The immediate dependency relation shows which variables influence the value of other variables through one level of computation (one rule inference or function computation) in the expert system.

Operations on Relations

Using the immediate dependency relation, one may compute the influences of variables may be completed through any number of levels of inference or function computation and composition. This requires union and composition relations, defined as follows:
Relation: A relation is, from a mathematical standpoint, a set of ordered pairs.
For example, the immediate dependency relation is shown as an ordered pair in figure 6.1:

{(LC,DI), (B,DI), (S,RT), (LT,RT), (DI,INV), (RT,INV)}
A pair (x,y) appears in the immediate dependency relation if and only if x influences the value of y.

Figure 6.1: Immediate Dependency Relation as Ordered Pairs

Domain: If R is a relation {x| for some y, xRy} is the *domain* of R. Some examples of domains are shown in figure 6.2.

Domain of the investment subsystem of KB1:

{ ("discretionary income" = yes, "risk tolerance" = high),

("discretionary income" = no, "risk tolerance" = high),

("discretionary income" = yes, "risk tolerance" = low),

("discretionary income" = no , "risk tolerance" = low)}

Domain of the immediate dependency relation for KB1:

{luxury car, boat, stocks, lottery tickets, discretionary

tolerance, risk tolerance, investment}

Figure 6.2: Examples of Domains

Range: {y| for some x, xRy} is the range of r. For example, the range of the investment subsystem of KB1 is { stocks, savings account}; the range of the immediate dependency relation is {0, 1}.

Composition: If R1 and R2 are relations, the relation (R1 o R2) is defined as follows: x (R1 o R2) z if and only if there is a y such that x R1 y and y R2 z.

For example, the composition of the immediate dependency relation of KB1 with itself is {(LC,INV), (B,INV), (S,INV), (LT,INV)}.

For an immediate dependency relation R among the variables of an expert system, (x,z) is in RoR if and only if there is a y such that (x,y) and (y,z) are in R; i.e., there is a variable y such that x influences y and y influences z. In other words, RoR shows the variables that indirectly influence another variable acting through a single intermediate variable.

Matrix representation: When range(R1) = domain(R2)
the composition operation R1 o R2 can be computed by matrix multiplication. A relation R is represented by a matrix M = {m(i,j)} if and only if

m(i,j) = 1 iff x R y where x is variable i and y is variable j

m(i,j) = 0 otherwise.

Table 6.1 shows the immediate dependency relation in matrix form.

If Mi represents Ri, B(M1 o M2) represents R1 o R2, where:

- M1 o M2 represents matrix product of M1 and M2.
- B(M) = {bm(i,j)} represents the Boolean operation on matrices, i.e.,

  bm(i,j) = 1 iff m(i,j) != 0

  bm(i,j) = 0 iff m(i,j) = 0.

Theorem 6.1: If R1 and R2 are immediate dependency matrices, B(M1 o M2) represents R1 o R2 when M1 represents R1 and M2 represents R2.

This theorem says that the representation of the indirect dependency relation with one intermediate variable can be computed by Booleanizing the matrix product of the immediate dependency matrix with itself.

Proof: Let M be the matrix that represents R1 o R2, based on a numbering of the relevant variables v1,...vn. The (i,j) entry of M is 1 if and only if vi influences vj. This means that there two sets of inputs where the vi's differ, and also where the results of applying (R1 o R2) to these inputs differ. On these two inputs, one of the inputs to R2 must vary on the two inputs; if no input to R2 varied, the output would also not vary on the two inputs.

Since at least one input variable to R2 varies when vi varies, let vk be such an input to R2. Since vk varies when vi varies, R1(i,k) = 1. Likewise, since vj varies when vk varies, R2(k,j) = 1. This means that:

the kth entry of row i = 1
the kth entry of column j = 1.

As a result, kth summand in the inner product

(Row i of M1) * (column j of M2) (6.3.2.1)

is 1. Since all entries of M1 and M2 are non-negative, the cartesian product (6.3.2.1) is non-zero. This means that (M1 o M2) has a non-zero (i,j) entry, so B(M1 o M2)(i,j) = 1. The upshot is that everywhere M is 1, B(M1 o M2) is also 1.

Now let (m,n) be a location in B(M1 o M2) which is 1. This will be true only if the (m,n) entry of M1 o M2 is nonzero. Since all entries of M1 and M2 are nonnegative, (M1 o M2)(m,n) > 0. This entry of M1 o M2 is the inner product

(row m of M1) * (column n of M2)

so the inner product is positive. This is possible only if there is a k such that the kth entry in each of these vectors is nonzero. This means that for some k, the kth entry of row m of M1 and the kth entry of column n of M2 are both 1, i.e.,

M1(m,k)=1
M2(k,n)=1.

This means that vm influences vk and vk influences vn. Therefore, vm influences vn. This shows that M, the representation of (R1 o R2), has a 1 wherever B(M1 o M2) has a 1. Combined with the earlier result, this shows that the two matrices M and B(M1 o M2) have the same set of 1's. Since both matrices have only 1 and 0 entries, the matrices are equal.

For example, in KB 1, B influences DI, as indicated by the 1 in the (B,DI) entry of the immediate dependency relation of KB1. In Table 6.1, this appears in the (2,5) location. Likewise, DI influences INV, and the (5,7) entry of the table is 1. This means that multiplying the table by itself, when the inner product of row 2 by column 7 is computed, the 1's in position 5 cause the inner product to be non-zero. This represents the fact that variable 2 (B) influences INV, variable 7, through the intermediary of variable 5, VI.

Table 6.2 shows the matrix product of the immediate dependency relation by itself. In this case, this is also the Boolean composition operation.

Table 6.2: Matrix Product of the Dependency Relation by Itself

immediate dependency	LC	B	S	LT	DI	RT	INV
luxury car (LC)	0	0	0	0	0	0	1
boat (B)	0	0	0	0	0	0	1
stocks (S)	0	0	0	0	0	0	1
lottery tickets (LT)	0	0	0	0	0	0	1
discretionary income (DI)	0	0	0	0	0	0	0
risk tolerance (RT)	0	0	0	0	0	0	0
investment (INV)	0	0	0	0	0	0	0

Power: If R is a relation,

R**1 = R

R**(n+1) = R o (R**n).

The power relation finds those variables which influence a variable through a chain of intermediate variables of some particular length. For R**n the chain of intermediate variables is of length n-1.

If M represents R and M**n is the product of n Ms, then B(M**n) represents R**n.
The previous table shows R**2 when R is the immediate dependency relation. Higher powers of the immediate dependency relation are empty (all zeros in the matrix representation).
Theorem 6.2: M**n represents the indirect influence of variables with n-1 intermediate variables.
Proof: Theorem 6.2 follows from Theorem 6.1 by mathematical induction.
Union: If R1 and R2 are relations with the same domain and range, the relation (R1 U R2) is the relation such that x (R1 U R2) y iff x R1 y or x R2 y.
The union of the immediate dependency relation and the composition of that relation with itself, for example is the relation D2 such that x D2 y iff either x directly influences y (y is immediately dependent on x) or x influences y through an intermediate variable.
Theorem 6.3: If Mi represents Ri, B(M1+M2) represents R1 U R2.
Proof: B(M1+M2)(i,j) = 1 iff M1(i,j) or M2(i,j). Iff x is the ith variable and y is the jth variable, M1(i,j) or M2(i,j) iff x R1 y or x R2 y, i.e.

x (R1 U R2) y.

Figure 6.2 represents

R U (R**2)

where R is the immediate dependency relation of KB1.
Accumulation: The accumulation operator R *a n is defined as follows:

R *a 1 = R

R *a (n+1) = (R *a n) U (R ** (n+1))

The accumulation R *a n of a relation finds all the variables that influence a variable through a chain of n-1 or fewer intermediate variables.
Theorem 6.4: R *a n represents the dependency relation between n-1 or fewer intermediate variables. If M represents R, B(M *a n) represents R *a n.
Proof: This follows from Theorems 6.2 and 6.3.
Dependency: The relations { lim R *a n } form an increasing sequence of relations, i.e., if (x,y) is in *a n, (x,y) is in *a m for m >= n. Therefore, the limit of this sequence as n --> infinity exists, and is equal to the union of the R *a n for all n. This limit will be called R*d.
Define the dependency relation D(R) as follows: x D(R) y iff the variable x influences the variable y. It is only possible for x to influence y if there is some (possibly empty) chain of intermediate, e.g., x, z1, ..., zn, y such that each variable influences its successor, i.e., each successive pair of variables is in the relation R. However, then x R**(n+1) y, so x R *a n y,

so x R*d y, and D(R) <= R*d.

However, if x R*d y, for some n, (x,y) R*a m for m > n (by definition of limit). Pick an m0 > n. Then x R*a m0 y, so for some m1 <= m0,
x R**(m1) y. Then there is a chain of m1+1 intermediate variables, z1,...zm1+1 such that x,z1,...,zm1+1,y is a sequence in which successive variables are in R, and R*d < D(R). Combining this with the previous result proves theorem 6.5.

Theorem 6.5: The limit R*d of the accumulation relations represents the dependency relation D(R).
Since both the sequences {B(M*n)} and {R*n} are monotone increasing and have only a finite number of possible values, each of these sequences is eventually constant. That constant is the limit of the sequence. Pick an n0 great enough so that each sequence has reached its limit. By Theorem 6.4, B(M *n0) represents R *n0 where M represents R. Since equal matricies represent equal relations, the limits can be substituted in this "represents" relation, proving
Theorem 6.6: The matrix lim(n->infinity)(B(M *a n)) represents D.
The dependency relation represents the relation that is true for all variables that influence a given variable, and false otherwise. Figure 6.2 is the accumulation of the immediate dependency relation of KB1. An entry in the table is 1 iff the variable on the right is dependent on a variable on the left.
To compute the dependency relation from the immediate dependency relation,

- Compute in sequence each R *a n.
- When the R *a n no longer change, the current R *a n is the dependency relation R*d.

Table 6.3. shows the dependency relation of the immediate dependency relation of Knowledge Base 1.

Table 6.3: Immediate Dependency Relation of KB1

	LC	B	S	LT	DI	RT	INV
luxury car (LC)	0	0	0	0	1	0	1
boat (B)	0	0	0	0	1	0	1
stocks (S)	0	0	0	0	0	1	1
lottery tickets (LT)	0	0	0	0	0	1	1
discretionary income (DI)	0	0	0	0	0	0	1
risk tolerance (RT)	0	0	0	0	0	0	1
investment (INV)	0	0	0	0	0	0	0

Finding Functions in a Knowledge Base

To carry out a partition of a knowledge base based on function composition, it is necessary to find functions embedded in the knowledge base. In particular, the goal is to find subsets SI and SO of the knowledge base variables such that the values of SO are a function of the inputs in SI and the variables in SI are used at most infrequently outside this function.

Choosing the Output and Input Variables of a Function

Each column vector in the dependency relation matrix shows which variables influence a variable. For example, the first 4 columns of the dependency matrix for KB 1 are all 0s, because these are input variables and are not influenced by any other variables in the KB. Discretionary income (DI) has 1's for the two variables that influence it, boat and luxury car. Investment has nearly all 1's, because all variables except itself influence its value.

To find the set of variables whose cartesian product will be the output of a function in the KB, cluster via high correlation the column vectors in the table. The clusters should be performed such that all members of a cluster are highly correlated with each other. This says that all the variables computed by a function use about the same set of input variables.

The variable clusters of the dependency relation of the immediate dependency relation of Knowledge Base 1 are:

{luxury car, boat}
{stocks, lottery tickets}
{discretionary income, risk tolerance}
{investment}

Once a set of output variables has been chosen, the set of input variables for the function consists of the union of all variables for each member of the output variable set. Table 6.4 shows variable clusters of the dependendency relation of KB1.

Table 6.4: Variable Clusters of the Dependency Relation of KB1

VARIABLE CLUSTER	INPUT VARIABLES
{LC, B, S, LT}	none
{DI}	{LC,B}
{RT}	{LT,S}
{INV}	{DI,RT}

Finding the Knowledge Base that Computes a Function

The knowledge base that computes a function, whose input and output variables have been found in the last section, can be computed by a matrix technique like that explained above. Let S(KB) be the set of all objects in the knowledge base of KB, i.e., all rules, functions and variables. The variables in S(KB) include all KB conclusions, all Boolean atomic formulas appearing in rules, and all variables asked the user, returned by functions, or found in some database.
The immediate dependency relation R is defined on S(KB) as follows:

- If Rule1 is a rule in KB and x is a variable appearing in the if part of Rule1, x R Rule1.
- If Rule1 is a rule that sets a variable y,

  Rule1 R y.
- If f is a function that sets or outputs a variable y,

  f R y.
- If x is a variable that is an input of a function f,

  x R y.
- If b is a Boolean atomic formula containing a variable x,

x R b.

Dependency Relations of Rules on Variables in Knowledge Base 1

In KB1, all atomic formulas set by the knowledge base are of the form

VARIABLE = VALUE

When this is the case, the immediate dependency of variables and rules is sufficient to obtain the dependency among variables. Table 6.5 shows how variables influence rules.

Table 6.5: How Variables Influence Rules

	R1	R2	R3	R4	R5	R6
LC					1	1
B					1	1
S			1	1		
LT			1	1		
DI	1	1				
RT	1	1				
INV						

Dependency Relations of Variables on Rules in Knowledge Base 1

Table 6.6 shows how rules influence variables.

Table 6.6: How Rules Influence Variables

	LC	B	S	LT	DI	RT	INV
R1							1
R2							1
R3						1	
R4						1	
R5					1		
R6					1		

Dependency Relations of Variables on Variables in Knowledge Base 1

Multiplying A*B creates the matrix showing how each variable influences others. Positive numbers in cell (R,C) indicate that the variable in row R influences the variable in column C. Making this into a Boolean matrix yields the immediate dependency matrix for variables in KB1. Tablw 6.7 shows the immediate dependency matrix for KB1.

Table 6.7: Immediate Dependency Matrix for KB1

	LC	B	S	LT	DI	RT	INV
LC	0	0	0	0	2	0	0
B	0	0	0	0	2	0	0
S	0	0	0	0	0	2	0
LT	0	0	0	0	0	2	0
DI	0	0	0	0	0	0	2
RT	0	0	0	0	0	0	2
INV	0	0	0	0	0	0	0

Using the extended immediate dependency relation R just defined, the user can compute a sub-knowledge-base that is sufficient to compute a set of variables. Let SO be a set of output variables for a function f, chosen as discussed in the previous section. Let RR be either one of the R *a n or the relation R *d. Then the sub-knowledge base that computes f is defined by

x is in Sub_KB(f) iff x RR y for some y in SO.

Hoffman Regions

For logical completeness and consistency of an expert system, an important concept is the Hoffman regions (suggested by Roger Hoffman of FHWA). If V1...Vn are the variables of a knowledge base, with domains D1...Dn respectively, a Hoffman region is a maximal subset of the input space, the cartesian product D1x...xDn, on which each atomic formula in the knowledge base has a single truth value. For any knowledge base, there is a unique set of Hoffman regions that cover and partition the input space.

A run of an expert system is completely determined by the values of the atomic formulas that appear in the KB rules. Provided that the expert system does not use external numerical software, there is no need to run two different test cases that evaluate the same on all the atomic formulas. If two different test cases evaluate some atomic formula differently, however, the firing of some rule, and hence the results of the expert system, may differ between the two test cases. Therefore, the set of test cases that must be tested are in 1-to-1 correspondence with the regions where all the atomic formulas have the same value. These regions where the atomic formulas are the same are called Hoffman regions.

Each point in input space determines truth values for each of the atomic formulas in the knowledge base. A relation H(P1,P2) can be defined on input point spaces as follows: H(P1,P2) is true if and only if P1 and P2 determine the same set of atomic formula truth values for all atomic formulas in the KB. H so defined is an equivalence relation, and partitions the input space into mutually disjointed regions that cover the input space.

It is generally not possible to find simple, exact descriptions for all the Hoffman regions when a knowledge base contains atomic formulas that contain several variables, e.g., exp(X)<Y^3. It is possible, however, to find an approximate set of Hoffman regions of descriptions such that:

- Every Hoffman region is in the approximate set of Hoffman regions.
- A member of the approximate set of Hoffman regions is either a Hoffman region, or is the empty set, i.e. is an empty region of input space.

The set of possible Hoffman descriptions D can be computed as follows:

1. For atomic formulas containing two or more variables, the Hoffman regions of these atomic formulas are TRUE and FALSE.
2. Sort all the atomic formulas containing only one variable into subsets, putting all the formulas containing the same variable together.
3. Normalize formulas containing relation operators so that the variable appears on the left.
4. Lexically sort the formulas for each variable as follows:
   - The major sort is by the right side of the formula.
   - The minor sort is by relational operator, where the relation operators in ascending order are: <, <=, =, >=, >.
5. Create a set of intervals for each numerical variable that:
   - Cover the real line, or at least the possible domain of the variable.
   - For all points in any interval, the truth values of the atomic predicates (of that single variable) are the same.
   - The intervals are maximal, given the truth value constraint.
6. For each string variable, let the Hoffman regions be the list of values that appear in the KB.
7. Let the Hoffman regions of the KB as a whole be the cartesian product of the Hoffman regions for the individual variables.

Note that in KBs with atomic formulas with more than one variable, the use of TRUE or FALSE as the Hoffman regions is a compromise to avoid having to decide exactly when combinations of these formulas are true, an in general undecidable problem. This means that some Hoffman regions may be unsatisfiable. Therefore, if exhaustive testing shows an inconsistency in some Hoffman region which is partly defined by atomic formulas of more than one variable, there are two possibilities:

- The Hoffman region is unsatisfiable, so the expert system is OK.

- the Hoffman region is satisfiable, and the expert system has an inconsistency.

If a Hoffman region is found where the expert system is inconsistent, it should be determined whether the Hoffman region is satisfiable. Table 6.8 illustrate this concept below.

The Hoffman Regions of KB1

Table 6.8: Hoffman Regions for KB1

LC=yes B=yes LT=yes S=yes	LC=yes B=yes LT=yes S=no	LC=yes B=yes LT=no S=yes	LC=yes B=yes LT=no S=no
LC=no B=yes LT=yes S=yes	LC=no B=yes LT=yes S=no	LC=no B=yes LT=no S=yes	LC=no B=yes LT=no S=no
LC=yes B=no LT=yes S=yes	LC=yes B=no LT=yes S=no	LC=yes B=no LT=no S=yes	LC=yes B=no LT=no S=no
LC=no B=no LT=yes S=yes	LC=no B=no LT=yes S=no	LC=no B=no LT=no S=yes	LC=no B=no LT=no S=no

When is a Partitioning Advantageous

Let CH(KB0) be the cardinality of the Hoffman region set of knowledge base KB0. The worst case in proving a result on a knowledge base KB with sub-KB KB1 is, using the result of the previous section, CH(KB1) + CH(~KB1). If this number is significantly smaller than CH(KB), the partitioning pays off in reducing the size of a VV&E problem.

Hoffman Regions of Partitioned KB1

The KB can be split into the following pieces:

Final conclusion KB: This contains rules 1 and 2, and determines the type of investment.

Risk tolerance KB: This contains rules 3 and 4, and determines the comfort level of the client regarding risk.

Discretionary income KB: This contains rules 5 and 6, and determines whether the client has discretionary income.

Each of these KBs has two input variables each with two values, or four Hoffman regions. Therefore the total number of Hoffman regions after partitioning is twelve, a 25% reduction. A greater reduction is found in many larger knowledge bases.

ILLUSTRATIONS OF KNOWLEDGE BASE 1

The knowledge base 1 (KB1) has six rules. There are seven variables which can take two possible values. It is, therefore, a seven dimensional, binary problem. Let's focus on rule 3 to understand the illustrations of KB1. It has two hypotheses, and one conclusion. The hypotheses are "Do you buy lottery tickets?"="yes", and "Do you currently own stock?"="yes". They are associated with the logical operator "or". The consequent is "Risk Tolerance"="low".

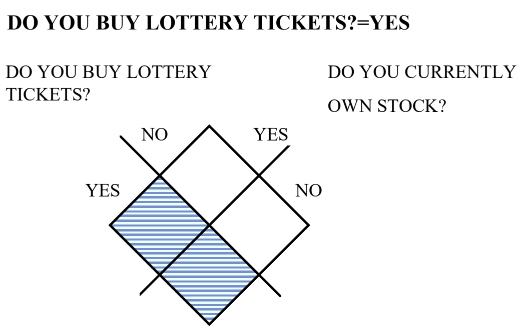


Figure 1

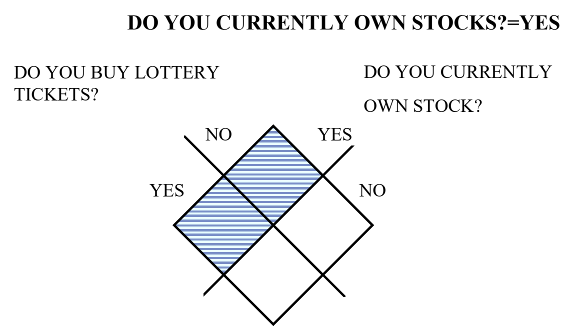


Figure 2

For the two variables of the hypotheses in rule 3, there are two possible values: "yes" or "no". The number of possible combinations of values for the variables is four. These four combinations appear in figure 1 as four square regions defined by the closed boundary (defining the domain of the variables) and the line boundaries separating the possible values for each variable. Each square is a Hoffman region.

If variable "Do you buy lottery tickets? " is assigned a value "yes", then two of the four regions are relevant. In figure 1, they are shown with a hatch. The two regions corresponding to hypothesis "Do you currently own stock?"="yes" are hatched in figure 2.

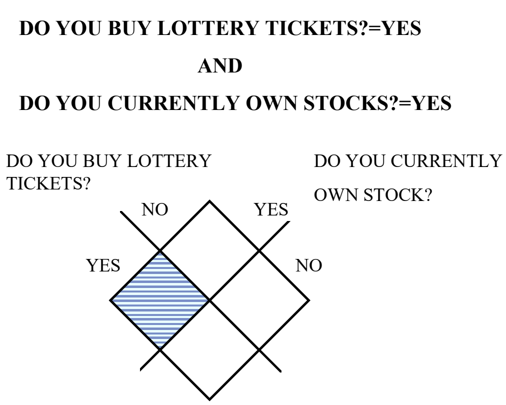


Figure 3

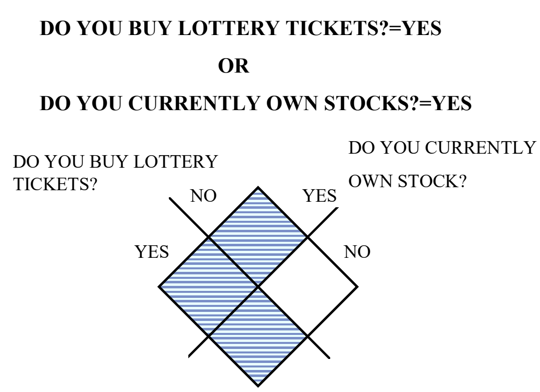


Figure 4

Figures 6-1, 6-2, 6-3, 6-4

In two dimensions, a Hoffman region is a surface as shown in this example. In three dimensions, it would be a volume, ect...

The logical operators are "and", "or", and "not". The last one is obvious in the case of a binary system: "not""yes"="no". In figure 1 and 2, the Hoffman regions corresponding to each hypothesis of rule 3 are hatched. When combined with an "and" logical operator, the intersection of the two sets of Hoffman regions that logical expression. It is shown in figure 3. The intersection in this case is a unique Hoffman region.

In rule 3, an "or" logical operator connects the two hypotheses. In this case, the union of the two sets of Hoffman regions is taken , as shown in figure 4.

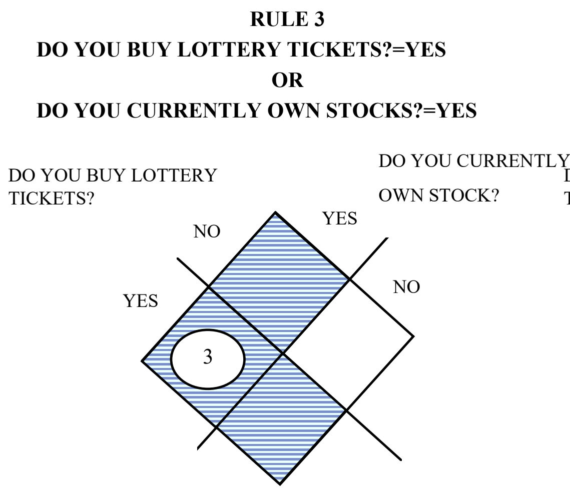


Figure 5

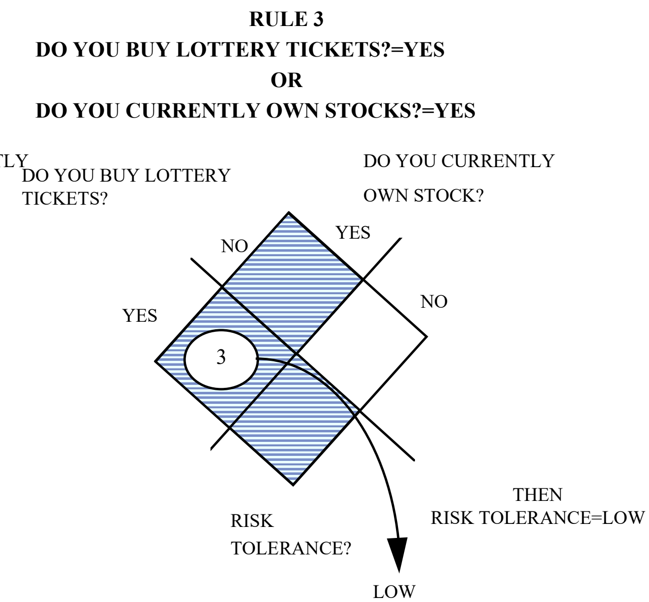


Figure 6

Next, the region defined by the logical expression of hypotheses is labeled with its rule number. For rule 3, the three Hoffman regions are labeled with a circled 3, as shown in figure 5. The consequent for the rule is linked to the label of the region of hypotheses. In figure 6, a curved arrow starts at the circled 3, and ends at the value "low" of the variable "Risk Tolerance".

Figures 6-5, 6-6

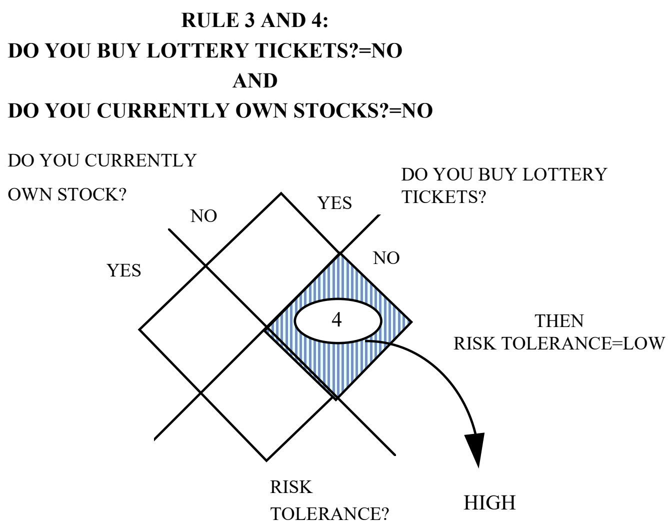


Figure 7

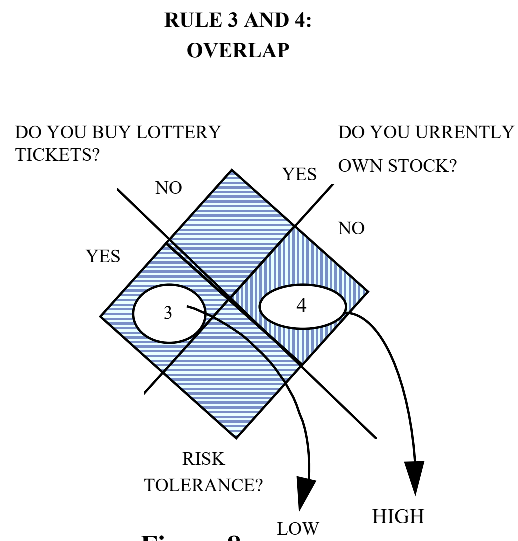


Figure 8

At this point, each rule can be represented using this scheme. Rule 4 has the same variables in its hypotheses and conclusions. Figure 7 shows the graphical representation of rule 4, and figure 8 shows rules 3 and 4 together.

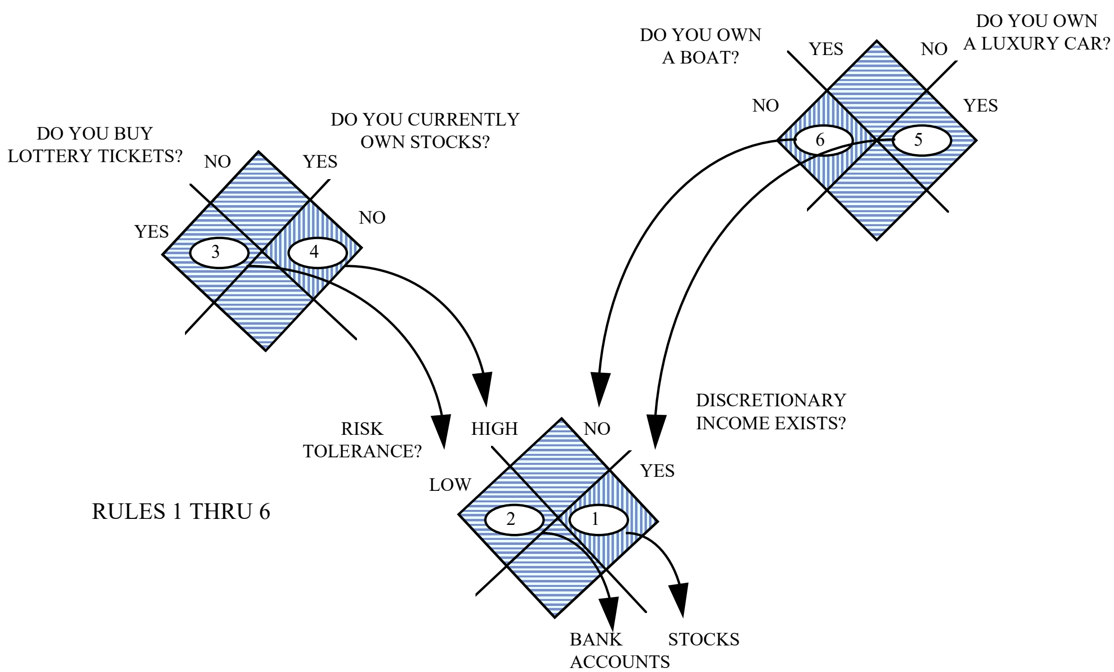


Figure 9

All six rules are shown in figure 9. Note that three clusters of rules become apparent: {R3, R4} in the upper left corner, {R5, R6} in the upper right corner, and {R1, R2} in the lower center of the figure..

Figures 6-7, 6-8, 6-9

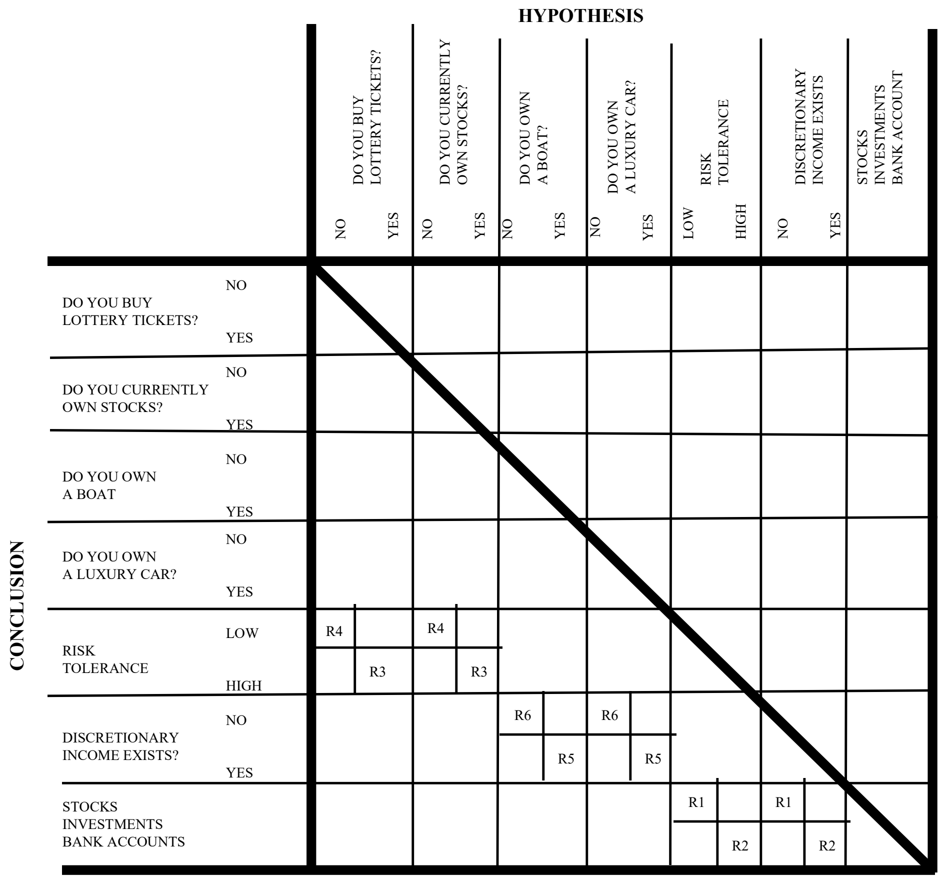
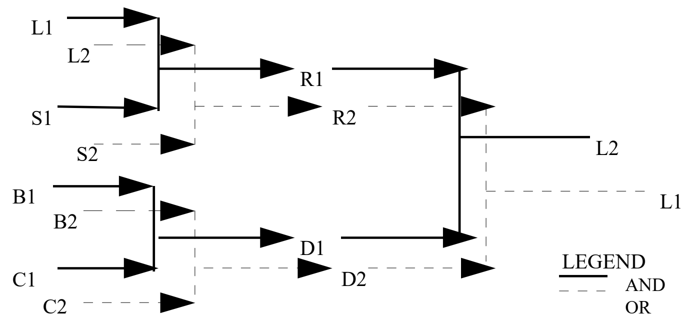


Figure 10

For knowledge bases other than binary systems and with more than two hypotheses in rules, an alternative illustration is proposed. An incidence matrix, with rule numbers as values, is developed. The rules are clustered using their commonality of hypotheses and conclusions. The clusters are then ordered so that the bandwidth of the incidence matrix is minimum. Within a cluster, the hypotheses are placed before the conclusions. Figure 10 shows the final incidence matrix for KB1. Note that the partitions are evident. There are three sub-matrices found in the lower triangle of the incidence matrix. They are the smallest matrices which include all variables of a cluster.

Figure 6-10

DO YOU BUY LOTTERY TICKETS?	NO	L1
	YES	L2
DO YOU CURRENTLY OWN STOCKS?	NO	S1
	YES	S2
DO YOU OWN A BOAT?	NO	B1
	YES	B2
DO YOU OWN A LUXURY CAR?	NO	C1
	YES	C2
RISK TOLERANCE?	LOW	R1
	HIGH	R2
DISCRETIONARY INCOME EXISTS?	NO	D1
	YES	D2
INVESTMENTS	STOCKS	I1
	BANK ACCOUNT	I2



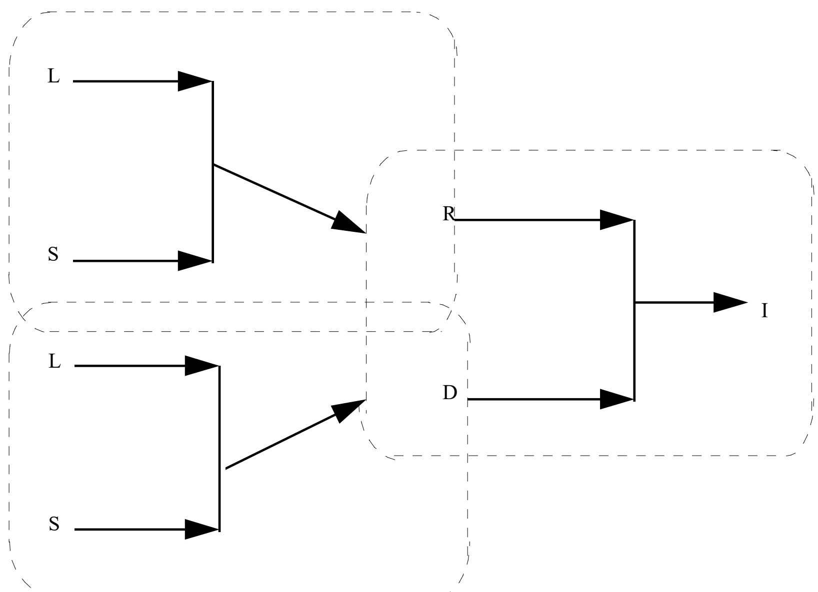


Figure 11

Another method of representing a knowledge base is the petri-net method. Each variable is given a name, and each value, a digit. For example, the variable "Do you buy lottery tickets?" is assigned the letter "L", and the values "no" and "yes", "1" and "2", respectively. For example, the hypotheses "Do you buy lottery tickets?"="no" is assigned to variable "L1". In Figure 11, a table in the upper left corner lists the correspondence between the hypotheses and the variables for the knowledge base KB1. There are also two graphical representations of KB1. The upper one relates the variables without details of the logical syntax. The lower one provides those details. The dashed line indicates that the hypotheses are subjected to logical operator "or", and a solid line, "and", as shown in the legend.

Figure 6-11

CASE STUDIES OF COMPLETENESS AND CONSISTENCY

The partition {R3,R4} of KB1 is used to illustrate the concept of completeness and consistency. In cases other than the first one, the two rules are modified by changing either the logical operator or the conclusions.

CASE 1: Complete and consistent subsystem

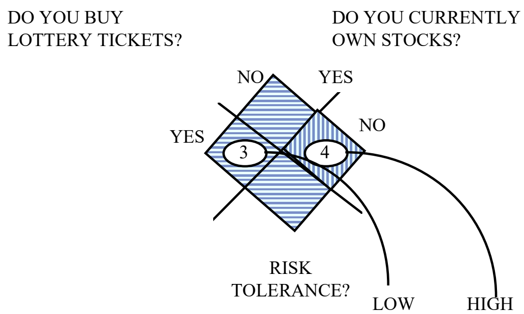


FIGURE 1

RULE 3:
DO YOU BUY LOTTERY TICKETS?=YES
OR
DO YOU CURRENTLY OWN STOCKS?=YES
THEN
RISK TOLERANCE=LOW

RULE 4:
DO YOU BUY LOTTERY TICKETS?=NO
AND
DO YOU CURRENTLY OWN STOCKS?=NO
THEN
RISK TOLERANCE=HIGH

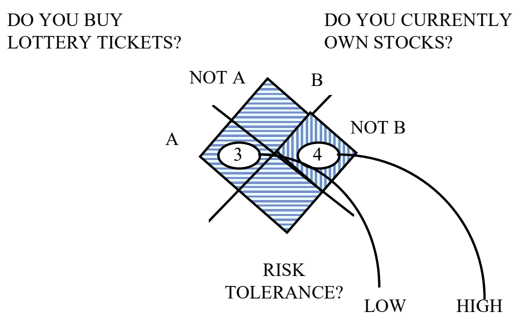
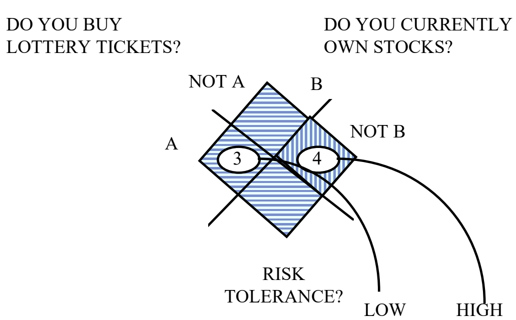


FIGURE 2

COMPLETENESS CHECK FOR SUBSYSTEM
LOW <u>OR</u> HIGH
(A OR B) OR (NOT A AND NOT B)
(A OR B OR NOT A) AND (A OR B OR NOT B)
TRUE AND TRUE
<u>TRUE</u>
<u>COMPLETE</u>



CONSISTENCY CHECK FOR SUBSYSTEM
LOW <u>AND</u> HIGH
(A OR B) AND (NOT A AND NOT B)
(A AND NOT A AND NOT B) OR (B AND NOT A AND NOT B)
FALSE OR FALSE
<u>FALSE</u>
<u>CONSISTENT</u>

FIGURE 3

Figures 6-1a, 6-2a, 6-3a

All Hoffman regions are assigned to a unique rule. The results of the formal procedure for checking completeness and consistency are shown in figures 2 and 3. In both checks, the procedures starts at the conclusions. A logical expression is built-up with all possible values of the variables in the conclusions.

CASE 2: Incomplete but consistent partition

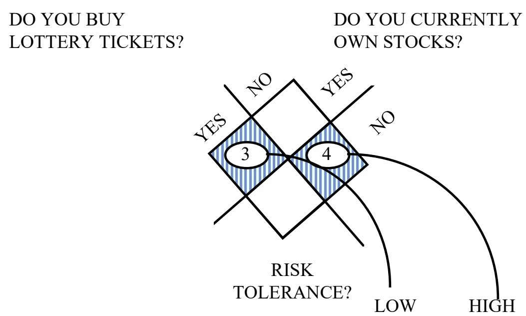


FIGURE 4

RULE 3:
DO YOU BUY LOTTERY TICKETS?=YES
AND
DO YOU CURRENTLY OWN STOCKS?=YES
THEN
RISK TOLERANCE=LOW

RULE 4:
DO YOU BUY LOTTERY TICKETS?=NO
AND
DO YOU CURRENTLY OWN STOCKS?=NO
THEN
RISK TOLERANCE=HIGH

The logical operator in rule 3 was changed from an "or" to an "and". Two Hoffman regions are without rule assignment shown by blank patterns. This partition has an incomplete set of rules.

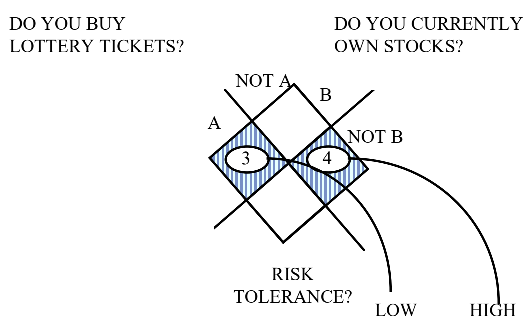


FIGURE 5

COMPLETENESS CHECK FOR SUBSYSTEM
LOW <u>OR</u> HIGH
(A AND B) OR (NOT A AND NOT B)
(A AND B OR NOT A) AND (A AND B OR NOT B)
TRUE AND TRUE
<u>TRUE</u>
<u>COMPLETE</u>

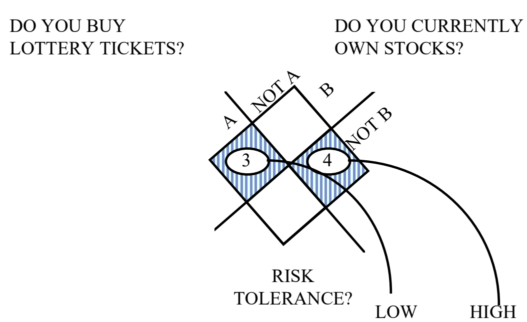
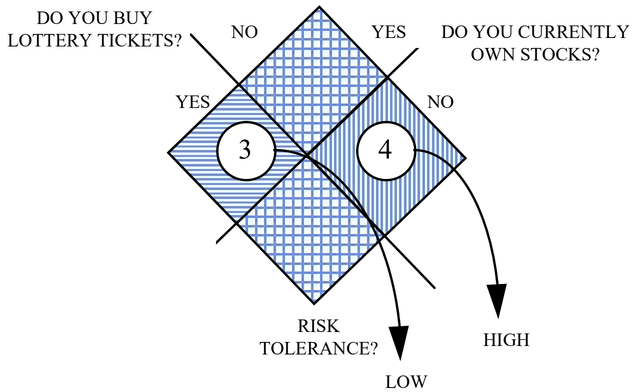


FIGURE 6

CONSISTENCY CHECK FOR SUBSYSTEM
LOW <u>AND</u> HIGH
(A AND B) AND (NOT A AND NOT B)
<u>FALSE</u>
<u>CONSISTENT</u>

Figures 6-4a, 6-5a, 6-6a

CASE 3: Complete but inconsistent partition

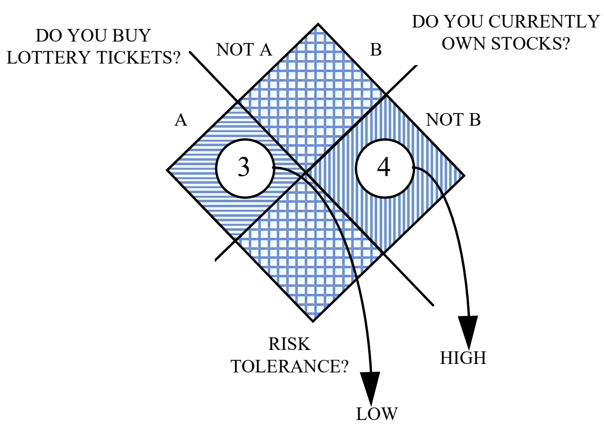


RULE 3:
DO YOU BUY LOTTERY TICKETS?=YES
OR
DO YOU CURRENTLY OWN STOCKS?=YES
THEN
RISK TOLERANCE=LOW

RULE 4:
DO YOU BUY LOTTERY
TICKETS?=NO
OR
DO YOU CURRENTLY OWN
STOCKS?=NO
THEN
RISK TOLERANCE=HIGH

Figure 7

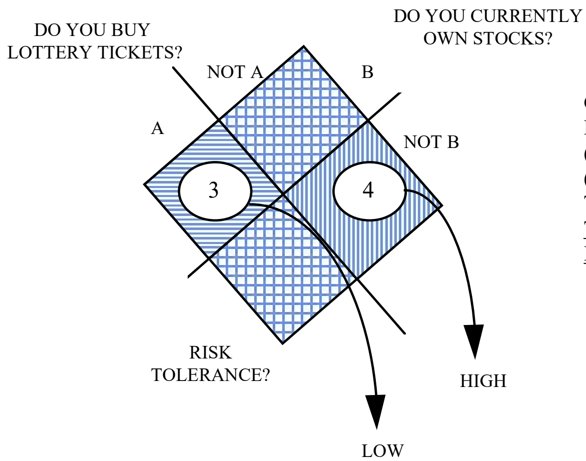
The logical operator in rule 4 was changed from an “and” to an “or”. Two Hoffman regions are assigned to two distinct rules shown here by an overlap in the hatch patterns. The partition has an inconsistent set of rules.



COMPLETENESS CHECK FOR SUBSYSTEM
LOW <u>OR</u> HIGH
(A AND B) OR (NOT A AND NOT B)
<u>TRUE</u>
<u>COMPLETE</u>

Figure 8

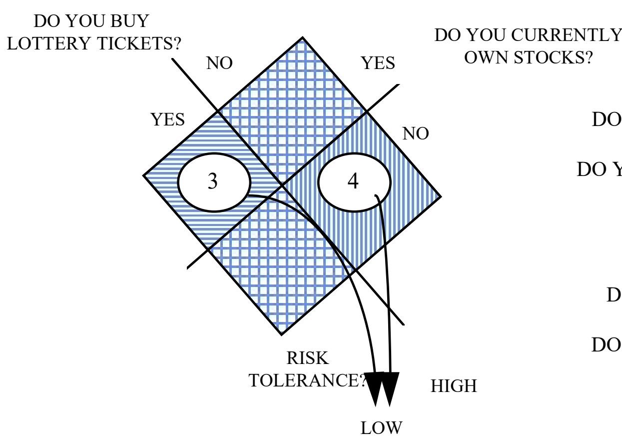
Figures 6-7a, 6-8a



CONSISTENCY CHECK FOR SUBSYSTEM
LOW <u>AND</u> HIGH
(A OR B) AND (NOT A OR NOT B)
(A OR B AND NOT A) OR (A OR B AND NOT B)
TRUE OR TRUE
<u>TRUE</u>
<u>INCONSISTENT</u>

Figure 9

CASE 4: Rules that can be lumped



RULE 3:
DO YOU BUY LOTTERY TICKETS?=YES
OR
DO YOU CURRENTLY OWN STOCKS?=YES
THEN
RISK TOLERANCE=LOW

RULE 4:
DO YOU BUY LOTTERY TICKETS?=NO
AND
DO YOU CURRENTLY OWN STOCKS?=NO
THEN
RISK TOLERANCE=LOW

Figure 10

The consequent in rule 4 was changed to be the same as the one for rule 3. The two rules are consistent, but they should be lumped into one.

Figures 6-9a, 6-10a

Knowledge Modeling

This chapter presents some *knowledge models* that can be used to partition knowledge bases using expert knowledge. The chapter includes:

- definition of knowledge models
- using knowledge models for VV&E
- using knowledge models in the expert system lifecycles
- some example knowledge models
- proof techniques for specific knowledge models
- specific knowledge models:

-- decision trees	-- ripple down rules	-- state diagrams
-- flowcharts	-- functionally modeled systems	

Appendix A presents some mathematical results used in the chapter about partitioning using the clear box methodology.

Introduction

Knowledge models are high level templates for expert knowledge. These templates express the high level structure of the expert knowledge. Examples of knowledge models are decision trees, flowcharts and state diagrams. By organizing the knowledge, a knowledge model helps with VV&E by suggesting strategies for proofs and partitions; in addition, some knowledge models have mathematical properties that help establish completeness, consistency or specification satisfaction.

More particularly:

- The knowledge model highlights the main points of a knowledge base, often obscured in the knowledge base.
- A knowledge model partitions a large KB into smaller, easier to verify, pieces.
- There are mathematical properties of the knowledge model that help establish the correctness of a knowledge base.

An Example of a Knowledge Model

PAMEX (Pavement Maintenance Expert System) is an expert system for pavement maintenance management [Aougab et. al., 1988]. A top level model of PAMEX consists of a partition of the problem space on the following three variables:

- Level of information about the pavement; the 3 values are extensive, some and little or none.
- Range of pavement serviceability index (PSI); the 3 values are above 2.8, between 2.8 and 2.0, and below 2.0.
- The level of treatment desired; the 3 values are long-range, mid-term and short-term.

For each of the twenty seven regions formed by the cartesian product of the three regions on each variable, there is a small expert system that handles problems in that region. Although all these small expert systems use the same pavement variables, i.e., PSI and other more specific pavement measurements. In this case, the model is a decision tree, discussed and illustrated in the next section.

Using Knowledge Models in VV&E

The steps in using a knowledge model in VV&E are:

- Collect the knowledge model from:
  - The domain expert(s) working on the project.
  - Standards documents in the domain.
  - Notes from knowledge acquisition at the time an existing system was built.
- Validate the knowledge; see Chapter 9 on knowledge validation for details. This step is to insure that the knowledge going into the expert system represents correct expert knowledge.
- Prove the expert system using the knowledge model is complete, consistent and satisfies its specifications; this chapter, as well as chapters on partitioning and small systems, provides information on how to develop these proofs.

Decision Trees

Introduction

A decision tree is a set of decisions that partitions the input space into a set of disjoint regions that cover the entire input space. In a decision tree system, a sequence of decisions based on user input and other data are used to classify the input problem before going on to the rest of problem solution.
The top of the decision tree corresponds to the start of the decision process. At each interior node of a decision tree, the problem is supposed to be assigned to one and only one of the subnodes. The solution of the detailed problems is often handled by specialized expert systems tailored to the specialized situations found by the decision tree.

Definition

A decision tree expert system has a structure that is described by a tree. A decision tree system has the following properties:

- Each interior node of the tree has a variable or expression assigned to it.
- Each edge to a subtree is labeled with a set of values for that variable or expression on the parent node.
- All possible values of a variable are on some edge.
- No variable value is on two different sibling edges.
- Associated with each leaf node is an output(s) or subsystem. A subsystem at a tip node N of a decision tree is called to solve the problems for which variables appearing in the tree have values associated with the path that leads to N.

Example

A decision tree for PAMEX is illustrated in figure 7.1 of the following page.

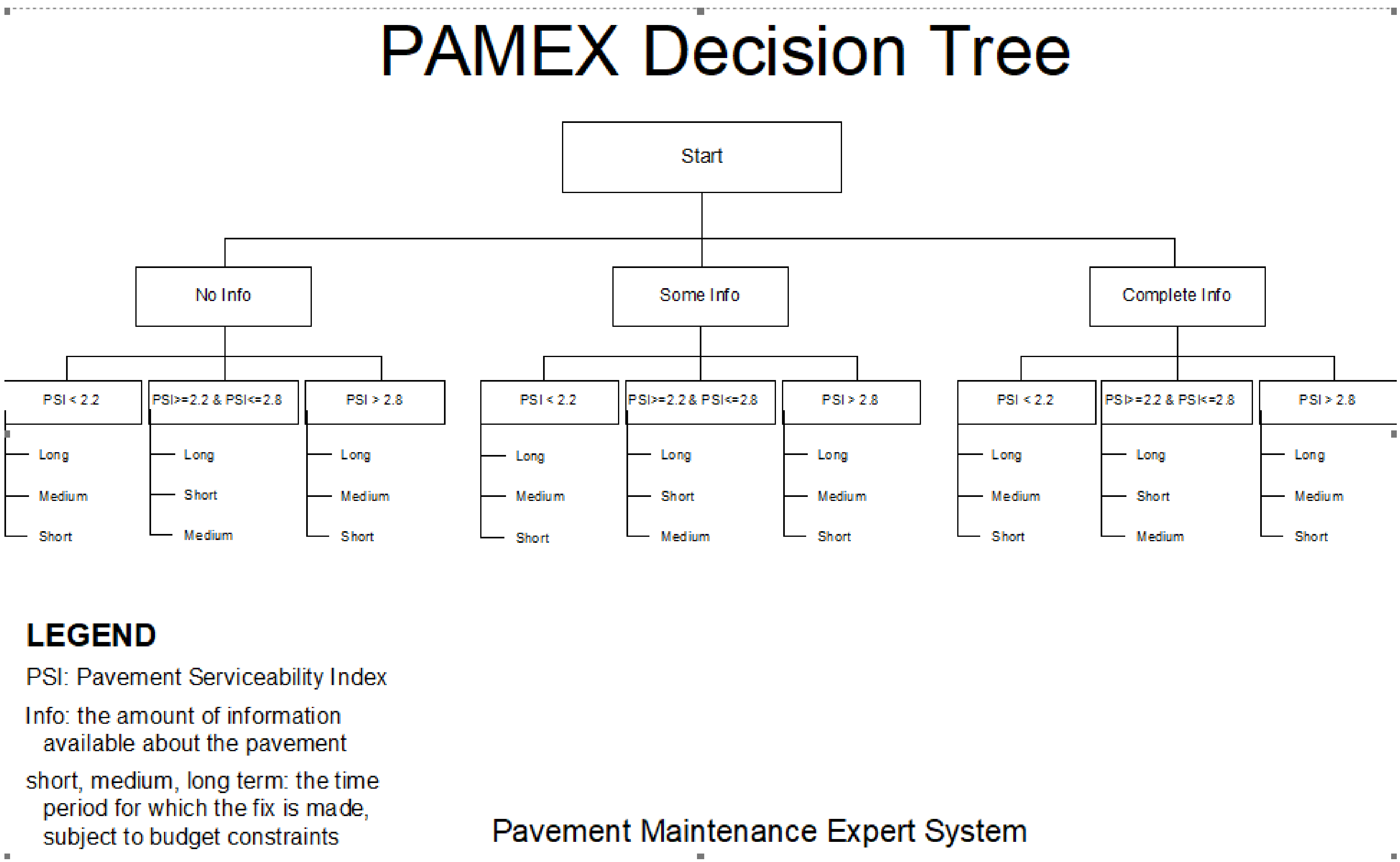


Figure 7.1: Pamex DT

Use During Development

Decision trees are a useful way to organize expert knowledge. Their use is indicated when the expert can describe in what order information is obtained and used to partially determine a solution. Drawing a decision tree from information the expert(s) have provided is a good way to present the knowledge engineer's conception of the information back to the domain expert for validation.

Use During VV&E

To model an expert system as a decision tree for the purpose of showing correctness, the following conditions should be satisfied:

- Each possible set of inputs should be in one and only one of the partitions generated by the decision tree.
- For each partition, there is an expert system (a subsystem of the entire system) that correctly solves problems in that partition.
- Experts validate the decision tree.
- The expert system assigns each input to the correct partition as the result of a finite computation.

To prove *completeness* of an expert system modeled by a decision tree, prove the following:

- Each possible problem in the input space is assigned to some partition of the decision tree.
- Each expert system assigned to one of the partitions computes a solution for each problem assigned to it.

To prove *consistency* of an expert system modeled by a decision tree, prove the following:

- Each possible problem in the input space is assigned to at most one partition of the decision tree.
- Each expert system assigned to one of the partitions computes at most one solution for each problem assigned to it.
- Each computed solution is internally consistent.

To prove *satisfaction of a requirement* of an expert system modeled by a decision tree, prove the requirement is satisfied for the expert system associated with each tip of the decision tree.

Ripple Down Rules

Introduction

Ripple down rules (RDRs) [Kang, et al, 1994.] are a special case of decision trees for reasoning with defaults. RDRs are guaranteed to be complete and consistent.

Definition

With ripple down rules, the knowledge base is organized as lists of rules. If the conditions ("if" part) of a rule are satisfied, then the expert system moves to the part of the knowledge base attached to this rule. In some cases, this is another list of rules. If so, the expert system tests the rules in the sublist. If there is no sublist of rules, or if none of the sublist rules are satisfied, then the conclusions "then part" of the rule is used. Figure 7.2 demonstrate an example of a small expert system for vehicles classification.

Example

As an example, a small expert system for vehicles classification is presented.
The main list is

L1.1: If NOA (Number -of Axles) is 2

Try List 1-1; Default = Car.

L1.2: If NOA is 3,

Try List 1-2; Default = 3 Axle-single unit Truck.

L1.3: If NOA is 4,

Try List 1-3; Default = 4 Axle-single unit Truck.

L1.4: If NOA is 5,

Try List 1-4; Default = 5 Axle-single unit Truck.

etc.

Here are the lists that fill out the next level of the knowledge base; note that this is not an exhaustive knowledge base.

L1-1.1: If S1 <= 12, it is a Car-Van-Pick up.

L1-1.2: If S1 <= 20, it is a 2 Axle-single unit Truck.

L1-1.3: If S1 > 20, it is a 2 Axle Bus.

L1-2.1: If S1 <= 12 & 8 < S2 <= 18, it is a Light Vehicle w/ Single Axle Trailor.

L1-2.2: If 7 < S1 <= 20 & S2 <= 8, it is a 3 Axle-singlr unit Truck.

L1-2.3: If S1 > 20 & S2 <= 8, it is 3 Axle Bus.

L1-2.4: If Else, it is a 2 Axle Tractor w/ Singlr Axle Trailor.

L1-3.1: If S1 > 7 & S2 + S3 <= 12, it is a 4 Axle-singlr unit Truck.

L1-3.2: If S1 > 7 & S2 <= 8 & S3 > 6, it is a 3 AxleTractor w/ Singlr Axle Trailor.

L1-3.3: If Else, it is a 2 Axle Tractor w/ Tandem Axle Trailor.

L1-4.1: If S2+S3+S4 < 16, it is a 5 Axle-singlr unit Truck.

L1-4.2: If S2 <= 8 & S4 <= 10.5, it is a 3 Axle Tractor w/ Tandem Axle Trailor.

L1-4.3: If S2 > 8 & S3 + S4 <= 12, it is a 2 Axle Tractor w/ Tridem Axle Trailor.

L1-4.4: If S2 > 8 & 12 < S3 + S4 <= 16, it is a 2 Axle Tractor w/ Tridem Axle Trailor Split.

etc.

Figure 7.2: Example ES

Similar rule lists could expand lists 1-3 and 1-4.
The expert system starts example, the system moves to list 1-2, and likewise for the other L1 rules. If none of the entry conditions to the rules in list L1 is satisfied, the default of L1, glass of orange juice, is the KB conclusion.
Under the condition that the user likes fruit, the system moves to list 1-2. If the date is not April or May, one of the fruits in L1-2 is selected. In April or May, the default of rule 1.1, banana is used as the KB conclusion.

Use During Development

Kang et. al., 1994 point out that it is possible to add correction rules to a running ripple down rules expert system. Whenever an error occurs, that error gets added to the last list of rules which the system tried before choosing an erroneous default.
Ripple down rule systems are ideally suited to problems where knowledge has the following structure:

- Early decisions made on a problem narrow the range of possible solutions, while later decisions pick particular solutions from a selected class.
- There is a default solution at each stage of the solution process.

Changing a Ripple Down Rule System

Ripple down rules are a special type of decision tree. For a knowledge base that consists of a series of more detailed decisions, but where the bases of the more detailed decisions vary for different points of the decision tree, the ripple down rules model is appropriate.
Given: an RDR, and a rule (if C then A) which the algorithm should execute, the algorithm *change* modifies the KB to make (if C then A) part of the system:
case 1: Top level list of RDR is empty.

If default(RDR) = A, do nothing,

else insert (if C then A) as a 1-element list of RDR.

case 2: The conditions on the first rule in the top level list of RDR = C.

Attach to the first rule the RDR with default = A and empty rule list.

case 3: The conditions on the first rule subsume C.

Replace the RDR attached to the first rule, denoted by R2, with *change*(R2).

case 4: C subsumes the conditions on the first rule.

Replace the first rule with (if C then A).

case 5: C and the conditions of the first rule can be simultaneously satisfied.

Insert (if C then A) before the first rule.

otherwise: Let RDR = H++T, where H is the first rule in the top level list, and T is the rest of the rules. Insert (if C then A) in T.

Use During VV&E

Completeness of a RDR system follows from the following theorem:

A Ripple-Down-Rule System is Complete.

Proof: Note that part of a RDR system attached to a top level rule is itself an RDR system. Define the level of a RDR system as follows: If the system has only 1 rule list, it is of level 1. If the system has N+1 rule lists, then it has level 1+Max(level of RDR subsystems of the top level rule list).

Let R be a RDR system of level N+1. Assume all RDR systems of level N are complete. For any input, either some top level condition is satisfied or not. In the latter case, the system concludes the default. In the former case, the system finds the conclusion computed by RDR rules from the first satisfied top level rule. If there is a rule list associated with that condition, the conclusion is from an RDR system of level at most N, and so exists. If there is no rule list, the conclusion is from the condition itself. Therefore, in all cases, an RDR system produces a conclusion.
In a similar way, it can be proven that all RDR systems are *consistent*. Consistency, however, requires an additional check: that the conclusions associated with each path through the ripple down rule tree are consistent.
Satisfaction Of Specification: To verify that an RDR satisfies a proposition P:

1. Verify or modify the default of the top level rule set.
2. Verify or modify the first rule, if any in the top level list to satisfy P.
3. Verify or modify the RDR system attached to the first rule, if any.
4. Let RDR = H++T, where H is the first rule in the top level list, and T is the rest of the rules. Verify or modify T to satisfy P.

Generalizations Of RDR: A generalization of RDR systems occurs when the conditions in RDR rules are replaced with specialized expert systems, whose purpose is to make the decision specified in the if part of the RDR rule. When, in an ordinary RDR system, an RDR rule if part is evaluated, a generalized RDR system may call an expert subsystem. This is a backward chaining process, although RDR systems are more structured than general backward chaining systems.
The same algorithms for VV&E on RDR systems also work for generalized systems, provided that the expert subsystems carry out the tests provided in the rule condition that the subsystem replaces.

State Diagrams

Introduction

A state diagram is a useful formal representation for the top level of process control expert systems.

Definition

A state diagram system is one where there is a unique *state* at every step of a solution, and at each state, there is a function that determines the next state.

Example

A state diagram can be used to model driver behavior on a road segment. A set of *states* indicates the situation and/or goal of the driver. For example, some possible states are:

- Distance ahead too small.
- Clear road ahead.

- Approaching desired exit.

A driver model based on these states is shown below. The case statement branches on the value of the variable *state*.

```
state = start_loop;
while ( state is not equal to exit )
case (state)
[
 case start_loop:
        if (distance ahead is too small)
                state = distance ahead too small;
        else (approaching desired exit)
                state = exit;
        else (clear road ahead)
                state = clear road ahead;
        else delay a small time increment;
 case clear road ahead:
        if (current speed < desired speed)
                increment speed;
        delay a small time increment;
        state = start_loop;
 case distance ahead too small:
        if (current speed < desired speed)
                { if (passing possible)
                        pass;
```

else decrease speed; }

delay a small time increment;

state = start_loop;

case exit;

return any current useful information to calling program

}

In this example, the decision to pass may be made by another expert system. In addition, fuzzy logic is often used to assign a membership grade representing how much the current situation belongs to each of the possible states. In this case, the expert system chooses a state with the highest membership grade and executes the code associated with that state.

State Diagram Systems Represented as Rules: Systems based on state diagrams may be encoded into expert system rules. The following include two of the rules that would implement the above example in rule form:

```
if state = start_loop
        and distance ahead is too small
        then state = distance ahead too small.
if state = start_loop
        and approaching desired exit
        then exit and return information to calling program
if state = start_loop
        and clear road ahead
        then state = clear road ahead
if state = start_loop
        and not ( distance ahead is too small
                        or approaching desired exit
                        or clear road ahead)
        then delay a small time increment
if state = clear road ahead
        and current speed < desired speed
        then increment speed
                and delay a small time increment
                and state = start_loop;
if state = clear road ahead
        and current speed >= desired speed
        then and state = start_loop;
```

```
if state = distance ahead too small
        and current speed < desired speed
        and passing possible
        then pass
                and delay a small time increment
                and state = start_loop
if state = distance ahead too small
        and current speed < desired speed
        and not passing possible
        then decrease speed
                and delay a small time increment
                and state = start_loop
if state = distance ahead too small
        and current speed >= desired speed
        then decrease speed
                and delay a small time increment
                and state = start_loop
```

Use During Development

State diagram models are useful during development when expert knowledge has the following characteristics:

- The problem solution consists of a series of distinct steps.
- Which step to choose is a complex, but knowledge-based decision.
- The possible paths through the steps may contain loops.

To run such a rule-based system based on state diagrams generally requires an inference engine that can do both forward and backward chaining with the same knowledge base, in a strategy called forward chaining with local backward chaining. In this strategy applied to the knowledge base, forward chaining keeps applying rules until a rule containing the command to exit the knowledge base fires. Backward chaining is used to establish the conditions within the rules, e.g., passing possible in the above example.

Use During VV&E

Completeness of a state diagram system can be established by showing that for any inputs, the system eventually reaches a *final* state in which in returns information and exits to the calling environment. In a complex system in which the predicates that control transitions between states are themselves expert systems, the proof of completeness is hierarchical:

1. Assume that the expert subsystems satisfy their specifications. Using this information, prove that the system reaches a final state.
2. Prove that the expert subsystems satisfy their specifications, and also that they terminate for any possible inputs.

Since a table of one value for each of a set of variables is consistent, state diagram systems that return a set of variable values when they reach a final state are *logically consistent*. The set of variable values may be unsatisfiable, however, given the specifications for the expert system and expert knowledge about the domain.

To show that the output of a state diagram system satisfies a specification for the expert system, show that:

- For each state, if the specs are satisfied on entering the state, they are also satisfied when leaving the state.
- The specs are satisfied at the start state. Often the specs are trivially satisfied at the start state, because the values of output variables are unknown.
- The system always reaches a final state.

Satisfaction Of Specifications: For state diagrams, to show that a specification is satisfied, show that for any input in the input set of the specification, the state diagram eventually reaches a final state in which the requirements of the specification are satisfied.

Flowcharts

Flowcharts are another method for recording expert knowledge and can serve as a model for the knowledge in an expert system.

Use During Development

Flowcharts can be implemented best using a procedural programming language, i.e., a language that permits

- Blocks, i.e., sequences of statements used as a single statement.
- Branching statements, e.g., if-then-else or switch statements.
- Loops, e.g., while, do and for loops.
- Function calls, permitting a procedure to be call other procedures or itself.

If, however, some procedural knowledge is included in a largely non-procedural knowledge base, the available implementation shell does not permit procedural programming, it may be more convenient to encode the procedural knowledge in rules.

In this case, a flowchart can be represented in rule form by associating a state with each box in the flowchart and by writing rules that describe the transitions between boxes represented by the lines in the flowchart.

Use During VV&E

Completeness, consistency, and satisfaction of specifications for flowcharts are similar to the same problems for state diagrams.
If the effect of the flowchart is to set variable values, slot values on objects, or build other data structures, the logical statements represented by these structures can usually be satisfied. The result of the flowchart is logically consistent but not necessarily consistent with the specifications for the expert system or other expert knowledge about the application domain.
Consistency: Flowcharts need not produce consistent output, even when the flowchart always reaches an exit box, or when all of the variables that are outputs of the system have a unique value.

If, however, all possible tuples of output variable values are consistent i.e., for any assignment of values to output variables, it is logically possible, and consistent with domain expertise, for the variables to have those variables simultaneously. Then, if for all inputs, the flowchart defines unique values for all the output variables, the flowchart is consistent.
Completeness: A flowchart is logically complete, if no matter what the inputs, the flowchart always reaches an exit box. For this to be true, the user must prove that the computation eventually exits from any loop entered within the flowchart; and that all functions called within the flowchart satisfy their specifications for all inputs, and perform their computation in a finite time.

Satisfaction Of Specifications: To show that a flowchart system satisfies its specifications, the basic strategy is to show that if the specifications are satisfied on entry to each box in the flowchart, they are satisfied on exit from the box. Specifications are generally satisfied before the initial box, because variables are not yet set to values, but showing somehow that specifications are satisfied at the start is a necessary part of the proof of specifications.
If box A of the flowchart has just one exit line L going to box B, then A, B, and L represent a sequence of separate computations. To show that this part of the flowchart satisfies the specifications, the system will demonstrate that the computations in A and B can always be carried out in only finite time; and that If the specifications are satisfied on entry to A and B, they are satisfied on exit. In proving the specificationss for B, the user can assume the results of the computations in A, in addition to the specifications that were assumed on entry to A.

If box A of the flowchart performs a test to decide a proposition P, and if A has exits to box B if P is true and C if P is false, then the user must demonstrate that:

1. The specifications are true at the exit box(es) when starting at B with the assumption of the specifications plus P, and that the computation always reaches an exit box in a finite computation.
2. The specifications are true at the exit box(es) when starting at C with the assumption of the specifications plus not P, and that the computation always reaches an exit box in a finite computation.

If a flowchart contains a loop, then the user must show that for all inputs satisfying the specifications that the following criterias are met:

- The specs are true on exit from the loop.
- Given the following assumptions at the loop exit:
    - The specifications
    - The results of computations in the loop.
    - The conditions for exit from the loop.

Upon the satisfaction of all input, the flowchart computation reaches an exit box in finite time and the specifications are true when reaching the exit box.

Functionally Modeled Expert Systems

Introduction

As discussed in the chapter on partitioning without expert knowledge (see Chapter 6), an expert system can be thought of as a *function*. A function maps sets of inputs (information the expert system receives from the user or other external sources) into a set of outputs reflecting actions taken and conclusions inferred by the expert system. Ideally, the function that an expert system represents is that which maps each set of problem inputs into the set of actions and inferences that an expert would make given those inputs. The expert system will be said to *implement* this function, and the function will be said to *model* the expert system, with the understanding that an expert system only approximates the behavior of an expert.

Some functions are built from simpler functions with operations such as (function) composition or cartesian product (operations discussed in more detail below). Sometimes, because of domain knowledge, the expert system should represent a function that is constructed from simpler functions. If that is the case, the structure of the function provides the knowledge engineer with tools for structuring and partitioning an expert system.

More particularly, the operations of cartesian product and function composition in the category of functions are of particular importance in modeling expert systems. Let E be an expert system such that the output of E involves setting variables O1,...,On such that the values of the Os are independent of each other. Then E implements the cartesian product of functions fi such that Oi = fi(Ii), where Ii is a subset of the inputs of the entire expert system found by computing the dependency relation (see Chapter 6 on partitioning without expert knowledge) starting with Oi.

If one of the fi is a composition of functions, e.g.

fi = h(g1(Ii), ..., gm(Ii))

then using the same techniques of Chapter 6, one can find subsystems of the original expert system that implement the gs and h can be found.

As discussed in more detail below, if the expert subsystems are complete, consistent and satisfy specifications, *and* if there is consistency and specification satisfaction among independently chosen possible values of cartesian component subsystems, the entire expert system is complete, consistent and satisfies specifications.

Note that this does *not* mean that completeness, consistency and specifications satisfaction of arbitrary subsets of an expert system imply corresponding results about systems as a whole. The

subsets *must* be those that implement functions used to construct the function that models the expert system, *and* certain additional requirements among the outputs of component systems must be met.
Expert knowledge is generally of great benefit in identifying both, independent outputs that can be used to decompose an expert system into a product of expert systems, and intermediate hypotheses that are functions of the problem inputs but are themselves inputs to a later function that produces some or all of the outputs of the system as a whole.

Following are some examples of composite functions which provide opportunities for structuring and partitioning expert systems.

Use During Development

These strategies often simplify development by replacing a single development task with two or more development tasks, which is less than the original task. ***During VV&E***, these strategies likewise replace a single VV&E task with two or more development tasks the total size of which is less than the original task.

In each of these cases, the key to whether the partitioning makes these problems smaller is found by counting Hoffman regions. If E is partitioned into E1,...,En, then if

(H(E1)+...+H(En)) / H(E)

is significantly less than 1, partitioning E into the Ei decreases the size of the development or VV&E problem. Note that usually, *some* rules and variables may be contained in more than one of the Ei.

Cartesian Product Systems: Sometimes an expert system E is required to make more than one decision, e.g., to find values for two different (sets of) variables. In this case, the user can represent the expert system function e of input I as

e(I) = (e1(I), ..., en(I)).

Using the techniques of Chapter 7, the user can find subsystems Ei which implement ei respectively. If H(X) is the number of Hoffman regions in expert system X, then if

(H(E1)+...+H(En)) / H(E)

is significantly less than 1, partitioning E into the Ei decreases the size of the VV&E problem. [Note that *some* rules and variables generally appear in more than one Ei.]

consistency: If each of the Ei is consistent, *and* if the union of consistent sets of output from each of the Ei is consistent, the entire expert system is consistent.

completeness: If each of the Ei is complete, the entire expert system is complete.

specification satisfaction: Generally, proving that specifications are satisfied will involve consideration of the interaction of the outputs of the Ei. However, if a specification is of the form

If C1 and C2 ... and Cn then S (1)

then (1) is equivalent to the set of specifications

If (AND Ei satisfies Ci) then S.

Final Layer Partitioning: In final layer partitioning, the expert system is partitioned into

- -- the *final layer expert system* that consists of all rules and functions that have as their direct outputs conclusions of the knowledge base.
- -- *information gathering expert subsystems* that conclude the inputs to the final layer system.

The final layer system contains all rules and functions that produce one or more of the conclusions of the entire expert system. The inputs of the final layer expert system are the inputs to these rules and functions. In KB1, the investment subsystem is the final layer expert system. For each of the input variables to the final layer expert system, there is an expert system that determines that input to the final level; that expert system can be found using the methods in the chapter on partitioning without expert knowledge. In particular, if the final level input variables are v1,...,vn, let E1,...,En be the expert systems that set these variables.

Those Ei and Ej which overlap greatly, so that

(H(Ei) + H(Ej)) / H(Ei union Ej) >= 1

should be combined into a single expert system that produces both vi and vj. If, on the other hand,

(H(Ei) + H(Ej)) / H(Ei union Ej)

is significantly less than 1, Ei and Ej should be kept separate. Note that as described in the chapter on partitioning without expert knowledge, clustering of vectors from incidence matrices can be used to determine which of the information gathering subsystems to combine.
Partitioning into a final layer subsystem and information gathering subsystems is particularly useful when there are many rules which compute outputs from the information gathered from the subsystems. PAMEX is an example of such an expert system. In this case, incompleteness or inconsistency in the final layer expert system causes the same error in the entire expert system; furthermore, if there are many rules in the final layer subsystem, such errors are easy to make.
Consistency: Consistency for the entire expert system occurs once the following criterias are met:

- The *final layer* expert system is consistent whenever it gets consistent inputs.
- Each of the information gathering subsystems is consistent.
- All unions of consistent output from each of the information gathering subsystems are consistent.

Completeness: If each of the information gathering subsystems is complete and the final layer expert system is complete, then the entire expert system is complete.
Satisfaction Of Specifications: Generally, proving that specifications are satisfied will involve consideration of the interaction of the outputs of the information gathering subsystems. However, if a specification is of the form

If C1 and C2 ... and Cn and Cf then S (1)

where Ci is a condition on subsystem Ei and Cf is a condition on the final layer,

then (1) is equivalent to the set of specifications

If (AND Ei satisfies Ci)

and the final layer satisfies Cf,

then S.

If the final layer satisfies

<u>Intermediate Variables</u>: Intermediate variables are variables that are computed or inferred from input variables, and are used to infer or compute conclusions.

Many expert systems can be decomposed into two sequential steps (an expert can often tell the user about such a decomposition):

1. Determine the value of some intermediate variables.
2. Draw conclusions from these intermediate variables.

In addition, an intermediate variable is useful for partitioning only if some of the input variables of the system as a whole are used for computing the intermediate variable.
In function notation, an expert system with an intermediate variable is of the form
e(x1,...,xn) = e(x1,...,xk, y), where y = g(xk+1,...,,Xn).

Results about completeness, consistency, and specification satisfaction are entirely analogous to those for final level partitioning. However, the role of the final level expert system is that expert system which implements the function

e(x1,...,xk, y)

with inputs x1,...,xk and y. This expert system can be found by the method in Chapter. 6. The single information gathering subsystem is:

g(xk+1,...,Xn).

Partitioning Of The Function Domain: Let E be an expert system which implements the function e(I), where I is a vector of inputs. Let the domain of I be some domain D, such that D is partitioned into mutually exclusive subsets {Di}, i.e.,

Union{Di} = D

Di intersection Dj = NULL for i != j

Let Ei be the expert system that implements the function

e restricted to Di

Then the following results relate correctness of E to the correctness of the Ei.

Consistency: If each of the Ei is consistent, so is E.

Completeness: If each of the Ei is complete, so is E.

Satisfaction Of Specification: If a specification is satisfied by each Ei, it is satisfied by E.

Examples of domain partitioning occur in decision tree systems. The effect of the decision tree is to partition the entire domain of the expert system into subsets; each of which satisfies the conditions along the path from some leaf node of the decision tree to the root of that tree.

ILLUSTRATIONS FOR PARTITIONING OF KNOWLEDGE BASE 1

The concept of relations was introduced in chapter 7. It is applied to KB1 to determine its subsystems. Each subsystem can be represented by a function which has a domain and a range shown in figure 1 for the subsystem "Risk Tolerance".

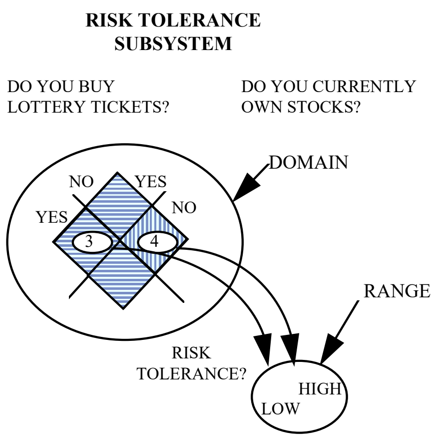


Figure 1

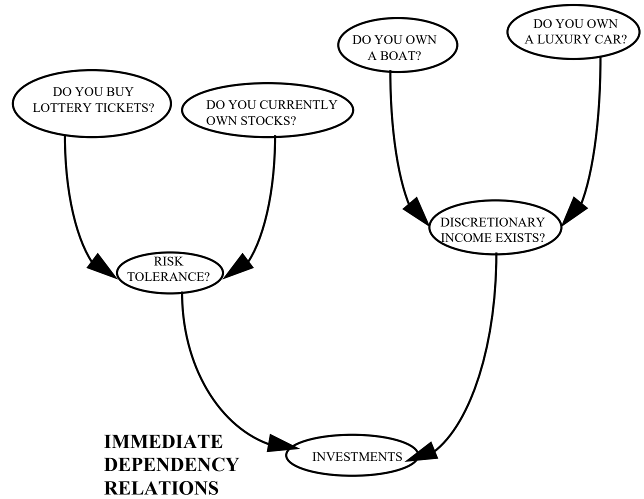


Figure 2

In figure 2, the immediate dependency relations of variables on variables are shown by connections. In order to identify the clusters of variables of each subsystem, an algebric procedure was defined. First, two relations are input: 1) the immediate dependency of rules on variables (shown in figure 3), and 2) the immediate dependency of variables on rules shown in figure 4).

DEPENDENCY RELATIONS OF RULES ON VARIABLES	RULE 1	RULE 2	RULE 3	RULE 4	RULE 5	RULE 6
VARIABLES IN HYPOTHESES						
DO YOU BUY LOTTERY TICKETS?					1	1
DO YOU CURRENTLY OWN STOCKS?					1	1
DO YOU OWN A BOAT			1	1		
DO YOU OWN A LUXURY CAR?			1	1		
RISK TOLERANCE	1	1				
DISCRETIONARY INCOME EXISTS?	1	1				
INVESTMENTS						

Figure 3

DEPENDENCY RELATIONS OF RULES ON VARIABLES	VARIABLES IN CONCLUSIONS: DO YOU BUY LOTTERY TICKETS?	DO YOU CURRENTLY OWN STOCKS?	DO YOU OWN A BOAT?	DO YOU OWN A LUXURY CAR?	RISK TOLERANCE?	DISCRETIONARY INCOME EXISTS?	INVESTMENTS
RULE 1							1
RULE 2							1
RULE 3						1	
RULE 4						1	
RULE 5					1		
RULE 6					1		

Figure 4

Figures 7-1, 7-2, 7-3, 7-4

VARIABLES IN CONCLUSIONS

IMMEDIATE DEPENDENCY RELATIONS OF VARIABLES ON VARIABLES (VARIABLES IN HYPOTHESES)	DO YOU BUY LOTTERY TICKETS?	DO YOU CURRENTLY OWN STOCKS?	DO YOU OWN A BOAT	DO YOU OWN A LUXURY CAR?	RISK TOLERANCE	DISCRETIONARY INCOME EXISTS?	INVESTMENTS
DO YOU BUY LOTTERY TICKETS?					2		
DO YOU CURRENTLY OWN STOCKS?					2		
DO YOU OWN A BOAT						2	
DO YOU OWN A LUXURY CAR?						2	
RISK TOLERANCE							2
DISCRETIONARY INCOME EXISTS?							2
INVESTMENTS							

Figure 5

The terms left blank in the matrix are zero. The product of matrix A from figure 3 and matrix B from figure 4 is called matrix C, shown in figure 5. If subjected to a boolean operation, its non-zero terms become unity. It corresponds to all connections in figure 2.

In figure 5, the immediate dependency relation matrix is shown prior to the boolean operation. The composition of this relation may be btained by multiplying matrix C by itself. Figure 6 shows the result as matrix D. It corresponds to the connection shown in solid line in figure 7, remembering that the composition operation provides all possible paths to an output from the inputs.

The dependency relation is the union of the immediate dependency relation and the composition operation. It is shown in figure 8 and 9.

VARIABLES IN CONCLUSIONS

COMPLETION (VARIABLES IN HYPOTHESES)	DO YOU BUY LOTTERY TICKETS?	DO YOU CURRENTLY OWN STOCKS?	DO YOU OWN A BOAT	DO YOU OWN A LUXURY CAR?	RISK TOLERANCE	DISCRETIONARY INCOME EXISTS?	INVESTMENTS
DO YOU BUY LOTTERY TICKETS?							1
DO YOU CURRENTLY OWN STOCKS?							1
DO YOU OWN A BOAT							1
DO YOU OWN A LUXURY CAR?							1
RISK TOLERANCE							
DISCRETIONARY INCOME EXISTS?							
INVESTMENTS							

Figure 6

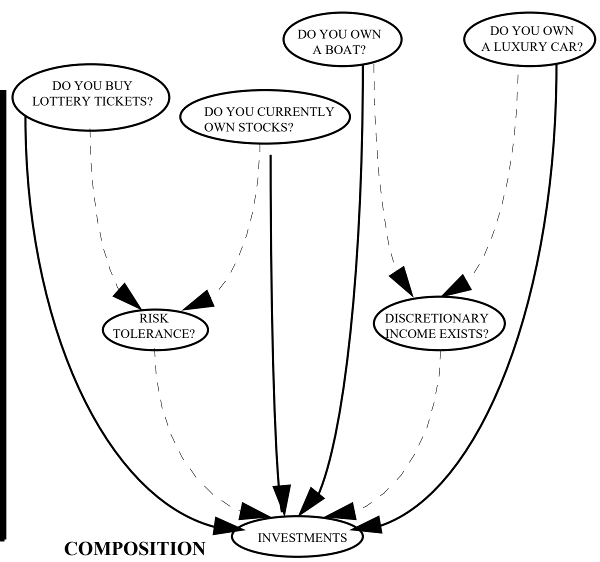


Figure 2

Figures 7-5, 7-6

VARIABLES IN CONCLUSIONS

DEPENDENCY RELATIONS OF VARIABLES ON VARIABLES (VARIABLES IN HYPOTHESES)	DO YOU BUY LOTTERY TICKETS?	DO YOU CURRENTLY OWN STOCKS?	DO YOU OWN A BOAT	DO YOU OWN A LUXURY CAR?	RISK TOLERANCE	DISCRETIONARY INCOME EXISTS?	INVESTMENTS
DO YOU BUY LOTTERY TICKETS?					1		1
DO YOU CURRENTLY OWN STOCKS?					1		1
DO YOU OWN A BOAT						1	1
DO YOU OWN A LUXURY CAR?						1	1
RISK TOLERANCE							1
DISCRETIONARY INCOME EXISTS?							1
INVESTMENTS							

Figure 8

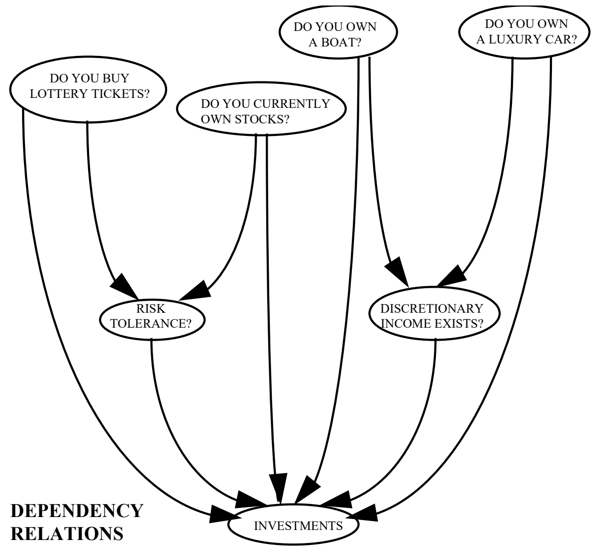


Figure 9

Figures 7-8, 7-9

VV&E for Small Expert Systems

Small expert systems are those for which direct proof of completeness, consistency, and specification satisfactionare practical, without partitioning the knowledge base. This chapter discusses techniques for these proofs.

The basic method for verifying properties of small systems is:

1. Represent the property to be verified as a logical formula.
2. Verify the logical formula using one of the following techniques:
   - Verify the formula on a case-by-case basis, e.g., by checking each Hoffman region.
   - Apply Boolean algebra simplifications to verify the formula.

Completeness

To verify completeness, the user must demonstrate that for all inputs, the expert system produces *some* conclusion. This is done by:

1. Constructing a logical formula that represents conditions under which the system is complete; this logical formula will be called the *completeness formula*.
2. Showing that the truth value of that formula is TRUE.

If the expert system E is *cartesian*, i.e,. it is required to produce values of more than one variable, then the completeness formula for E is the AND of the completeness formulas for systems which set *each* of the required output variables for E.

For a system E that is required to take one of some set of actions, a pseudo-variable is created of which values are the enumerated set of acceptable actions. Then, the completeness formula for E is the completeness formula for a system that outputs the value of this variable.

The completeness formula for an expert system in E which sets a single variable v is constructed by an iterative substitution process from an initial formula. That initial formula is:

(v = e1) OR (v = e2) OR ... OR (v = en) (8.1.1)

where v = ei is an expression from a rule conclusion that sets v.

It is generally not possible to establish the truth of (8.1.1) directly. However, the user can build a formula that expresses the truth of (8.1.1) in terms of the input variables of the system. To build this formula, the user needs the following *hypothesis function* on atomic logical formulas in E:

Let X be a formula of the form, (8.1.2)

VARIABLE = VALUE

If there is a rule in E containing X in its conclusion,

H(X) = OR(Hi(X))

where Hi is the hypothesis (if part) of a rule in E which contains X in its then part.

Otherwise H(X) = X.

Using the function H, one can define a logical formula that expresses (8.1.1) in terms of input variables:

Let F0 be a variable over logical formulas. (8.1.3)

F0 = (8.1.1);

while

(F0 contains an atomic formula for which H(X) != X)

F = the result of substituting H(X) for X in f;

return F0;

The resulting logical formula, which will be called COMPLETENESS, expresses completeness in terms of input variables to the expert system E. E is complete if the truth value of this formula is TRUE. To prove that COMPLETENESS is TRUE:

1. Write COMPLETENESS in conjunctive normal form.
2. Eliminate ORs containing logical opposites or all possible values of a variable.

If the resulting logical expression is TRUE, the system is complete. If the resulting logical expression is something else (call it COMPLETE0 for discussion purposes), then COMPLETE0 expresses the conditions under which the system produces a conclusion. Although not logically true, COMPLETE0 may be true because of mathematical theorems or domain knowledge. Alternatively, NOT COMPLETE0 may be satisfiable. In this case, the expert system E is not complete.

Figure 8.1 below illustrates the above explanation of completeness.

Completeness of Investment Subsystem

To show the completeness of the investment subsystem (call it INV) of KB1, the first step is to construct the formula (8.1.1) for INV:

investment = stocks OR investment = "bank account" (8.1.a)

Expressing this in terms of input conditions gives

("Risk tolerance" = high (8.1.b)

AND "Discretionary income exists" = yes)

OR

("Risk tolerance" = low

OR "Discretionary income exists" = no)

Writing this in conjunctive normal form gives

("Risk tolerance" = high (8.1.c)

OR "Risk tolerance" = low

OR "Discretionary income exists" = no)

AND

("Discretionary income exists" = yes

OR "Risk tolerance" = low

OR "Discretionary income exists" = no)

The first term is TRUE because high and low are the *only* possible values for risk tolerance. Likewise the second term is TRUE because yes and no are the only possible values for discretionary income exists. Therefore, the formula expressing completeness of INV is TRUE, and INV is complete.

Figure 8.1: Completeness of Investment Subsystem

Consistency

To verify consistency, the user must demonstrate that for all inputs, the expert system produces *a consistent set* of conclusions, i.e., that for each set of possible inputs, all the conclusions of the expert system can be true at the same time. (As noted in an earlier chapter, determining which sets of possible conclusions are consistent generally requires expert knowledge.)
To establish consistency, the user must do the following:

1. Construct a logical formula that represents conditions under which consistency fails; this logical formula will be called the *consistency formula*.
2. Show that the truth value of that formula is FALSE.

For a system E that is required to take one of some set of actions, a pseudo-variable is created whose values are the enumerated set of acceptable actions. Then the consistency formula for E is the consistency formula for a system that, perhaps among other things, outputs the value of this variable.
If there are no sets of inconsistent possible outputs, the system is consistent. Some expert systems are designed to recommend a set of components of a solution, and no one component contradicts any other. An investment advisor who recommended to each investor a set of desirable investments would be an example of this. For such systems, consistency is not an issue.
Let I1,...,In be the sets of mutually inconsistent possible conclusions of E. Each I consists of some set of conclusions, e.g.,

Ii = {Ci1, ... , Ci(mi)} (8.2.1)

where the Cs are possible conclusions of E.

The consistency formula for E is:

F(I1) or F(I2) ... or F(In) = FALSE (8.2.2)

where

F(Ii) = Ci1 and ... and Ci(mi) (8.2.3)

It is generally not possible to establish the truth of (8.2.3) directly. A formula can be built, however, that expresses the truth of (8.2.3) in terms of the input variables of the system. Just as for completeness, this formula is constructed by substituting the OR of rule hypotheses that infer a conclusion for that conclusion. Substituting the hypothesis function (8.1.2) into (8.2.3) using the iterative algorithm (8.1.3) constructs the consistency formula.

The resulting logical formula, which will be called CONSISTENCY, expresses consistency in terms of input variables to the expert system E. E is consistent if the truth value of this formula is TRUE. To prove that CONSISTENCY is TRUE:

1. Write CONSISTENCY in disjunctive normal form.
2. Eliminate ANDs containing logical opposites or other contradictory sets of conjuncts.

If the left hand side of the resulting logical expression is FALSE, the system is consistent. If the resulting logical expression is something else (call it CONSISTENT0 for discussion purposes); then CONSISTENT0 expresses the conditions under which the system produces possibly contradictory conclusions. Although not logically false, CONSISTENT0 may be false because of mathematical theorems or domain knowledge.

Alternatively, CONSISTENT0 may be satisfiable. In this case, the expert system E is not consistent, and is inconsistent for the inputs which satisfy CONSISTENT().

Consistency of Investment Subsystem

To show the consistency of the investment subsystem (call it INV) of KB1, the first step is to construct the formula (8.2.3) for INV. The only set of inconsistent conclusions is

{investment = stocks, investment = "bank account"} (8.2.a)

Therefore, (8.2.3) for INV is

investment = stocks AND investment = "bank account" (8.2.b)

To show INV is consistent, one must show that (8.2.b) is FALSE.

Expressing this in terms of input conditions gives

("Risk tolerance" = high (8.2.c)
AND "Discretionary income exists" = yes)
AND
("Risk tolerance" = low
OR "Discretionary income exists" = no)
= FALSE

Writing this in disjunctive normal form gives

("Risk tolerance" = high (8.1.d)
AND "Discretionary income exists" = yes)
AND "Risk tolerance" = low)
OR
("Risk tolerance" = high
AND "Discretionary income exists" = yes)
AND "Discretionary income exists" = no)
= FALSE

The first term is FALSE because high and low are contradictory values for risk tolerance. Likewise the second term is FALSE because yes and no are contradictory values for discretionary income exists. Therefore, the left hand side of (8.1.e) is an OR of FALSEs, and is FALSE. This establishes the truth of the consistency formula for INV, and therefore INV is consistent.

Figure 8.2: Consistency of I Subsystem

Specification Satisfaction

While the vast range of possible specifications (as well as the Goedel Incompleteness Theorem) makes it impossible to give a general method for proving specifications, there are some particular kinds of specifications where certain methods are useful.

Many valid specifications are not directly provable, because they are not expressed in the variables and propositions used for the knowledge base. Before a specification can be proved, it must be translated into the variables and relations used in the knowledge base. Translating specifications into the language of a knowledge base requires expert knowledge. Furthermore, this translation process may expose conditions under which the specifications are violated.

Step 1: Find all the possible conclusions that are constrained by the specification.

Step 2: Show that each of these conclusions are only made when permitted by the specification; i.e. for the specification S, and each conclusion C identified in Step 1,

If C then S(C) (8.3.1)

where S(C) is the result of substituting the variable values contained in C into S.

Let EC be the conditions under which the expert system E concludes C. EC is computed by procedure (8.1.3) above. Suppose:

EC implies S(C) (8.3.2)

Then whenever C occurs, i.e., when EC is true, S(C) is also true. On the other hand, if expert knowledge causes one to question (8.3.2), there is reason to think that the expert system can conclude C when S(C) is false.

Figure 8.3 shows a reasonable specification for Knowledge Base1.

A reasonable specification for KB1 is that it never recommend an unaffordable investment.

Step 1: The conclusion, investment = stock, is an investment that might not be affordable.

Step 2: Formulate how each conclusion is affected by the constraint, e.g.,

Expert system concludes "investment = stock" implies stock is affordable.

The successive substitutions of H(X) for X in this statement, using algorithm (8.1.3), produce a succession of ever more detailed statements about when the specification is true. For INV, these statements are:

If "Risk tolerance" = high

AND "Discretionary income exists" = yes

the stocks are affordable.

If ("Do you buy lottery tickets" = yes (8.3.a)

OR "Do you currently own stocks" = yes)

AND

("Do you own a boat" = yes (8.3.b)

OR "Do you own a luxury car" = yes)

THEN the stocks are affordable.

The truth of these statements depends on expert knowledge. If experts doubt *any* of them, it is probably because the conditions found in KB1 under which it concludes investment = stocks, are not sufficient to guarantee the specifications. In fact, (8.3.a) seems plausible while (8.3.b) seems weak. This indicates that the conditions for concluding the intermediate hypotheses,

If "Risk tolerance" = high

AND "Discretionary income exists" = yes

are not completely expressed in KB1.

Figure 8.3: Example Specification for KB1

Specification Based on Domain Subsets

Many specifications are of the form:

If the input is in some set S, (8.4.1)

then the output satisfies a logical formula P.

where S is defined by a logical formula C(I) over the input variables, and B(C1,...,Cn)) is a formula built over the conclusions, Ci, of the expert system; i.e. (8.4.1) becomes

If C(I) then B(C1,...,Cn) (8.4.2)

To prove specifications like (8.4.2), *symbolic evaluation* of the knowledge base is a useful technique; the user can try to prove (8.4.2) by symbolic evaluation using either forward or backward chaining. With these proof methods, the user simulates the inference engine operating on inputs using the knowledge base. However, instead of actual input values, the only thing known about the inputs is that they satisfy C(I). This may be enough, however, to establish that the if a position of some of the rules are satisfied, then the conclusions will be derived within the said part of the rule. If so, these conclusions have been proved true on the basis of the assumptions, C(I). Further reasoning may lead eventually to showing that B is true.

Here is a forward chaining algorithm to prove B given C(I):

1. Assume C(I) = TRUE. : (8.4.3)
2. If the truth value of B can be established using known information,

   do so and goto X.
3. If the if part of a previously unsatisfied rule can be satisfied,

   then set the then part of the rule to TRUE and goto 2

   else quit, failing in the attempt to prove B.
4. If B is true,

   then the specification has been proved

   else if B is false

   then the specification is not satisfied.

Here is the backward chaining algorithm:

1. CURRENT = if C(I) then B(C1,...,Cn). (8.4.4)
2. If the truth value of CURRENT can be established, do so and quit with the following result:

   If CURRENT is true the specification has been proved, but if CURRENT is false, the specification is not true.

3. If there is an atomic formula A for which H(A) != A (see 8.1.2)

    substitute H(A) for A in CURRENT;

    Goto 2.

4. Quit with failure to establish the specification.

Figure 8.4 shows the symbolic evaluation of the KB1 example from Chapter 5.

To illustrate symbolic evaluation, the following specification will be verified on the original KB1 of Ch. 5:

If current savings < $3000, recommend that the investment is savings account. (8.4.a)

To prove the requirement, one assumes the condition

"Savings balance" < $3000 (8.4.b)

and tries to prove that the expert system concludes that

investment = "bank account".(8.4.c)

The strategy for carrying out this proof is to use a modification of the expert system's inference engine. It must be assumed that the inference engine makes inferences according to the rules of propositional logic, but which among all the possible inferences that could be made are actually made is left to the scheduling strategy programmed into the inference engine. For illustrative purposes, a backward chaining strategy is assumed.

Using a backward chaining strategy to prove that the expert system concludes 8.4.c, the user starts with that conclusion and shows that it is satisfied. The only way to do this in Knowledge Base 1B is to satisfy the "if" part of Rule 2. These conditions are true whenever

"Discretionary income exists" = no. (8.4.d)

Rule 6 makes this conclusion whenever

"Savings balance" <= $3000. (8.4.e)

so (8.4.a) follows.

Notice that this proof mimics the inference engine of the expert system. In fact, every step of the proof could be carried out by the inference engine except for the last step of concluding 8.4.a. However, a modified inference engine could carry out that step if, whenever a truth value for an inequality was needed, a knowledge base about inequalities was consulted. In fact, such a knowledge base appears in the appendix of this chapter, and contains a rule that says

If X<=C then X<C.

Using this rule, a modified inference engine that consults an knowledge base about when atomic formulas are true is able to *automatically prove the desired condition* (8.4.a) *about the knowledge base*.

Figure 8.4: Symbolic Evaluation

The main differences between actual and symbolic inference engines are:

- Actual inference engines collect actual values for variables and use them in evaluating the conditions of rules to see if those rules fire.
- Symbolic inference engines have available logical conditions about the value of variables, e.g., the hypotheses C(I). Symbolic inference engines use this known information to infer whether atomic formulas in rule hypotheses are true or false. In an appendix to this chapter appear rules for symbolically determining the value of some arithmetic atomic formulas.

To construct a symbolic inference engine from an actual inference engine, the function that determines the truth of atomic formulas is replaced, but leaves the rest of the inference engine code intact. The *actual inference engine* determines the truth of atomic formulas by:

- looking up the actual values of variables
- substituting them into the atomic formula
- determining the truth value of the result.

In a symbolic inference engine, the user can also evaluate atomic formulas when only *conditions about* variable values are known, but the actual values are unknown. The symbolic inference engine evaluates atomic formulas by:

- Assuming the known conditions about the variables in the knowledge base.
- Using this information to establish the truth of the atomic formula.

To build a symbolic inference engine requires the user to replace the function for the actual evaluation of atomic formulas with a function for symbolic evaluation, and to leave the rest of the inference alone.

Figure 8.5 list the steps involved for the actual inference engine.

Actual inference engine: For KB1, suppose the user said that his or her savings balance was $2000.
Then the truth value of the atomic formula

savings balance < $3000 (8.4.a)

can be determined by substituting in $2000 for "savings balance" to produce

$2000 < $3000 (8.4.b)

This inequality is seen to be TRUE.
Symbolic inference engine: Suppose that

"savings balance" <= $2000 (8.4.c)

is known to a symbolic inference engine. The symbolic inference engine tries to prove

"savings balance" <= $2000 (8.4.d)

IMPLIES "savings balance" <= $3000

This formula is seen to be true. In fact, using the following row of the table in the appendix for evaluating atomic formulas,

ATOMIC FORMULA TRUTH CONDITION FALSE CONDITION

[a,b]=<c	b<=c	a>c

one may conclude (8.4.d).

Figure 8.5: Symbolic Inference Engine

Effect of the Inference Engine

The consistency and completeness techniques, and the forward and backward symbolic inferences engines presented so far in this chapter assumes that fairly standard inference engines are used in processing the expert system knowledge base. These standard algorithms are capable of making all the inferences of propositional logic. Inference engines can depart from these algorithms by either error or design. For example, an inference may stop inference after the first knowledge base conclusion.

The most probable effect of departures in the inference algorithm from the standard is that the inference algorithm makes *fewer* inferences than the standard algorithm. An inference engine error of commission, a false inference, is more likely to be found during testing of the inference engine, while an omitted inference is harder to detect.

Consistency: Omitted inferences do not affect the above methods for finding consistency. The omissions merely mean that there are fewer than expected conclusions to be inconsistent.

Completeness: Omitted inferences *do* affect completeness. The omitted inferences may cause the inference engine not to make an expected conclusion. Where the inference engine is known to omit some propositional logic inferences, it is suggested that completeness be verified using a symbolic version of the same inference engine used on the knowledge base, incomplete though its inferences may be.

Satisfaction Of Specification: Specifications can be verified using symbolic versions of the *same* inference engine used to run the knowledge base. This provides the best insurance that the inference engine will actually make inferences that correspond to those made during the proof of the specification according to the rules of logic.

Inference Engines for Very High Reliability Applications

For applications in which very high reliability is required, it is essential that the inference engine be known to make correct inferences. CLIPS (C Language Integrated Production Systems, first released by NASA in 1988) is the only expert system shell for which it is claimed that the shell has been certified correct.

In order to know that an inference engine makes correct inferences, it is necessary that the inference engine be proven correct. One possible standard of correctness is that the inference engine makes all inferences possible with propositional logic given the information gathered from the user and other sources. This, or an alternative standard, should be proved by carrying out a formal correctness proof on the source code of the inference engine (practical only for small, simple inference engines).

On the basis of this or other proven description of inference engine behavior, algorithms can be constructed like those shown in this chapter for consistency, completeness, and symbolic evaluation for specification satisfaction.

In order to carry out in practice a correctness proof or equivalent description of the inference engine it is necessary that both; the source code of the inference engine be available, and that the source code of the inference engine be short and simple enough to allow a proof to be carried out, given the primitive state of program correctness proofs.

Validating Underlying Knowledge

If there are errors in the knowledge from which a knowledge base is built, there will usually be errors in the performance of the expert system. This chapter discusses methods for validating the knowledge from which a knowledge base is built.

Introduction

If there are errors in the knowledge from which a knowledge base is built, there will usually be errors in the performance of the expert system. There are several ways that the KB can come to represent incorrect knowledge:

- The expert(s) provide incomplete or incorrect knowledge.
- The knowledge engineer fails to correctly understand or code the expert's knowledge.
- Formalizations of knowledge, e.g. using the range of a variable to test for some underlying condition, may fail to capture all instances of the underlying condition.

There are two kinds of validation that must occur on a knowledge base: *logical* and *semantic*. Logical validation checks how the rules and objects work together to reach logical conclusions. In particular, logical validation checks for *consistency*, i.e., that all the conclusions of the knowledge base can be true at the same time. Logical validation also checks for *completeness*, i.e., that the knowledge base reaches a conclusion for all inputs. While earlier chapters of the Handbook focused on logical completeness and consistency, this chapter addresses semantic correctness and completeness.

Logical completeness and consistency are necessary for a knowledge base to be valid. However, logical completeness and consistency are not sufficient for knowledge base validity. For example, Knowledge Base 1 in the Introduction is logically complete and consistent; it contains no logical errors. However, KB 1 makes investment decisions based only on risk tolerance and discretionary income. It uses no information about actual income, debt, fixed expenses, age and other important inputs to good investment decisions. In other words, while KB 1 is logically correct, it is seriously semantically incomplete. To be valid, a knowledge base must be *semantically complete*, i.e., it must base its decisions on all information considered to be relevant by the expert.

An exception is that thorough testing (see Chapter 10, "Testing") may show that some information can be left out without affecting performance. However, knowledge that the expert thinks is needed should be included until testing shows that an expert system performs correctly without that knowledge.

Similarly, a knowledge base can be logically consistent but not semantically consistent for its intended application. *Semantic consistency* occurs when all facts, rules, and conclusions of the knowledge base are true for the application for which the expert system is intended. To illustrate the difference between logical and semantic consistency, consider ordinary Euclidean and spherical geometry. Both are logically consistent mathematical systems from which logically consistent expert systems can be built. However, for everyday life, Euclidean geometry is consistent and spherical geometry is inconsistent with observed facts, while for long distance navigation, the reverse is true.

It is important to note, however, that a knowledge base that is logically inconsistent by definition gives contradictory advice and is therefore semantically incorrect. Likewise, a knowledge base that is logically incomplete fails to provide a solution under some circumstances, and is semantically incomplete. **Logical completeness and consistency are prerequisites for semantic validation of a knowledge base**.

Validating Knowledge Models

Knowledge models are used as a major component of correctness proofs, but these proofs are worthless if the underlying knowledge about the application domain is false in the domain. Therefore, it is important to validate the knowledge models with domain experts.
There are several ways to validate a knowledge model:

- Use a knowledge model from a standards document in the domain. The standards process that created the model can be assumed to have validated the knowledge in the standard.
- Create a knowledge model through the joint development and consensus of a team of recognized experts in the domain. For example, the knowledge base for Quick Medical Reference, an internal medicine advisor, was created by a series of specialized consensus committees in very specialized fields (e.g., Hepatitis B). The knowledge model created in this way can be assumed to contain the best available expertise, and the participation of multiple experts increases the chances that one of them will catch any error that creeps into their discussions.
- Create a knowledge model with a single expert and review the knowledge model with other experts.

When creating a knowledge model using a single expert where correct performance is critical, it is important to validate this knowledge with outside experts not connected with development of the model. The following steps detail the validation process of the knowledge model:

- Present the knowledge model to the outside experts. In some situations it may be advisable to have someone other than the domain expert author of the knowledge model, do the presentation, to insure that professional courtesy does not interfere with critiquing the knowledge model.
- Collect all questions, comments, and objections to the knowledge models, or parts thereof.
- Sort and organize these comments into questions about parts of the knowledge model.
- Organize the questions into a cultural consensus test (see the following sections) to validate individual items.
- Give the test to the outside expert, to determine the extent of agreement on each of the items.
- If some of the items are not validated, perform additional knowledge acquisition and modification of the model to resolve the problems pointed to by the invalidated items. This may include additional discussions, bringing in more experts, literature searches, or redoing parts of the model.

Note that in validating the knowledge model, or in other knowledge validation activities, **it is important to insure that the specialized expertise of experts used in validation cover the intended domain of the expert system**. Most technical fields today are too big and complex to be mastered in their entirety by a single expert, or even a few experts. Therefore, in critical applications, it is important to validate every part of the knowledge base with experts in that particular specialty. An example of this careful validation was the construction of the Quick Medical Reference expert system for internal medicine, and its predecessor systems Internist and Cadeusius. Although the final system contained nearly a thousand diseases, groups of specialists in particular diseases (e.g. hepatitis B) were brought in to collectively discuss and validate the knowledge base in their particular area of special expertise.

After performing these validation steps, it is important to assess the performance of the domain expert (see the later section, Overall Agreement Among Experts). If the current domain expert differs from a consensus of other domain experts, then there are two possible courses of action:

- Replace the domain expert with one who represents a consensus of current domain knowledge.
- Continue the expert system with the disputed knowledge model, with the realization that the system will *not* reflect a consensus of expert knowledge. In this case it is unlikely that the system will perform in a way that matches a consensus of domain experts. Continuing development is a legitimate course in experimental or non-critical systems, but is not advisable in critical expert systems.

An expert system in which the knowledge has not been validated should be used only for applications in which there is no serious consequence of an error by the expert system.

Validating the Semantic Consistency of Underlying Knowledge Items

Even if the expert knowledge has been properly encoded into an expert system knowledge base, the KB will probably produce errors if the underlying expert knowledge is wrong. Therefore, it is important to validate the expert knowledge behind the knowledge base. This is particularly important, because there are a number of ways in which errors can creep into the knowledge on which an expert system is built. Some of these errors are:

- The expert is wrong or out of date; in fact, all experts are probably wrong or out of date on a few points.
- The knowledge base was correct when written, but knowledge has changed.
- The knowledge engineer misunderstood the expert.
- Errors were introduced in maintenance.

Given a fact that has been encoded into the knowledge base, how can one validate that fact represents correct expertise? One approach is to do an experiment such that:

- One outcome is expected if the fact represents currently accepted expertise.
- Another outcome is expected if the fact does not represent currently accepted expertise.
- There is a statistical test that discriminates to an acceptable level of confidence between these two cases.

The specialty of cultural consensus within anthropology provides techniques for validating knowledge in a statistically rigorous manner. These techniques can be applied to knowledge validation for knowledge bases as explained below.
The basic method for validating a knowledge item is:

- Ask a panel of experts whether it is true or false.
- Tally the TRUE/FALSE answers.
- Analyze the results statistically.

Creating a TRUE/FALSE Test

In asking the experts to decide if the knowledge item is true or false, it is important not to bias them by letting the expert know which answer agrees with the current assumption in the knowledge base. Do not, for example say, "You agree with this, don't you?". To present the items for validation in a context in which both TRUE and FALSE are a priori equally likely, disregarding the truth of the item(s) being tested, do the following:

1. Start with a collection of TRUE/FALSE questions about half of which are true and half of which are false, and which are about the domain of the knowledge base. It is important that these environment-creating questions are indistinguishable by the test taker from the questions that actually test KB knowledge.
2. Scatter TRUE/FALSE questions that actually test KB items throughout the list of environment-creating questions.
3. Adjust the test if necessary so that TRUE and FALSE have approximately equal probabilities of being right.

Although this method is adequate for the purposes of this handbook, more detailed information about constructing unbiased tests can be found in literature about survey and test design.

Giving the Test

In applying the cultural consensus method to knowledge base validation, there are some issues that must be handled carefully to get maximum information from the test. First of all, the knowledge engineer must realize and explain to the experts that it is not they but the knowledge base that is being tested. The items on the test represent assertions on which the knowledge base is based, and these are being validated by experts. The reason for using multiple experts is not a lack of confidence in any one expert, but a desire to validate assumptions made in the knowledge base to a statistically significant confidence level. It is important to explain this to all the experts used in knowledge base validation, to insure that no hostility toward the knowledge engineer or the project develops, hostility that would rob the project of valuable contributions to the knowledge base by the expert.
Secondly, the experts used for validation should be carefully instructed to call an item false if it is not always true. This is to guard against the very real possibility that some of the rules in the knowledge base have entry conditions that are too broad. The test can even be given in a form where there are three answers to each question, TRUE, FALSE and SOMETIMES TRUE.

SOMETIMES TRUE and FALSE can be combined as FALSE, i.e., the item was not considered true, when the test is scored.

Formulating the Experiment

Once the test for the knowledge base items has been written, an experiment must be constructed using the test results to validate the items. To do this, the test must be given to a group of experts to evaluate and score the results.
The test must be given to enough expert so that the correctness of each knowledge item based on test results can be distinguished from chance test results. Following is a simple statistical method to validate knowledge base items.

Analyzing the Test Results

A knowledge base item is statistically validated if:

- A majority of the experts answer that the KB item is true (or otherwise supply the test answer(s) that one would predict under the assumption that the experts think the KB item is true).
- The majority is so overwhelming that if the experts did *not* think the KB item was true, the chance of having results that at least this strongly suggest a belief in the KB item is less than some preassigned threshold, traditionally 5% or 1%.

Table 9.1 shows the chance of finding unanimous agreement given the "null hypothesis", that the experimental results are due to chance rather than belief in the KB item.

Table 9.1: Confidence Level

NUMBER OF EXPERTS	CONFIDENCE LEVEL
1	50%
2	75%
3	87.5%
4	94.75%
5	96.88%
6	98.48%
7	99.22%
N	1 - 1 / 2**N

This means that it is probably a good idea to ask at least four experts to verify each important assumption backing up the knowledge base. When four or more experts agree unanimously, the assumption is reasonably validated. Six to seven experts agreeing provides a high level of confidence in the assumption.

Table 9.2 shows the confidence levels results when one expert disagreeing with the rest of the group:

Table 9.2: Confidence Levels with One Expert Disagreeing

NUMBER OF EXPERTS	CONFIDENCE LEVEL
1	0%
2	25%
3	50%
4	68.75%
5	81.25%
6	89.06%
7	93.75%
8	96.48%
9	98.05%
10	98.93%
11	99.41%
12	99.68%

This means that when one expert out of eight disagrees, the KB item is validated to a reasonable level, and is validated to a high level when one expert out of ten disagrees.
In general, if there are N experts of which M disagree, the confidence level achieved by this level agreement is

1 - (1 / 2**N) * SUM(m = 0 to M)combinations(M,N)

where combinations(M,N) is the number of combinations of M objects chosen from N.

This is computed by

combinations(M,N) = M!*(N-M)!/N!

where K! is the factorial of K.

Overall Agreement Among Experts

The above method of validation based on cultural consensus rests on an assumption that the experts share the same basic knowledge, i.e., the same ideas about how to solve the problems covered in the knowledge base, and are validating the specifics of that common approach, as expressed in the knowledge base. Sometimes, however, experts do not agree in their basic knowledge and approach to a class of problems. To detect whether all the experts take the same basic approach to problem solving, observe the following :

1. Cluster the experts: Represent each expert as the vector of answers on the TRUE/FALSE test. Find a clustering of the experts based on these vectors.
2. Test for similarity: Test to see if all the experts belong to the same cluster.
2A. Common cluster: If all the experts belong to the same cluster, then the computation of item confidence presented above remains valid.
2B. More than one cluster: If there is more than one cluster among the experts, analysis of the differences among experts must be conducted, as discussed below. Then the cultural consistency of individual KB items should be retested.
For the small number of experts that are involved in validating a knowledge base, clusters of experts can be determined by hand inspection of the correlation matrix of test answer similarity of experts.

Approaches to Disagreement Among Experts

When experts do not agree, as evidenced by the existence of more than one cluster of experts, the following approaches are useful:
1. Throw away outliers: If it can be determined by interviewing other experts that an expert that is not part of a larger cluster of experts represents a little-held school of thought within their's specialty, and if the more mainstream approach represented by the large cluster of experts successfully solves the problems for which the expert system is intended, eliminate the outlier expert from the validation sample of experts.
2. Choose a valid subset of experts: If two clusters of experts work from totally different assumptions, pick a cluster that achieves optimal results and use them both as the source of domain expertise and experts for validation. Do not try to include two conflicting schools of expertise in the same knowledge base.
3. Use the separate approaches as subsystems: If approaches represented by distinct clusters of experts do better on different subsets of the target domain, it may be possible to build a system in which the differing approaches reside in separate expert subsystems. These subsystems could participate in a weighted vote to determine an overall conclusion, where the weight given to a vote is the heuristically determined confidence factor that a particular subsystem can solve the problem under consideration. Since this approach leads to a more complex, expensive system, it should only be used when the separate approaches are not adequate by themselves.
4. Analyze disagreements: Two or more clusters of experts may be a symptom of unresolved controversies within the professional specialty supplying the expertise for the expert system. In this case, the expert system development team needs to decide if there is enough agreement among experts to build an expert system that gives reliable advice in the domain for which it is intended.

Clues of Incompleteness

Clues that a knowledge base is semantically incomplete may exist within the knowledge base itself. One is that the knowledge base is logically incomplete. Another is that variables, statements, conclusions, etc., are defined but not used. This may indicate that an expert started to supply knowledge that would use them, but never completed that part of the knowledge base. Therefore, the entire knowledge base should be checked for items that are defined but not used, and each one of these should be used or eliminated on expert advice.

Variable Completeness

Variable completeness is a special case of semantic completeness. A knowledge base is variable complete if it uses all of the important input variables in making its conclusions. The steps in checking variable completeness are:

1. Determine and codify what inputs the KB uses in determining each variable and the truth of conclusions.
2. Ask experts to confirm the knowledge codified in Step 1.

There are two ways to determine and codify the variables used in making decisions:

- Computerized analysis of the knowledge base.
- Keeping careful knowledge acquisition and coding notes.

In either case, the goal is to be able to formulate questions of the form

The knowledge base currently uses variables V1,V2,V3...VN to decide X.

- Are there additional variables that should be in this list of inputs? What are they?
- Are there input variables that are not needed? If so, then what are they?

Once these questions have been defined, they should be presented to experts, possibly first to the experts used in building the expert system, and then to independent experts. The process of asking experts about input completeness should be continued until the variable set stabilizes. Then the variable sets should be validated using the technique described above for knowledge item validation.

Semantic Rule Completeness and Consistency

Once the inputs to making decisions have been validated, the actual rules that make each decision should be validated. One problem in validating knowledge bases has been that the size of knowledge bases, and their relative lack of easily perceived structure, makes them difficult for domain experts to read. To lessen this problem, the knowledge base can be partition into the pieces that determine the value of each important variable and conclusion. Each such piece represents the knowledge in the knowledge base about a particular subtopic of the domain, and some conclusion drawn from that subtopic. The expert(s) is asked to examine each piece of the knowledge base separately, and answer the following questions:

- Is the information expressed in the rules that set the value of some particular variable or statement correct?
- Is the information complete? Or are there other conditions to consider, either in individual rules or as new rules?

By focusing the expert's attention on a single variable at a time and the conditions for setting that variable, a large knowledge base is broken down into pieces that are easier to comprehend.
A backwards chaining strategy, can be used to go through the variables and statements in an order that is logical to an expert. Start with the overall outcomes of the knowledge base, and for each pull out all the rules that set that conclusion. Validate these rules. Then do the same for rules that set the conditions in the "if" parts of validated rules. Continue the backward chaining

validation process until validated pieces cover the entire knowledge base. The question, "In this knowledge base piece valid", i.e., is the information correct and complete, can be considered a knowledge item, and validated to the desired level of confidence using cultural consensus, as discussed above. For knowledge bases where reliability is critical, this piecewise validation should be carried out.

Validating Important Rules

Particular emphasis should be placed on validating rules that cover and appear to cover many inputs, or which process critical cases. Rules that appear to cover many cases are those with few atomic formulas in their "if" parts. These rules should be pulled out and validated by experts. To determine which rules typically handle common cases, the knowledge engineer in charge of validation should collect a set of typical input data from one or more experts. Each data set is run on the expert system, keeping track of which rules fired in processing this data. Those rules are presented to the experts for validation.

Exactly the same process is used to validate critical cases; data sets are gathered from experts, the data sets run, and the firing rules validated by the experts.

Validating Confidence Factors

Rule bases may contain assertions about the confidence of conclusions under various conditions, as illustrated by this rule from PAMEX:

if DS = 14

and NOT Deterioration Cause Indicator = Structural Failure

and NOT Deterioration Cause Indicator = Weather Severity

and Skid Number = Low

and DV2 >= 15

and DV15 < 30

then conclude Aggregate Spray, confidence = 0.8

and conclude Open Friction Course, confidence = 0.8

A problem in validating the knowledge base is to insure that the confidence values are semantically consistent. In particular, if three rules with many "if" conditions in common have confidence values for a conclusion of, for example, 0.9, 0.85 and 0.5, it is important to insure that the low confidence factor is justified by domain knowledge. Either through a coding error, or because different experts supplied the confidence factors, it is possible that the large difference is an artifact of building the expert system.

The basic strategy for validating the confidence factors is:

- Predict the confidence factors for rule conclusions by estimating them heuristically from the conclusion confidences of similar rules.
- Compare the predicted confidences to those actually written into the knowledge base.
- Validate the confidences where the predicted and actual differ by more than some threshold.

The first step in implementing this validation consists of rule simplification. The following rule simplifications should be carried out before predicting confidence factors:

- From a rule of the form "if A then B and C", form tow rules, "if A then B" and "if A then C", so that the confidence factors of B and C will be validated separately.
- Normalize the relational operators by:

  -- replacing all < and<= operators with > and >= operators

  -- replacing X>=Y with X>Y OR X=Y

  --replacing X!=Y with NOT X=Y
- Write the “if” parts of rules in disjunctive normal form, i.e., as an OR of ANDs of atomic formulas and negations of atomic formulas.
- From a rule of the form "if A OR B then C" form two rules, "if A then C" and "if B then C", so that the two conditions A and B can be validated separately.

The predicted confidence factors are based only on rules having the same conclusion, i.e., to validate the confidence factor of B ins "if A then B", it is only necessary to look at other rules with conclusion B. Therefore, although the rule simplifications multiply the number of rules, partitioning by conclusion breaks the rules into subsets of manageable size.

Confidence factors are assigned to atomic formulas in rules in a two-step process. The first step is to assign confidence factors to the atomic formula itself. The second step is to modify that confidence factor if the atomic formula is the argument of a NOT. If an explicit confidence factor appears with an atomic formula, use that as the initial confidence factor for the formula in a rule. Otherwise, if an atomic formula appears in a rule, use 1 as the initial confidence factor. If an atomic formula does not appear in a rule, use 0.5 as its confidence factor. Now, having defined confidence factors for the atomic formulas themselves, modify them to account for NOTs as follows: if an atomic formula with initial confidence C is an argument of NOT, its confidence is 1-C; otherwise, its confidence is C.

At this point, a confidence factor has been assigned to every atomic formula in every rule "if" part. Given two rules, R1 and R2 for each atomic formula A, let A1, A2 denote the respective confidence factors. Then define:

distance(R1, R2) = sqrt(SUM(atomic formulas A)(A1-A2)**2)))

i.e., the square root of the sum of squares of difference between corresponding confidence factors. Using this distance, an estimated confidence factor can be conducted by using a generalized regression neural network (GRRN), which is described in the appendix to this chapter.

In interpreting the differences between actual and estimated confidence factors, it must be decided how much difference should trigger validation. Small differences of 0.1 and possibly 0.2 probably represent expert judgments. Larger differences may indicate errors in the knowledge base, but may also indicate valid expertise. Confidence factors with large differences between predicted and actual values should be validated in a 2-step process. First validate the confidence factors with a single expert, e.g., the project domain expert; then, if doubt remains, validate the confidence factors with multiple experts using cultural consensus. Since differences may represent expert knowledge, if the expert validates a confidence factor, it may be accepted as valid, or at least as valid as any other knowledge item supplied by the expert. Like other knowledge items, the single-expert-validated confidence factors may be further validated by multiple experts. However, most of these confidence factor differences reflect the fine structure rather than the major assumptions of knowledge bases, and the priority of validating most of them is small. If the difference is large and the consequence of the difference is judged to be serious, however, the confidence factor should be validated by multiple outside experts.

Testing

This chapter discusses how a simple experiment can be designed to test whether an expert system satisfies a specification.

Simple Experiments for the Rate of Success

The most common statistic measuring how well a system satisfies a specification is to observe the expected fraction of inputs on which the system will satisfy the specification. One can estimate this fraction of an experiment based on the following steps:

1. Select a data sample.
2. Run the expert system on the data sample.
3. Analyze the experimental data.
4.

Selecting a Data Sample

Each specification for the expert system is of the form:

If the input satisfies certain conditions, then the output satisfies certain other conditions.

A sample of N data items for a specification is a set of N data items that satisfies the conditions in the if part of the specification. Furthermore, the sample should satisfy the following additional condition:

> If x is a variable which is thought to affect the reliability of the expert system on the specification, the distribution of x in the sample should approximate the distribution of x in the underlying population.

There are several ways to collect a sample:

random subsample: If a sample of data was put aside for testing during the initial phase of the system lifecycle, the experimenter can draw a random subsample from this sample.

monitoring: Potential inputs can be collected from the environment where the expert system is to perform. A subset of the observed inputs that satisfy the conditions of the specification to be tested becomes the sample for the experiment.

generated input data: Where actual data is not available or practical, a computer program can be used to randomly generate data satisfying the input conditions of the specification.

The size of the sample that should be selected is estimated below.

If a specification has been proved to be satisfied, the existence of the proof may increase the reliability achieved by a test. This effect is also discussed below.

Estimating a Proportion (Fraction) of a Population

If the expert system is run on N data items, and it satisfies the specification on K of those items, then

> K/N = the experimental point (i.e., single number) estimate of the proportion of the underlying population satisfying the specification

If the sample size N is sufficiently large, the distribution of sample proportions (the values of K/N) is approximately normal. This occurs when both the following conditions are true:

$$K = N*(K/N) > 5 \qquad (1)$$

$$N*(1-K/N) > 5$$

When this is true, the standard error of the proportion is

$$s\_e(K/N) = sqrt((K/N)*(1-K/N)/N) \qquad (2)$$

When the conditions (1) for normality are not satisfied, the Poisson distribution, discussed below (see page 131), can be used to estimate the satisfaction of a specification.

The Confidence Interval of a Proportion

In this section the goal is to find an interval of proportions (fractions) of a population since most of the time the observed satisfaction of the specification for a new sample will be in the interval. In particular, the goal is to find an interval such that the probability of the observed satisfaction being in the interval is sat for sat close to 1.

The steps in computing the interval are:

1. Conduct the experiment to test the specification. Observe

   the sample size N

   the number of times K the specification was satisfied on the sample

   Conduct enough trials so that the requirements for approximate normality are satisfied.

2. Compute s_e(K/N)

3. From a statistical table, find the standard normal deviate (snd) of sat, often called the "z-score" and denoted by z.

The standard normal deviate is a multiple of the standard error marking out a central region of the normal distribution that contains a given fraction of the total area (which is 1) under the normal distribution. In particular, z(sat) is the number such that the area under the normal distribution between -z(sat) and z(sat) is sat, i.e.,

$$\text{INTEGRAL}\Big|_{-z(sat)}^{z(sat)} normal(x)dx \qquad (3)$$

where normal(x) is the standard normal distribution,

$$n(x) = (1/2*pi)* exp(-x^2/2) \qquad (4)$$

While there is no closed form for z, tables of z-scores are widely available in statistics texts; typical values are shown below:

sat	*z(sat)*
50%	0.68

75%	1.15
90%	1.65
95%	1.96
98%	2.33
99%	2.58
99.5%	2.81
99.8%	3.08

When K successes are observed in N trials, the sat confidence interval is

K/N +/- z(sat)*s_e(K/N) (5)

Choosing Sample Size

The goal for a system developer is often to show that a system will perform at least as reliably as some threshold. Statistically, this means that with a confidence of at least C, a specification is satisfied in at least fraction F of a sample on which the specification applies. A typical statement of this form is

> The expert system correctly diagnoses pavement maintenance remedies at least 90% of the time with 95% confidence.

This means that if another experiment using the same sample size was conducted, at least 95% of the time the measured fraction on which the specification is satisfied would be at least 90%. Given a desired fraction F and a confidence level C, the user can obtain the size of sample needed to achieve these parameters in the following way:

1. Conduct a small initial experiment to estimate the fraction on which the specification is satisfied. This initial estimate will be denoted F0. If F0<F and the sample size of the initial experiment guarantees that there is reasonable confidence in F0, the expert system does not satisfy the proportion F. If F0 is equal or only slightly larger than F, the size of the experiment needed to narrow the confidence interval around F0 to exclude F will be unreasonably large; in practice, it will be impossible to statistically validate the satisfaction with proportion F and confidence C.

2. Given that F0 >F, compute:

s_e = (F0-F)/z(C) (6)

To achieve F and C, choose a sample size such that the standard error is less than or equal to s_e. This means choosing an N such that

sqrt(F0*(1-F0) / N) =< s_e (7)

or

N >= F0*(1-F0)/s_e^2 (8)

For example, if

preliminary experimental proportion (F0) = 93%

minimum acceptable proportion (F) = 90%

confidence interval = 95%

then

s_e = (93%-90%)/z(95%) = 0.03/1.96 = 0.153

and

N >= 93% * (1-93%) / 0.153^2 = 277.9

This estimate of sample size is approximate, because the preliminary proportion F0 used in the computation is only the result of a small preliminary experiment, and will contain some random error. Therefore, the experimenter should, if possible, design an experiment so that an initial experiment can be continued by testing more data. This is possible provided that the probability of drawing any data item in the continuation of the experiment is the same as drawing that data item in the initial experiment.

Estimating Very Reliable Systems

For systems that do not fail often, it is difficult in practice to observe the five or more failures that causes the proportion to be approximately normally distributed. In this case, use the Poisson distribution as follows to estimate a confidence interval for the satisfaction proportion.
The Poisson distribution describes the number of occurrences of some random event in given interval of time or region of space. For example, the number of fish over any square meter of a lake, where the lake bottom is uniformly attractive to fish, is approximately Poisson distributed. The formal requirements for an occurrence to be Poisson distributed include:

- Each occurrence is independent of the others.
- Each interval can potentially contain an infinite number of occurrences.

In practice, the second requirement can be approximated if a large number of occurrences can occur in a region; what is "large" for this purpose will be estimated below.
If the average number of occurrences in a region is L, the probability of finding k occurrences is:

P(k) = exp(-L)*L^k/k! (9)

The probability of K or more occurrences is:

SUM exp(-L)*L^k/k! (10)

k>=K

a series that converges geometrically once L/k < 1.

For testing a specification, a region is defined to consist of N trials, where N is a number such that N or more occurrences is very unlikely, as computed by (10).
The requirement that a specification is satisfied at a proportion at least F, means that

(N-Fail)/N > F (11)

where N is the number of trials in a region, and Fail is the number of failures observed in N trials. This means that the number of failures Fail should satisfy

Fail < (1-F)* N (12)

This says that the number of failures should be less than the acceptable failure rate, 1-F, times the number of trials in a region. Using (9) the user can compute the probabilities of observing failure rates satisfying (12). Denote the sum of these probabilities by

$$P = \sum_{k < (1-F)*N} P(k) \qquad (13)$$

Then if P >= F, the expected success rate is at least F.

How a Proof Increases Reliability

Suppose that in a Hoffman region that a specification has been proved, and that the specification has been verified in a single experimental trial

The question to be asked then is:

> What is the probability that the specification would fail on a new trial with inputs in the Hoffman region?

By the definition of Hoffman region, all atomic formulas that determine the computational path of the system have the same truth values for the inputs of the second trial. Therefore, the output on the new trial should be identical to that of the first, on which the specification was satisfied. The only way for the outcome of the second trial to be different is for a system error to have occurred. Therefore, the probability of a failure on a Hoffman region for which both a proof and a single trial experimental verification is available is the probability of an underlying computer hardware or system software error occurring during the computation, as the Pentium bug illustrates, this is a small, but non-zero probability.

In order for a fielded system to perform reliably, the probability of a computer system error must be kept small. However, this computer system error probability applies approximately equally to all knowledge bases. Therefore, once the underlying reliability of the computer system is established, resources should not be expended testing for this error. In particular, where a proof exists, one experimental trial per Hoffman region is sufficient to verify a specification with probability 1-Fc, where Fc is the probability of a computer system error.

Evaluation and Other Management Issues

Evaluation, which includes field testing, addresses the issue “is the system valuable?”. This is reflected by the acceptance of the system by its end users and the performance of the system in application. This chapter addresses this issue and some general guidelines which help in the distribution and maintenance of expert systems.

Evaluation

Pertinent issues in evaluation are:

1. Is the system user friendly and do the users accept the system?
2. Does the system give "correct" results and is the logic of the system correct?
3. Does the expert system offer an improvement over the practices it is intended to supplement?
4. Is the system useful as a training tool?
5. Is the system maintainable by other than the developers?

There are no set procedures for the evaluation of expert systems. In fact, it may be quite difficult to achieve. Sometimes evaluation is ignored until very late in system development (always with disastrous results). However, there are some things which can be done to make this process more effective. First, for systems under development, the developer should design for testing. For completed prototypes this is impossible, however, workshops and substantial interaction with the target end users can make the process of field testing much more valuable. It is critical that the end users be kept aware of how important their contributions are and that their efforts are greatly appreciated.

Workshops and follow-up efforts with end users and experts can provide valuable improvements to an expert system and incentives for its use. After the expanded prototype has been constructed, based on knowledge from the experts and end users on the development team, a workshop or series of workshops involving a larger community of experts and end users will usually result in major improvements to the system. The participants should be introduced to the computer program (expert system) and to the general concepts used in the development of the system. During the workshops, the knowledge structure and the parameters used in the decisionmaking process and their interrelationships should be reviewed. Expected benefits from workshops include:

- A verification check, i. e., review and improvement of the logic used in developing the system rules.
- Enhancement of the knowledge base, i. e., finding oversights in the rules and the relationships between them.
- More user oriented and user friendly interface.
- Development of vested interest in the user community, i. e., establishment of a cadre of field users who want the system to succeed.
- A better and more useable system.

As part of the evaluation process, the assumptions made during planning of the system should be reexamined. For example, during the planning process, assumptions on the frequency of use, availability of input data, usefulness of system output, ease of use, etc. should have been documented. These assumptions should all be tested during field trials.

The personnel who provide field support for the field evaluation must be carefully screened and selected. The wrong selection of support personnel can sabotage even the very best of expert systems. The field support personnel must have the following capabilities:

- Expert knowledge of the domain of the expert system. During this phase the field support staff may be called upon to not only provide support for the expert system that is dependent on domain knowledge, but to answer domain specific questions from the end users. A non domain expert can do irreparable damage to the credibility of the system during this phase.
- Sophisticated computer skills. The field support personnel have to go into a strange environment and correctly and efficiently install an expert system and then provide training using this unfamiliar equipment. Installation problems can always be expected.
- Excellent language skills in the language of the domain experts and end users. The field support personnel must be able to express concepts clearly and concisely to the users at their duty stations and in their language to understand the nuances that the end users try to convey. Anything less than FLUENCY in the language of the end users is not acceptable. In addition to language skills, the field support personnel must have excellent interpersonal skills.

Whenever possible, field testing should be conducted using the same end users who participated on the development team and/or verification and validation activities. This eliminates any indoctrination period and builds on the vested interest of the end users. Some of the activities to be conducted in field testing are:

- Field operating environment - It is necessary to become acquainted with the field operating environment the expert system will be installed in. Even though operating environment was considered through the domain experts and end users, it will, in fact, appear different to every observer. There will be factors that will affect operations that were not considered before the actual installation and field trials begin.
- Installation of expert systems - The expert system will have to be installed on the equipment provided by the end user/tester. This step will usually not be routine as computers or operating systems, etc., may have to be reconfigured to accommodate the expert system.
- Additional training for the end user/tester will need to be provided. Regardless of prior training, the end user/tester will need support to overcome the various nuances that appear during field test conditions. At this point it is also necessary to define the roles of various parties who participate in the field testing. Who runs the expert system? Who approves the use of expert system recommendations? Who applies the recommendations from the expert system? Who actually collects field data? Who does the preliminary screening and preparation of any data collected?
- After the expert system is installed on the end user's computer, the requirements and terms for the field tests must be reviewed. Competing demands on the end user are

always more extensive at the end user's duty station than they were during previous meetings. It is critical to get a renewed commitment. The commitment to support the end user/tester in every way possible must also be reaffirmed. The end user must understand how critical their support is and that the sponsor values it.

- Specific test cases should be identified and discussed. This includes previously identified sample test cases and new conditions that may be encountered in the field.
- The formal mechanism for incorporating findings from the field tests should be developed and reviewed in detail with the end user/tester. Also, the formal mechanism for sharing information among end users/testers must be developed and discussed with all end users/testers.
-

Distributing And Maintaining Expert Systems

Once the expert system development effort has been completed, the tasks of distribution and maintenance begin. Although there are no fixed rules governing these tasks, there are some general guidelines which can make these tasks easier to perform.

Distribution

There are three major criteria that a developer should follow in order to facilitate the distribution of a given expert system:

1. Identify and involve the user community before starting the development of an expert system. This should insure that the expert system actually solves a real set of problems. Also, it will give the user community a vested interest in the testing and application of the system.
2. Develop the system using standard hardware and software. Although there are a number of exotic pieces of expert systems hardware and software, the cost of these items is often high and the uncertainty of survival of these products in the market place makes it unreasonable to expect potential users to procure them just to use an expert system.
3. Use development software that does not require distribution licenses or where an unlimited distribution license can be purchased for a reasonable fee. The time and funding expended on paying fees for each system distributed, as required for many available development tools, can become an unwanted administrative and financial burden.

Maintenance

The task of system maintenance is one that should be planned for from the inception of the expert system development project. Maintenance can be facilitated by following a few good development rules. These include:

1. Design the expert system to be as transparent as possible. Since the system maintenance will probably not be conducted by system designers, it is necessary that the structure of the expert system be as straightforward and clear as possible.

2. The developers should use logical and understandable names for objects and knowledge structures within the system and avoid the use of cryptic names and obscure abbreviations.
3. The developers should also avoid the use of overly complex and obscure software structures, even though their use may provide some type of performance benefits.
4. Simplicity should be one of the guiding principals in the development effort.

The system must be well documented. The documentation should be produced as the system is developed, not as an after thought once the system is finished. The knowledge engineer should:

- Identify where the system's knowledge resides (e.g., in the knowledge base in the form of facts and rules and in the inference engine in the form of heuristic search techniques).
- Document the inference procedures that the system uses in producing solutions.
- Assure that as part of the documentation an explicit model of the problem solver is included.
- Assure that the documentation also provides a comprehensive and well documented test procedure for the system.

The expert system itself should contain an extensive set of both user "help" text and explanation text which explains how the system produced a given solution. The documentation and the help and explanation text should be produced during the development phase and not added after the system has been built. One of the guiding principles that developers should use is "a poorly documented system will have a short useful life."

Each version of a given expert system should have a version number. This will make it easier to provide users with updated copies of the system.

Establish a mechanism for soliciting, receiving, and acting upon feedback from the user community. This will facilitate the identification and removal of "bugs" in the system and will also make it easier to retrofit the system to satisfy specific user community needs after the system has been distributed to the user community.

Appendix

Symbolic Evaluation of Atomic Formulas

A common type of atomic formula in a rule-based expert system is of the form

<VARIABLE> <RELATION> <CONSTANT>   (A.2.1)

where <RELATION> is one of the relations, =, <, >, <=, or >=.
The following table shows when an atomic formula of this form is true or false given conditions on <VARIABLE> of same form, A.2.1.
In this table, "TRUTH CONDITION" specifies conditions under which the atomic formula is true for all numbers in the interval. "FALSE CONDITION" specifies conditions under which the atomic formulas are false for all numbers in the interval. The following restrictions on the variables a, b and c apply:

a is in [-INFINITY,INFINITY)

b is in (-INFINITY,INFINITY]

c is in (-INFINITY,INFINITY)

ATOMIC FORMULA	TRUTH CONDITION	FALSE CONDITION
(a,b)<c	b<=c	a>=c
[a,b)<c	b<=c	a>c
(a,b]<c	b<c	a>=c
[a,b]<c	b<c	a>=c
(a,b)=<c	b<=c	a>=c
[a,b)=<c	b<=c	a>c
(a,b]=<c	b<=c	a>=c
[a,b]=<c	b<=c	a>c
[a,b]=c	a=b=c	a != b or a != c or b != c
(a,b)>c	a>=c	b<=c
(a,b]>c	a>=c	b<c
[a,b)>c	a>c	b<=c
[a,b]>c	a>c	b<c
(a,b)=>c	a>=c	b<=c
[a,b)=>c	a>=c	b<=c
(a,b]=>c	a>=c	b<c
[a,b]=>c	a>=c	b<c

General Regression Neural Nets

A general regression neural net (GRNN) is a method for estimating a function from a set of its values at particular points in its domain. Although the GRNN algorithm can be put in the form of a neural net, it is best understood as an interpolation. In particular, GRNN interpolates from known data points by computing a weighted average of nearby points. The weights in this average decay exponentially with distance from the point where the function is being estimated.

Notation

The following notation will be used:

- Upper case letters (e.g., P, X, X2 etc) denote points in the input space.
- Lower case letters with subscripts represent numbers for different fields (axes) for the point named by the corresponding upper case letter. The subscripts identify which axis the number represents. Axis subscripts follow any subscripts that are part of the name of the point. Examples: p2, xi, x2i.

Prerequisites for GRNN

To carry out a GRNN computation, it is necessary that a distance function be defined between any two points in the input domain. The Euclidian distance function works well for GRNNs, although any distance function can be used. The Euclidean distance is defined by

d(P1, P2) = sqrt(SUM(over fields i)(p1i - p2i)**2)))

A weight function from pairs of points to real numbers is defined as follows:

w(P1,P2) = exp(-K*d(P1,P2))

In other words, the weight assigned to P2 for a GRNN estimate at P1 decays exponentially with the distance from P1 to P2. K is parameter that determines how fast the decay occurs.

The GRNN Interpolation

Following is the GRNN interpolation of a function fn:

grnn(P1) = SUM(all points P2 in data set)w(P1,P2)*fn(P2))

This says that the GRNN estimate of *fn* at a point is the weighted average of the known function values, where the weights decay exponentially with distance from the point where the estimate is being made.

References

Agarwal, R., (1993) Formal validation of a knowledge-based system using a variation of the Turing Test, *Expert Systems With Applications, 6,* pp. 181-192.

Agarwal, R., Kannan, R., & Tanniru, M. (1993) Formal validation of a knowledge-based system using a variation of the Turing Test, *Expert Systems With Applications, 6,* p. 181-192

Agarwal, R & Tanniru, M (1992) A structured methodology for developing production systems, *Decision Support Systems, 8,* 483-499.

Agarwal, R & Tanniru, M (1992a) A Petri-Net Based Approach for Verifying the Integrity of Production Systems, *International Journal of Man-Machine Studies, 36, (3),* pp 447-468

Amla, N. & Ammann, P. (1992) Using Z specifications in category partition testing, in *COMPASS '92; Proceedings of the Seventh Annual Conference on Computer Assurance, Systems Integrity, Software Safety, Process Security,* June 15-18, 1992, Gaithersburg, MD.: Piscataway, N.J. IEEE.

Aougab, H., Schwartz, C., & Wentworth, J., (1988, March) Expert System for Management of Low Volume Flexible Pavements, *Computing in Civil Engineering: Microcomputers to Supercomputers; Proceedings of the Fifth Conference,* March 29-31, Alexandria, VA. pp 759-768

Bahill, A., Jafar, M. & Moller, R. (1987) Tools for extracting knowledge and validating expert systems, *IEEE International Conference on Systems, Man and Cybernetics, 2,* 857-862.

Botton, N., Kusiak, A., Raz, T. (1989) Knowledge bases: integration, verification, and partitioning, *European Journal of Operational Research, (42),* p. 111-128.

Chandrasekaran, B. (1983) On evaluating AI systems for medical diagnosis, *AI Magazine, 4 (2)* p. 34-38

Cheng, Y. & Fu, K. (1985) Conceptual clustering in knowledge organization, *IEEE Trans. Pattern Analysis Machine Intelligence, PAMI-7,* p. 592-598.

Childress, R. (1992, Feb.) AAAI'91, Part III, Knowledge-Based Systems: Verification, Validation, and Testing, *IEEE Expert,* p. 73-75

Coenen, F., Bench-Capon, T., & Kent, A. (1994) A Binary encoded incidence matrix representation to support KBS verification in Plant, R. T., Chair, *Validation and Verification of Knowledge-Based Systems;1994 Workshop Program, Seattle, July - August, 1994: Workshop Notes*: American Association for Artificial Intelligence

Cragun, B., Steudel, H.J. (1987) A decision-table-based processor for checking completeness and consistency in rule-based expert systems, *International Journal of Man-Machine Studies, 26,* p. 633-648

Daniel, Wayne W. Biostatistics: a Foundation for Analysis in the Health Sciences. Wiley, New York, 1978.

Davis, R. (1976) *Applications of Meta Level Knowledge to the Construction, Maintenance and Use of Large Knowledge Bases,* Report STAN-CS-76-552, Stanford, CA., Stanford Artificial Intelligence Laboratory, Stanford University

Duda, R.O., Hart, P.E., Barrett, P., Gaschnig, J.G., Konolidge, K., Reboh, R. and Slocum, J., (1979) *Development of the Prospector Consultation System for Mineral Exploration.* Final Report SRI Project 6415, Menlo Park, SRI International.

Fenton, N., Kaposi, A. (1987) Metrics and software structure, *Information and Software Technology, 29 (6),* p. 301-320.

Franklin, W. R., Bansal, R., Gilbert, E., Shroff, G. (1988) Debugging and tracing expert systems, in *Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences,* v.3, Los Alamitos, CA: IEEE Computer Society Press, pp. 159-167.

Freeman, M. (1985) Case study of the BEACON project, in *Expert Systems and Prolog*, IEEE Videoconferences, Seminars via Satellite.

Gaschnig, J. (1979) Preliminary performance analysis of the Prospector consultant system for mineral exploration in *Proceedings Sixth International Joint Conference Artificial Intelligence,* v.1, San Mateo: Morgan Kaufmann, pp. 308-310

Gaschnig, et al. (1983) Evaluation of expert system: issues and case studeis, in Hayes-Roth, F., Waterman, D. A. and Lenat, D. B. eds., *Building Expert Systems*, Reading, MA: Addison-Wesley,

Geissmann, J.R., Schultz, R.D. (1988) Verification and validation of expert systems, *AI Expert, 3 (2),* pp. 26-33.

Ginsberg, A. (1988) Knowledge-base reduction: a new approach to checking bases fro inconsistency & redundancy, *in Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88), vol. 2,* Menlo Park, CA: AAAI Press, p. 585-589.

Gupta, U.G. (1993) Validation and verification of knowledge-based systems: a survey, *Journal of Applied Intelligence,3*, p. 343-363.

Gupta, U.G., Biegel, J.E. (1990) RITCaG: Rule-based intelligent test case generator in *Proceedings AAAI Workshop on Validation and Verification of Expert Systems,* Boston, July 29 (unpublished)

Harrison, P. R., Ratcliffe, P. A. (1991) Towards standards for the validation of expert systems, *Expert Systems With Applications, 2*, p. 251-258.

Hickham, D.H., Shortliffe, E. H., Bischoff, M.B., Scott, C.A., & Jacobs, C. D. (1985) The treatment advice of a computer-based cancer chemotherapy protocol advisor, *Annals of Internal Medicine, 103* (6) Part 1, 928-936.

Hoel, Paul G. Introduction to Mathematical Statistics. Wiley, New York, 1954.

Jacob, R. & Froscher, J. (1986) Developing a software engineering methodology for rule-based systems, in *Proceedings 1985 Conference on Intelligent Syst. Machines*, Oakland Univ., pp. 179-183

Jacob, R. & Froscher, J. (1990) A Software engineering methodology for rule-based systems, *IEEE Transactions on Knowledge and Data Engineering, 2* (2), 173-189.

Kang, B.H., Gambetta, W., & Compton, P, (1994) Verification and validation with ripple down rules, in Plant, R. T., Chair, *Validation and Verification of Knowledge-Based Systems;1994 Workshop Program, Seattle, July - August, 1994: Workshop Notes:* American Association for Artificial Intelligence

Kang, Byeong, Windy Gambetta and Paul Compton. Verification and validation with ripple down rules. Workshop Notes, Validation and Verification of Knowledge-based Systems, American Association for Artificial Intelligence, July 31-Aug. 4, 1994.

Kang, Y. & Bahill, T. (1990, Feb.) A Tool for detecting expert-system errors, *AI Expert,* p. 46-51.

Koski E.M., Makivirta, A,, Sukuvaara, T., Kari, A. (1991-92) Development of an expert system for haemodynamic monitoring: computerized symbolization of on-line monitoring data, *Int J Clin Monit Comput 8(4):*289-93

Lane, A. (1989, June) An end to dueling rules, *Byte*, 303-308.

Lehmann, E.D., Deutsch, T., Roudsari, A. V., Carson, E.R., Sonksen, P.H. (1993) Validation of a metabolic prototype to assist in the treatment of insulin-dependent diabetes mellitus, *Medical Informatics, 18 (2),* p. 83-101

Liu, N. K., Dillon, T. (1991) An approach towards the verification of expert systems using numerical Petri nets., *International Journal of Intelligent Systems 6(3)* p. 255-276

Mengshoel, O.L, (1993, June) Knowledge validation: principles and practice, *IEEE Expert: Intelligent Systems and their Applications, 8 (3),* 62-68

Miskell, S.G., Happell, N., Carlisle, C., (1989) Operational Evaluation of an Expert System: The FIESTA Approach, *Heuristics, The Journal of Knowledge Engineering, 2 (2),* Systemsware Corporation, Rockville, Maryland.

Nazareth, D. L. (1989) Issues in the verification of knowledge in rule-based systems, *International Journal of Man-Machine Studies, 30,* 255-271.

Nguyen, T. A., Perkins, W. A., Laffey, T. J., Pecora, D., (1987) Knowledge base verification, *AI Magazine, 8 (2),* p. 69-75.

O'Leary, D. (Nov.-Dec. 1988) Methods of validating expert sytems, *Interfaces,* 18 (6) p. 72-79

Parsaye, K. (1988) Acquiring and verifying knowledge automatically, *AI Expert, 3 (5),* p. 48-63.

Potter B, Ronan S.G. (1987 Jul) Computerized dermatopathologic diagnosis. SO - *J Am Acad Dermatol 7(1):*119-31

Raz, T. & Botten, N. A. (1992) The Knowledge base partitioning problem: mathematical formulation and heuristic clustering, *Data and Knowledge Engineering, 8,* p. 329-337.

Renard, F. X., Sterling, L., Brosilow, C. (1993) Knowledge verification in expert systems combining declarative and procedural representations, *Computers and Chemical Engineering, 17 (11)* pp. 1067-1090

Shwe, M.A., Tu, S.W., Fagan, L.M (1989 Jan) Validating the knowledge base of a therapy planning system. *Methods Inf Med;28(1)* 36-50

Suwa, M., Scott, A., Shortliffe, E. (1982) An approach to verifying completeness and consistence in a rule-based expert system, *AI Magazine (3) 3,* p. 16-21.

Traylor, B., Schwuttke, U. & Quan, A. (1994) A Tool for automatic verification of real-time expert systems, in Plant, R. T., Chair, *Validation and Verification of Knowledge-Based Systems;1994 Workshop Program, Seattle, July - August, 1994: Workshop Notes:* American Association for Artificial Intelligence

Valiente (1992, Jan.) Using layered support graphs for verifying external adequacy in rule-based expert systems, *Sigart Bulletin. 3 (1)* p. 20-24.

Vanthienen, J., Dries, E. (1993) Illustration of a decision table tool for specifying and implementing knowledge based systems*, Proceedings, 5th International Conference on Tools with Artificial Intelligence TAI '93,* Boston, Mass., Nov. 1993., pp. 198-205

Verdaguer, A., Patak, A., Sancho, J.J., Sierra, C., Sanz, F. (1992) Validation of the medical expert system PNEUMON-IA, *Computers and Biomedical Research, 25,* p. 511-526.

Wentworth 1989, (1989), "Overview of Artificial Intelligence Applications in Transportation," *Proceedings of the Third Annual Conference on Micro- computers in Transportation*, San Francisco, California.

Yu, V.L., Fagan, L.M., Wraith, S.M., Clancey, W.J., Scott, A.C., Hannagan, J.F., Blum, R.L., Cohen, S.N., (1979) Antimicrobial selection by a computer: a blinded evaluation by infectious disease experts. *Journal of the American Medical Association, 12 (242),* 1279-1282