APPLICATIONS OF RULE-BASE COVERAGE MEASURES TO EXPERT SYSTEM EVALUATION

BY VALERIE B. BARR

A dissertation submitted to the
Graduate School – New Brunswick
Rutgers, The State University of New Jersey
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
Graduate Program in Computer Science

Written under the direction of
Dr. Casimir Kulikowski
and approved by

_________________________________________________________________

_________________________________________________________________

_________________________________________________________________

_________________________________________________________________

New Brunswick, New Jersey
May, 1996
© 1996
Valerie B. Barr
ALL RIGHTS RESERVED
Applications of Rule-Base Coverage Measures to Expert System Evaluation

by Valerie B. Barr

Dissertation Director: Dr. Casimir Kulikowski

A rule-base coverage analysis method has been developed which provides an assessment of both the rule-base under review and the test set that has been used for evaluation. Lack of coverage can result from either incompleteness of the test data or errors in the rule-base. A series of heuristics have been developed which use coverage information and meta-knowledge about the larger population to suggest additional test cases from the population, in the event that the initial test set is incomplete. This forms the basis of an incremental approach which allows us to both increase completeness of the test suite and improve coverage of the rule-base.

Rule-based system testing usually faces the difficult dual problems of incompleteness and errors in both the rule-base and the test data. Performance of a system on a limited suite of test data is never sufficient to predict performance on a larger set of data in routine use without additional assumptions. An important one of these is the assumption of representative coverage of the population for which the system is intended. The heuristic approach to test data selection is demonstrated using information generated by TRUBAC, a tool which implements the coverage analysis methods. We have applied these techniques to analyze a number of prototype systems for diagnosis of rheumatological diseases. In addition, we demonstrate the use of coverage information
to identify class dependencies and guide rule-base pruning. We also introduce a complexity metric for rule-bases. Finally, we discuss extensions of the coverage measures for rule-based systems with dynamic computation of certainty factors.
Acknowledgements

This dissertation represents the culmination of a process that began in 1986 when David Beilin, now at North Carolina A&T, suggested that I apply for a teaching job at Pratt Institute, where he was working at that time. Thanks to David I discovered that I loved teaching, but I felt very strongly that I would not return to graduate school without a research topic that really interested me.

In 1987 Elaine Weyuker, now of Bell Laboratories, gave me the opportunity to work on data flow testing research, specifically testing ASSET [FW85], a data flow testing tool for Pascal programs. Several months later, while teaching an Artificial Intelligence course, I began to wonder about the possibility of applying a data flow testing approach to expert systems. At that point I realized I had a research topic and should consider a return to graduate school for my PhD.

I entered Rutgers as a part-time student, and I thank Diane Souvaine, at Rutgers, for convincing me that I should consider attending full-time instead. I also thank Diane for helping me develop a deeper understanding and appreciation of algorithms and algorithmic analysis in the courses I took with her, as well as teaching me a lot about teaching by her example in the classroom. And I also thank her for being a mentor and a source of advice throughout my time at Rutgers.

As I began to zero in on my ultimate research topic, I had to examine what had been done in the general area of overlap between data flow analysis and functional and logic programming languages. I thank Barbara Ryder who, in addition to chairing my exam committee and sitting on my thesis committee, also supervised a great deal of the preliminary work which helped me focus my dissertation research.

As my research focused more specifically on the testing of expert systems, I asked Casimir Kulikowski to serve first on my exam committee, and subsequently as my
committee chair and primary adviser. I thank him for being willing to do so and for generously sharing his deep knowledge of the field of expert systems, as well as his editorial expertise. His guidance has been steady, never telling me the answer but always sending me in the right direction. This project would not exist in its present form without his help.

I also thank the remaining members of my thesis committee for their help and insights. Haym Hirsh has continually provided me with helpful discussions of various issues that came up as I developed my approach and tool, as well as made editorial suggestions that have helped make this a more readable document. Allen Ginsberg raised a number of points that led to greater clarity, and engaged me in some very enjoyable philosophical discussions.

Of course, part of this process involved courses and exams and TAing and getting to know my fellow graduate students. I found the community of graduate students in the Computer Science department at Rutgers to be very collegial. More people have helped me during the last six years than I can list here, from my exam study group to people who sent Latex tips in response to posting. I would, however, like to particularly recognize the help and support of Chun Wai Liew and Deborah McGuinness.

In the age of Internet, it is possible to get help and information from around the globe. I would like, in particular, to thank Milan Merhar, of Xyplex Corp., for his speedy advice concerning the finer points of C and UNIX when I encountered stumbling blocks during the implementation of TRUBAC. I would also like to thank the participants in the VAVTALK mailing list of verification and validation researchers, particularly Alun Preece, who responded to my queries for rule-bases I could test. Finally I would like to note that many senior level computer scientists around the world took time from their research to respond to my early queries for pointers to the literature and my later queries about test data.

Returning to graduate study after ten years in industry was quite a change and there are a number of people and institutions who I would like to recognize for their help along the way. Phyllis Frankl, of Polytechnic University of New York, has helped me in many capacities during this process, both as friend and colleague. With her extensive
work in software testing she has frequently served as a sounding board for my ideas, regarding both the theory and the implementation. In addition, the CS department at Polytech has generously provided me with office space and access to their computing facilities, thereby allowing me to spend less time commuting and more time working.

Mount Holyoke College awarded me a fellowship to support a portion of my graduate studies and followed that with an appointment for three semesters as a Visiting Instructor in their Department of Mathematics, Statistics and Computer Science. The opportunity to teach an exciting and talented group of students and interact with my colleagues there was a great help as I was starting this research.

Ultimately I must thank my family. Camilla Yohn-Barr was born after this project was well underway, and she has given me ample opportunity to ponder real versus artificial intelligence and the marvel of human knowledge acquisition. This project would never exist without help from Susan Yohn. From my decision to return to graduate school, through to the finished dissertation, Susan has supported me in more ways than I can possibly enumerate here. To say I am grateful is hardly sufficient.
# Table of Contents

Abstract ................................................................. ii
Acknowledgements ....................................................... iv

List of Tables ............................................................ xi
List of Figures ............................................................. xii

1. Introduction and Overview .......................................... 1
   1.1. Introduction .................................................. 1
   1.2. Terminology .................................................. 10
   1.3. Outline of the Dissertation .................................. 13

2. Foundations .......................................................... 15
   2.1. Guidelines for Expert System Analysis ..................... 15
       2.1.1. Kinds of analysis ..................................... 17
       2.1.2. Testing Criteria ..................................... 19
       2.1.3. Test Data ............................................. 20
   2.2. Traditional Evaluation Methods .............................. 21
   2.3. Control Flow Analysis ....................................... 23
   2.4. Control Flow Metrics ....................................... 26
   2.5. Data Flow Testing of Procedural Programs ................ 27

3. Testing with Rule-Base Coverage ................................. 29
   3.1. Overview of Testing with Rule-Base Coverage Measures .... 30
   3.2. Rule-base Representation ................................... 31
   3.3. Rule-base coverage measures ................................ 33
3.4. Rule-base Analysis ................................................................. 40
  3.4.1. Graph Construction ......................................................... 40
  3.4.2. Static Analysis ............................................................... 41
  3.4.3. Dynamic Analysis ........................................................... 43
  3.4.4. Evaluating the Rule-Base Coverage Measures ................. 44
   Each-class ................................................................. 44
   Each-hypothesis ............................................................ 44
   Each-class-every-finding ................................................... 44
   Each-class-every-sub ........................................................ 45
   All-edges ............................................................... 46

3.5. Error Identification .............................................................. 47

3.6. Heuristics for Test Data Selection ........................................ 49

3.7. Class Dependence ............................................................... 53

3.8. Rule-Base Pruning .............................................................. 55

3.9. A Metric for Rule-Based Systems .......................................... 56

3.10. Comparison of Testing Criteria ........................................... 63

4. The TRUBAC Tool ................................................................. 68
  4.1. Rule-base Language .......................................................... 69
  4.2. Graph Construction ........................................................ 71
  4.3. Static Analysis ............................................................... 77
  4.4. Dynamic Analysis ........................................................... 77
  4.5. Applications of TRUBAC .................................................. 80
   4.5.1. EXPERT Rule-Bases .................................................. 80
          Classes ............................................................... 80
          Findings ............................................................ 81
          Rules ............................................................... 82
          Results of using TRUBAC on EXPERT rule-bases ............ 86
   4.5.2. PROLOG Rule-Bases ................................................. 89
Results of using TRUBAC on Prolog rule-bases .................. 92

4.6. Use of Heuristics for Test Data Selection ......................... 97

4.7. Example of Rule-Base Pruning ...................................... 102

4.8. Evaluation of Random Test Selection .............................. 105

4.9. Discussion of TRUBAC as a V&V Method ......................... 108

5. Related Work ............................................................. 110

5.1. Verification Methods .................................................. 111

5.2. Validation and Dynamic Analysis Methods ....................... 112

5.3. Graph or Network Based Methods ................................... 114

5.3.1. CASNET .......................................................... 114

5.3.2. KB-REDUCER .................................................... 115

5.3.3. COVER .......................................................... 116

5.3.4. Pr/T Nets ......................................................... 117

5.3.5. Hypergraph representation ....................................... 118

5.3.6. Path Hunter/Path Tracer ......................................... 119

5.3.7. Logical Path Graph Model ...................................... 121

5.4. Summary ............................................................... 123

6. Conclusions, Analysis, and Future Work ............................ 124

6.1. Conclusion ............................................................ 124

6.2. Analysis ............................................................... 125

6.2.1. Graph Size ....................................................... 125

6.2.2. Graph Construction ............................................... 128

6.2.3. Static Analysis ................................................... 128

6.2.4. Dynamic Analysis ................................................. 128

Running test cases ...................................................... 129

Evaluating the rule-base coverage measures ......................... 129

6.3. Future Work .......................................................... 132

6.3.1. Extension – Performance Prediction ............................ 132
3.3.2. Extension - Handling Certainty Factors ........................................... 133
Rule-Base Coverage Measures with Certainty Factors ................. 134
Data Selection Heuristics in Systems With Certainty Factors .... 136
Use of Semantic Information ............................................................... 137
3.3.3. Additional Extensions ................................................................. 140

Appendix A. Format of EXPERT Rule-Bases ............................. 144
Appendix B. Simple EXPERT Rule-Base .............................................. 145
Appendix C. TRUBAC Version of Simple Rule-Base ...................... 147
Appendix D. Sample TRUBAC Session ............................................... 149
Appendix E. Test Data Used In TRUBAC Session ......................... 153
Appendix F. Prolog-Based Language .................................................. 155
References ............................................................................................. 157
Vita ....................................................................................................... 163
List of Tables

2.1. MYCIN Test Results ......................................................... 22
3.1. Overlap of sub-classes for class pairs ................................. 54
4.1. Size of EXPERT rule-bases and associated graphs .......... 86
4.2. Size of revised EXPERT rule-bases and associated graphs. 89
4.3. Relationship of classes to findings and rules in Tapes rule-base. 95
4.4. Unused sub-class to class relationships. .............................. 98
4.5. Use of findings and sub-classes to conclude classes. ........... 101
4.6. Reduction in size of RHEUM rule-base. ............................. 103
4.7. Number of test cases which conclude each class. .............. 106
4.8. Coverage results after random test case selection. ............. 107
6.1. Possible CF range coverage for classes ........................... 138
6.2. Possible CF range for findings ......................................... 139
6.3. Revised CF range coverage for classes ......................... 139
List of Figures

1.1. Conventional Rule-Base Evaluation .................................................. 4
1.2. Rule-Base Evaluation with Coverage Analysis .................................. 4
1.3. Evaluation with Coverage Analysis and Data Re-sampling ............... 5
2.1. Simple program, showing basic blocks ........................................... 25
2.2. CFG of simple program ............................................................... 26
3.1. Overview of approach ................................................................. 31
3.2. DAG representation of a simple rule ........................................... 32
3.3. DAG representation of two related rules ....................................... 32
3.4. DAG with NOFM nodes ............................................................... 35
3.5. DAG of simple rule-base ............................................................ 37
3.6. Lattice of coverage measures ...................................................... 38
3.7. Multiple paths from a finding to a class ....................................... 38
3.8. Multiple paths to a sub-class ...................................................... 39
3.9. Pseudo code for Each-class ....................................................... 44
3.10. Pseudo code for Each-hypoth .................................................... 45
3.11. Pseudo code for Each-class-every-finding ................................. 45
3.12. Pseudo code for Each-class-every-sub ....................................... 45
3.13. Pseudo code for All-edges ........................................................ 46
3.14. Model of classification system ................................................... 50
3.15. Graph of simple conjunctive rule .............................................. 57
3.16. Graph of simple disjunctive rule ............................................... 58
3.17. Graph of simple rule-base ......................................................... 58
3.18. Graph of rule-base with OR ..................................................... 59
3.19. Graph of rule-base with OR and SUB ....................................... 59
Chapter 1
Introduction and Overview

1.1 Introduction

This thesis proposes a number of ways of overcoming some of the limitations in current methods of rule-based expert systems evaluation. Usual approaches to verification and validation (V&V) of rule-based systems typically employ a static structural analysis method to detect internal inconsistencies (verification), followed by a dynamic, functional validation in which system behavior is compared with expected, or specified, behavior. The weakness of a strictly functional approach to the validation phase is that the test data available may not adequately cover the rule-base, and, at best, limited information about the coverage will be obtained. System performance statistics are usually presented as if they apply to the entire rule-base, rather than just to the tested sections, which can lead to false estimates of system performance.

In this thesis we present a rule-base coverage analysis method which enhances the functional evaluation of rule-bases by providing information that can lead to identification of both incompleteness in the test data and potential errors in the rule-base. The approach described in this thesis carries out a structural analysis of the rule-base using five rule-base coverage measures which identify sections not exercised by the test data. By using the approach described here, it is possible to improve completeness of the test suite and increase coverage of the rule-base, thereby increasing the kinds of cases on which the rule-base has been tested. The coverage information can be used to both assess the testing which has taken place and guide further testing.

The main claims of the thesis are:

1. By adding coverage analysis of the rule-base to the evaluation process, together
with meta-knowledge about some of the characteristics of the cases to be covered, we can obtain information about both the rule-base under review and the test set being used for evaluation. This information can then be used to enhance further testing. For example, if we have meta-knowledge about the findings and sub-classes involved with each test case then we can use that information to select cases which will cover specific sections of the rule-base.

2. Because lack of coverage of the rule-base during testing can result from either incompleteness of the test data or errors in the rule-base, coverage information, coupled with heuristics for data selection, can be used to guide construction of a more complete test suite.

3. Coverage analysis can be used to guide rule-base modifications. It identifies untested sections of the rule-base which can then be handled in several ways depending on assumptions about the test data. If the test suite is complete, then untested sections can be pruned. If the test suite is incomplete then additional test data or direct review of portions of the rule-base may be necessary.

4. Information that is used to determine coverage can also identify dependent classes within the rule-base, classes which can be inferred based on a high percentage of shared sub-classes.

The thesis describes:

1. a hierarchy of rule-base coverage measures as well as a rule-base representation which facilitates application of the coverage measures.

2. a set of heuristics for resampling the population of available test cases, based on coverage information.

3. strategies for rule-base pruning and identification of class-dependencies, based on the coverage information.

4. a rule-base complexity metric based on complexity metrics used for procedural programs.
5. discussion of extensions of this work to rule-bases with dynamic computation of certainty factors.

The utility of the above is illustrated by examples based on rule-bases which were prototypes for the AI/RHEUM system. In particular, we demonstrate the information we can get about both the rule-base and the test set based on application of the coverage measures. By applying the data selection heuristics we then build a test set which provides the same coverage as the complete available pool of test cases but requires running only 20% of the available cases. We also experiment with rule-base pruning and identification of class dependencies, and demonstrate that the coverage analysis does successfully identify class dependencies and provide information which is then used to guide rule-base pruning, leading to a smaller rule-base which is equivalent to the original in relation to the test data in terms of performance.

This thesis presents a number of ways in which coverage analysis, obtained during execution of test cases over a rule-base, can be used to highlight problems in both the test suite and the rule-base, thereby pointing to areas in which we cannot guarantee or predict the system's performance. In general, as shown in Figure 1.1, we can take a set of test cases for which we know the results, run them through the rule-base, and evaluate performance of the system based on a comparison of actual results to expected results. However, in addition to comparing actual and expected results, we must also consider completeness of the test set and coverage of the rule-base by the test data, as indicated in Figure 1.2. Performance of the system, as indicated by the comparison of actual and expected results, can only be expected in the tested sections, while performance in the untested sections can not be predicted. Finally, if the test set is made up of cases which have been selected from a larger population, we can use the coverage information to direct re-sampling of the population in order to choose additional cases for the test suite, as shown in Figure 1.3.

By completeness of the test set we are referring to the degree to which the data represents all types of cases which could be presented to the system under intended conditions of use. By coverage of the rule-base we are referring to use of the inference
Figure 1.1: Conventional Rule-Base Evaluation.

Figure 1.2: Rule-Base Evaluation with Coverage Analysis.
relations during the process of evaluating the test data. In the trivial case, if the rule-base were correct and the test suite were complete, then the test data would completely cover the rule-base, all actual results would agree with the expected results, and we could predict completely correct performance of the rule-base in actual use.

In the more usual situation we may have errors and incompleteness in the rule-base, as well as inadequacies in the test data. If we only judge the system based on a comparison of actual and expected results, we could have a situation in which a rule-base performs well on the test data, but actually contains errors which are not identified due to incompleteness of the test data. This could lead to a false prediction of correct performance on all cases, when in fact we cannot make any accurate prediction about performance of the rule-base in those areas for which there is an absence of test data.

Analysis of the coverage which results from execution of the test data allows for several types of rule-base analyses and refinements. Lack of coverage can result from either an incompleteness of the test data or errors in the rule-base. Meta-knowledge
about the problem domain and the complete population of available cases can be used to determine if the lack of coverage indicates that the test suite must be supplemented with particular types of test cases, the rule-base can be pruned, or the rule-base needs to be modified. The ultimate goal of using coverage measures and heuristics for test case selection is to improve the completeness of the test suite and increase coverage of the rule-base, thereby increasing the kinds of cases for which the rule-base has been executed during testing.

In this thesis we discuss the ways in which analysis of rule-base coverage can be used during the evaluation of a rule-based system to identify problems with the test data or with the rule-base. The thesis presents both the coverage analysis methods and a series of heuristics for test data selection, assuming a population of potential test cases is available. It is possible that, when testing classification systems, a large population of cases is available. However, many of these cases may represent situations which are easy to classify. Furthermore, running all available cases may be extremely time consuming for a large classification system. A random selection of test cases may give statistical confirmation that the system works properly for the tested situations, but may not cover all types of situations. The heuristics presented provide a mechanism, used in conjunction with coverage analysis, for selecting additional test cases from the available population in order to obtain broad coverage of the rule-base during the testing process. We also consider the application of the information obtained from rule-base coverage assessment to identification of class dependencies, rule-base pruning, and guided selection of additional test-cases. In addition, we introduce a metric for measuring the complexity of a rule-base in terms of basis execution paths. Throughout we use examples based on early versions of the AI/RHEUM [Kin85] rule-base for diagnosis of rheumatological diseases, along with coverage information provided by TRUBAC (Testing with RUEe-BASE Coverage), a tool which implements the coverage analysis method.

Evaluation of a knowledge-based system is a multi-faceted problem, with numerous approaches and techniques. The results generated by the system must be evaluated, along with the usability of the system, its features, how easily it can be enhanced.
and whether or not it has a positive impact on the people who are using the system in-place of an approach which is not computer based. The system's performance must also be evaluated in light of its intended use [Jac90]. If the expert system is meant to function as an intelligent assistant then it must satisfy the criterion of being a useful adjunct to the human problem solver. However, if the system is expected to carry out the reasoning of a human expert then a more rigorous evaluation of the system would be needed.

Over the last 20 years, during which time there has been considerable development and use of knowledge-based systems for medical decision support, there has been a heavy emphasis on functional analysis, addressing two primary questions: does the system give the results we expect on test cases, and does the system improve the effectiveness of those who use it? In the sub-area of rule-based systems, functional analysis has typically been preceded by a static structural analysis of the rule-base in order to eliminate certain kinds of structural flaws (see Chapter 2). The emphasis on functional analysis can lead to seemingly strong statistical statements about the correctness of a system, demonstrating that it gives the correct result, or the same result as a human expert, in a high percentage of test cases. However, functional testing does not guarantee that all parts of the system are actually tested. If some section of the rule-base is not exercised during the functional test then we do not have any information about that section of the system and whether it is correct or contains errors. Furthermore, many performance problems for rule bases are the result of unforeseen interactions between rules [OO93]. A test suite of known cases may never trigger these interactions, though it is important to identify them in order to correct them before a system is put into actual use.

The method presented in this dissertation enhances functional analysis of rule-based classification systems with a rule-base coverage assessment. While the approach described here may be extensible to other categories of rule-based systems, this remains the subject of future work. Classification systems are used in many areas, with a wealth of applications in medical applications such as diagnosis, antibiotic prescription, and treatment or therapy suggestions.

The view taken in this dissertation is that functional testing is not sufficient for
adequate evaluation of a rule-based system. Current dynamic testing methods may neither sufficiently test the rule-base, nor give information indicating the ways in which the testing has been inadequate. Reliance on black box testing as the only form of dynamic or functional evaluation can lead to a “brittle” system [Bun87], one which can only handle problems on which it has been tested but fails on problems not provided as test data, or fails in one area after a small change has been made in a different area [FdK93]. Some form of white box [Mye79] evaluation of the system, based on the system’s structure and logic, is necessary in order to attain higher confidence in the conclusions reached by the knowledge based system. Therefore, we propose a form of rule-base validation with extra requirements, in the form of rule-base coverage measures, which are evaluated during the functional analysis process.

The underlying premise of this work is that an ideal testing method would be one that guaranteed that all possible reasoning paths through a rule-base had been exercised. However, as with procedural software, this is often an unreasonable and/or unattainable goal, possibly due to a lack of test data, to unexecutable program paths, or to the size of the rule-base. Furthermore, even if each possible path is exercised, we cannot realistically do so with each distinct set of test values that could cause its traversal.

Our rule-base coverage approach to system analysis allows clear identification of sections of the rule-base that have not been exercised during functional testing, indicating either weaknesses in the test set or sections of the rule-base that may not be necessary. An incomplete test set can then be supplemented with additional cases chosen from the available population, guided by a series of heuristics and the coverage analysis. Alternatively, if there is no test data which covers certain parts of the system, it is possible that those sections should not remain a part of the system at all. The coverage information also provides information about the relationship between findings and hypotheses in a system and semantic information about the problem domain which can be used to guide rule-base pruning. These methods have been implemented in the TRUBAC tool, which we also describe.
The issue of rule-base coverage during testing is not one that has been widely discussed in the verification, validation and testing (VV&T) community. However it has been touched on, particularly by researchers whose work has involved a graph-based or data flow approach to rule-base analysis (for example, see discussion of [Kip92] in Chapter 5). We attempt in this work to demonstrate that a testing approach based on rule-based coverage measures is not only feasible, but also can provide greater confidence in the performance of a rule-base than can more common approaches to dynamic testing. We are not suggesting that every rule-base must satisfy the strongest coverage measure during testing. However, as is the case with other kinds of software, part of the expert system development process involves an assessment of the "cost" of a system failure and an allocation of resources to system testing. We are proposing that the rule-base coverage measures introduced in this dissertation be used as a component of the testing strategy employed, particularly in situations in which, because of the nature of the system and the impact of system failures, extensive testing is desired.

As expert systems get larger and multiple experts are used as sources for the knowledge, it becomes increasingly difficult for system developers to do an adequate job of testing without some form of automated oversight. Typically testing of a rule-based system is broken down into two phases, verification (a static process) and validation (a dynamic process). The approach described in this dissertation provides information which significantly enhances the validation process. Usually validation of an expert system is largely a functional analysis process, performed by running some number of test cases which have known results. The system's answers and line of reasoning are compared to the known results and the expert's line of reasoning. If these agree for a significant percentage of the test cases then the system is considered to be acceptable. In the case of rule-based systems, this approach to validation seems to give a measure of how accurately the rule-base performs on the test cases, but gives no information about the extent to which the rule-base is actually exercised by the test cases. It is possible that there may be entire sections of the rule-base that were not used at all in the evaluation of the test cases, in which case the accuracy figure obtained applies to only a segment of the rule-base.
The primary advantages of using a rule-base coverage assessment with functional testing are:

- coverage information, coupled with heuristics for data selection, can guide construction of a more complete test suite.

- coverage analysis identifies unexecuted sections of the rule-base which may contain errors, and warrants further testing or direct examination by the expert.

- unexecuted sections of a rule-base can indicate inadequacies or bias in the test data

- information about additional relationships within the rule-base, such as overlap among the groups of sub-classes which lead to pairs of classes, provide information about class dependencies which is useful for system retest after system refinement.

- unexecuted sections of a rule-base can indicate sections of the rule-base that may be candidates for pruning

- the rule-base representation proposed for validation can also be used in a very straightforward fashion to carry out a number of verification tasks as well.

- the approach we describe can be extended for rule-based expert systems which incorporate dynamic computation of certainty factors, which are not handled by current validation methods.

In addition to presenting the rule-base coverage testing approach, we also consider why approaches that have been used to analyze and test procedural software are not suitable for rule-based systems. As a result of this discussion we present a metric that allows for a quantitative assessment of the complexity of a rule-base and its testing.

### 1.2 Terminology

There are a number of factors in the development and use of expert systems which increase the importance of verification and validation. As expert systems become more
widely used they are increasingly being used in areas, such as medical diagnosis, battle management [Cha87], nuclear power plant operation, and other applications in which an error in the system could have potentially disastrous consequences, if not overruled by the user.

The research area of knowledge base verification, validation and testing (VV&T) is replete with definitions of these terms\textsuperscript{1}. In this dissertation we use the following definitions [ABC82]:

- **Verification** – A static analysis process. Detect internal inconsistencies in a rule-base, such as redundancy, conflict, or cycles (when not allowed). Check that the system is logically sound and complete.

- **Validation** – A dynamic analysis process. Determine that the system is behaving in accordance with the specification. We take this to mean “gives the right answer”\textsuperscript{2}. Alternatively, we want to check that the conclusion of the system “resembles” that of the human expert who provided knowledge for the system [Pre89].

- **Testing** – examine the program’s behavior by executing it on sample data sets. In the expert systems field, unlike in software testing in general, *validation* is usually taken to incorporate *testing*.

Verification can be carried out statically, by studying the rule-base without running test cases. However, validation must be a dynamic process wherein the rule-base is applied to test cases and the resulting conclusions are evaluated. As will be seen in Chapter 5, expert system validation has historically been treated as a black box [Mye79], functional activity in which the test data selection is based on the system’s input and output specifications only, without taking into consideration the internal structure of the system. Typically a standard for satisfactory validation might be to run a test suite which causes all classes of the system to be concluded by at least one test case, or it

\textsuperscript{1}In some cases the confusion caused by repeated, different use of these terms is avoided by the introduction of new terms. For example, some researchers [LMP90] make a distinction between structure and function, referring to “structural verification” and “functional verification”, in place of verification and validation respectively.
might be to attain a certain percentage of "correct" answers, when compared with one or more human experts.

The work presented in this dissertation attempts to address several issues in the development and testing of rule-based expert systems\textsuperscript{2}. Other automated VV&T methods have been largely dependent on their knowledge base development environment. In addition, many of the automated methods previously developed do only verification or validation, not both. Many of the existing systems also work on knowledge base models which are simpler than the systems that are in use today [LMP90]. In particular, existing methods do not deal with control structure and uncertainty management.

This situation has led to calls for better automated tools [Naz89, And92, Rus88] and to an indictment of some of the existing tools [And93]. Andert [And92] proposes that automated tools be implemented so that VV&T can be performed independently of the particular development environment used to construct the knowledge base. While not presenting a specific methodology, Andert points out the need to do both static and dynamic analysis of the knowledge base, and suggests the combination of several existing VV&T techniques into one tool suite. These techniques could include pairwise rule comparisons and decision-table techniques, graph detection algorithms on logic trees, completeness checking, and branch testing.

Work on the Path Hunter/Path Tracer tools [GPC+93, PGCR93], an outgrowth of the work on the COVER tool [GPSS93], presents a somewhat more developed view. The Path Hunter/Path Tracer tools use a combination of static and dynamic testing, where the dynamic testing is a combination of functional testing and structural testing. The authors suggest that an initial test set be constructed based on functional testing criteria and run on the expert system. Then a determination should be made of the extent to which the test set exercised all structural components of the system. Then further test cases should be generated and run to cover any deficiencies detected. A weakness of this approach is that the selection of test data is driven both by structural criteria and by the problem specification which guided system development. This can

\textsuperscript{2}The tool described in this dissertation could also be used to test a rule-base developed via a learning system [WK91], not just rule-bases created by knowledge engineers.
still leave untested significant aspects of the system’s reasoning capability, particularly those that were developed erroneously, perhaps contradicting the problem specification.

Andert raises further criticisms of knowledge base developers and the VV&T community. He claims [And93] that developers are unaware of the need for verification and validation, and they are hampered further by what he sees as an unavailability of tools for them to use. His view is that the focus of research has been on static anomaly detection, which is not adequate for validating real-world knowledge bases, and many of the existing tools are not easily available outside of the research setting. He argues that more dynamic analysis techniques should be developed, which will lead to better coverage of knowledge bases and a higher degree of confidence in the reliability of knowledge based systems. The work presented in this thesis addresses some of these issues by defining a sequence of coverage measures by which the completeness of rule-base validation can be assessed.

1.3 Outline of the Dissertation

In Chapter 2 we present foundation material from expert systems and program testing which motivated the testing approach described in subsequent chapters. This includes a discussion of issues that arise from consideration of expert systems in the context of some traditional software analysis methods. We discuss the control flow analysis of procedural programs and some complexity metrics.

In Chapter 3 we present our approach to rule-based system testing, based on five rule-base coverage measures. We then present the heuristics for test case selection, based on information obtained from the coverage analysis and meta-knowledge about the population of available cases. We also discuss the use of additional rule-base information to identify class dependencies and to provide information that can direct rule-base pruning. Building on the foundational material in Chapter 2, we show that control flow analysis breaks down as a method for analyzing rule-based systems. We introduce a complexity metric for rule-bases which is motivated by the control flow metrics. We also compare a set of data flow testing criteria for procedural programs with the criteria
we have proposed for rule-based systems.

In Chapter 4 we present a brief discussion of the TRUBAC tool, a C language implementation of our testing method. We also present the application of TRUBAC to rule bases from several different development platforms. We then present examples of the use of information from TRUBAC to guide rule-base pruning and to guide test case selection. We also use the coverage measures to evaluate the quality of testing when test cases are randomly selected from the available test suite. Finally we discuss TRUBAC, reflecting the rule-base coverage testing approach, as a V&V tool.

Chapter 5 contains a review of related work in the VV&T area. Both static and dynamic methods are discussed, with particular focus on methods which use a graph structure and/or an execution path model. Finally we discuss the ways in which our approach differs from other methods and its advantages.

Finally, in Chapter 6 we present conclusions and discuss the complexity of our approach, looking at the construction of the rule-base representation as well as the static and dynamic analysis steps. We also discuss future work in this area including extensions of our approach to enable more accurate performance prediction, to evaluate systems which have dynamic computation of certainty factors, and to aid in the incremental development of rule-bases.
Chapter 2
Foundations

The rule-base testing method described in this dissertation was motivated by control flow analysis and data flow testing of procedural programs. In this chapter we first discuss the sources of errors found in expert systems, as well as various aspects of expert system evaluation. We then briefly discuss functional evaluation methods commonly used to validate rule-based systems, focusing on the MYCIN system [BS85] as an example of a typical approach to the dynamic testing or analysis of an expert system. We follow that with discussion of a number of aspects of traditional software analysis. These include control flow analysis of procedural programs and an associated complexity metric, as well as data flow testing of procedural programs.

2.1 Guidelines for Expert System Analysis

There exist a number of models for measuring knowledge-base performance, based on the error rate of a system when used to evaluate sample data [Ind91]. We could compute an error rate based on the number of correct and incorrect responses made by the system. For classification systems, this type of error rate is based on the actual class versus the class selected by the system. While this is just one of many models for measuring knowledge-base performance, it is clear that the performance measure produced applies only to that part of the system that is actually executed by the test data.

Before discussing the factors involved in verification and validation of expert systems, it is important to mention how knowledge base errors can occur [Naz89, DW93, GPSS93]. Sources of errors in the knowledge base include ([GPSS93]): an expert’s lack of expertise or understanding of the problem domain; specific errors in the heuristics
used by the expert(s); and knowledge representation errors made by the knowledge base
designer. More specifically, errors can arise from:

- Typographical mistakes, which may be hard to identify if names are not chosen
  well in the rule-base.
- Inadvertent errors, in which part of a rule is left out or implication is reversed
  (the development platform used may help prevent this sort of error).
- Errors due to not thinking through the application problem domain, or due to
  limited knowledge.
- Errors due to changes made in an attempt to enable the system to have wider
  application coverage.
- Errors introduced during system maintenance, when rules are added and changed.
- Inability to handle cases that are more complex than those on which development
  of the system was based.
- Errors due to order-dependent rules, with problems caused by unforeseen rule
  interactions.
- Errors caused by incorrectly assuming that an input set is unrealistic, when in
  fact it is possible.
- Errors caused by under-qualified rules, i.e. rules that should cover mutually ex-
  clusive conditions but instead cover overlapping conditions.

There are a number of factors that make the verification and validation of expert sys-
tems difficult, particularly when compared with the testing of conventional software
([SG88]). First, expert systems are usually not as modular as are procedural pro-
grams, so the testing task cannot be broken down into smaller sub-tasks1. Second,
construction of test cases can be very difficult since it is hard to determine possible

---

1For example, when using ASSET [FW85] to test a Pascal program involving many procedures,
each procedure is tested separately, starting with procedures which make no calls, and continuing with
procedures for which all called procedures have already been tested.
execution sequences by studying the static rule-base. Historically ([SG88]) this has also made it difficult to apply coverage measures. Third, an expert system can generate multiple, correct answers for a given set of input, which can complicate the task of determining if the system functions correctly. Fourth, the relative ease or difficulty of verifying a knowledge base is connected to the problem domain for which it was developed ([Pre89]). In a wide domain, such as medicine, verification will be more difficult since there are many facts that can affect each outcome of the system. However, it is in just such situations that verification is even more critical than in narrow domains. It is much more difficult to check such a system by hand, and testing simply by enumeration of test cases is difficult due to the combinatorial explosion of both possible inputs and possible inference chains through the system.

There are three issues that must be considered when attempting expert system testing or analysis. First, what sort of analysis are we trying to do, what are we trying to learn about the expert system? Second, what are our criteria for testing the system, what are the goals of the testing process? Third, how much test data do we have, and where does it come from?

2.1.1 Kinds of analysis

There are a number of kinds of analysis that are carried out on conventional software. These include criticality and sensitivity analysis, as well as efficiency analysis and consideration of the maintainability of the software. While these can be applied to expert systems as well, there are additional high-level analyses that we should be prepared to carry out. These include ([GK87]):

- *Interaction analysis* – are the interacting components of the expert system able to produce the desired results.

- *Truth analysis* – evaluate the truth of the facts and rules stored in the knowledge base. This may require review of the knowledge base by an expert.

- *Uncertainty analysis* – do we get correct results from the assignment of certainty factors and the way in which they are combined. Is the method by which certainty
factors (CFs) are used in the system sufficiently understandable and related to
the way experts use it in the problem domain.

Interaction analysis usually involves looking for two separate types of problems:
completeness and consistency problems ([Nas88]). In testing for system completeness,
we look for facts which are never used, unreachable conclusions, dangling conditions,
useless conclusions, missing rules, and illegal attribute values.

In testing for system consistency, we look for the absence of ([Naz89] and others):

• redundancy in pairs of rules or inference chains
  – identical rules
  – antecedents subsumed with identical consequents
  – antecedents identical with subsumed consequents
  – antecedents and consequents subsumed

• conflict within a single rule, in pairs of rule or inference chains
  – consequent of a rule conflicts with the antecedent of the same rule (self-contradictory rule)
  – final consequent of inference chain conflicts with conjunction of antecedents along the chain (self-contradictory inference chain)
  – rule pair deduces contradictory facts from same antecedent
  – two inference chains deduce contradictory facts from same (compound) antecedent information

• circular rules or inference chains

• unnecessary if conditions

Redundancy may not be a problem in expert systems without uncertainty measures, as long as any changes made to one of a pair of redundant rules are also made to the other. However, in a system with uncertainty measures, redundancy can lead to the false increase of certainty. In addition, we want to find situations in which different rules can lead to the same conclusion, but might do so with different certainty ([LMP90]). Furthermore, if the certainty factors are built into the rules (so that there is uncertainty, but it is not dynamically computed) then rules which are otherwise the same but have different confidence factors will not be considered redundant.
• illegal attribute values

• unsatisfiable conditions – a rule condition tests a value against a parameter, but there is no way that the parameter can hold that value ([Pre89])

2.1.2 Testing Criteria

There are two basic criteria to consider when testing expert systems. First, we need to know what data (or kind of data) the system should handle, and make sure that it does so correctly. Second, we need to identify what data the system can handle, which may be a superset of the data it should handle, and make sure that such data represents legitimate situations in the problem domain and that the expert system does something reasonable when presented with it. Given these two rather broad guidelines, there are many approaches to determining a particular set of test cases for a particular expert system.

In [GK87] seven criteria are presented to guide the selection of test cases. We present here those that are relevant for this research, with brief discussion of how they relate to our testing approach.

1. Test every requirement. This is equivalent in our method to presenting test cases, the union of which includes all facts needed by rules in the rule base.

2. Test every possible decision, using equivalence partitioning to keep down the number of test cases; invoke every fact, object, rule; cause the testing of every outcome of every rule.

3. Test outputs and process by which the system produced the outputs.

4. Test a judicious selection of condition (finding) combinations, especially those believed to be particularly significant in system operation or that have the best chance of revealing errors. The weakness of this criterion is that it leaves the test case selection process entirely up to the subjective judgment of the tester, providing no real constraints or guidance for it. One goal of our testing approach is to make the process of test case selection more objective, where test cases are
chosen based on their objective ability to contribute to some aspect of rule-base coverage.

We must keep in mind ([GK87]) that while the determination of whether or not the expert system gives a correct result for a particular set of inputs may be a subjective decision, the selection of the input set need not be and should not be completely subjective.

2.1.3 Test Data

In any discussion of testing expert systems we must address the issue of what data should be used to test or analyze the expert system. There are historically a number of problems that arise when developers get to the stage of testing expert systems ([OBS87]). Often there may be only a small sample of test data, or no data at all. It is also possible that the majority of available cases were used to guide system development, and are therefore not useful for system testing. Even if there are test cases available, they may not be representative of the full range of allowable input, and therefore may not lead to reasonable or sufficient coverage of the rule-base.

If there is sufficient test data available, one would like to create a library of test cases which can be used repeatedly as the rule base is modified ([GS88]). This library should include both obvious and subtle cases, cases which involve boundary conditions, meaningless combinations of valid and invalid data, and obvious error conditions on which the system should fail. However, it is important to note that this variety of test cases still does not guarantee coverage of the rule-base. It is usually possible to generate all of these various types of test cases and still not test some sections of the rule-base.

In the past, one solution to a shortage of test data was to ask the experts to provide "synthesized" test cases. However, in our work we take a different approach to this problem. Rather than ask the expert to make up test data, we propose using the rule-base representation and the coverage measures as the foundation for generation of test cases which would cover specific sections of the rule-base. These test cases, with their diagnoses or conclusions, can be presented to an expert to see if they "make sense" in
the context of the problem domain. Since the expert may have a particular bias as to what are interesting cases within the problem domain, letting the coverage measures drive the data synthesization process will broaden the testing coverage.

2.2 Traditional Evaluation Methods

The traditional approach to dynamic testing is well stated in the *Suggestions for the Design of a Model* in the EXPERT user's guide [WKKU87].

Test the model on a database of cases. Consider those cases with inaccurate conclusions, and determine how the model may be changed to correct the errors. Revise the model and examine the conclusions for this case and the net-effect of these revisions on the other saved cases.

A recent [O'L91] survey of expert systems developers showed that using known test cases is the most common validation method. Among the 34 developers who responded to the survey, 68% of their validation methods involved either known test cases or parallel use of the expert system with the system it was to replace. Another 30% of validation methods involved direct examination of the knowledge-base by either the expert who had provided the knowledge or by the developer. However, for any sizeable rule-base it is not reasonable to assume that direct examination of the rules will lead to identification of unforeseen rule interactions or other errors that involve multiple rules. In addition, examination of the rule-base will not increase confidence in the correctness of untested sections of the rule-base unless it focuses on those sections which have been identified in some way.

Traditional approaches to expert systems testing are exemplified by the steps taken to test MYCIN. During development of the system, MYCIN was tested by running several hundred cases of retrospective patient histories, prospective patient cases, and meningitis cases found in the medical literature [BE85]. There is also fairly extensive discussion (Chapters 30 and 31 of [BS85]) of the way in which MYCIN was evaluated in order to determine if it had attained an acceptable level of functional expertise in the problem domain. In the first evaluation attempt, experts were asked to assess whether
Ratings by 8 evaluators of 10 cases per expert

<table>
<thead>
<tr>
<th>Expert</th>
<th># acceptable</th>
<th>% acceptable</th>
</tr>
</thead>
<tbody>
<tr>
<td>MYCIN</td>
<td>52</td>
<td>65</td>
</tr>
<tr>
<td>Faculty-1</td>
<td>50</td>
<td>62.5</td>
</tr>
<tr>
<td>Faculty-2</td>
<td>48</td>
<td>60</td>
</tr>
<tr>
<td>Infectious dis. fellow</td>
<td>48</td>
<td>60</td>
</tr>
<tr>
<td>Faculty-3</td>
<td>46</td>
<td>57.5</td>
</tr>
<tr>
<td>Actual therapy</td>
<td>46</td>
<td>57.5</td>
</tr>
<tr>
<td>Faculty-4</td>
<td>44</td>
<td>55</td>
</tr>
<tr>
<td>Resident</td>
<td>36</td>
<td>45</td>
</tr>
<tr>
<td>Faculty-5</td>
<td>34</td>
<td>42.5</td>
</tr>
<tr>
<td>Student</td>
<td>24</td>
<td>30</td>
</tr>
</tbody>
</table>

Table 2.1: MYCIN Test Results

MYCIN had given the correct answer for a number of test cases. In this evaluation attempt the experts knew they were evaluating a computer based expert system and MYCIN was rated as 75% accurate. In a subsequent evaluation attempt of the same nature MYCIN was again considered to be 75% accurate, far from the 90% accuracy rate stated as a goal by the developers.

Finally an evaluation process was carried out in which the experts were prevented from letting a bias against expert systems affect their evaluation. Summaries of 10 patient cases were developed. The summaries were run through MYCIN, and also given to seven people with varying degrees of expertise. Each of these “experts” and MYCIN prescribed a therapy regimen for each test case. The collection of prescriptions was then given to eight different experts, along with the case summaries. These experts then made their own prescription for each of the cases and judged the 10 prescriptions they were shown with the summaries. In this study it was found that MYCIN performed better than any of the experts against whom its prescriptions were compared (see Table 2.1. from [Ye85]). However, it still attained only a 65% approval rating when its performance was judged by experts in the field. While this approach carried out a thorough comparison of MYCIN’s functional capability and accuracy to that of human experts, it gives no guarantee of how well the chosen test cases actually covered MYCIN’s rule-base structurally when they were evaluated by MYCIN. We must
view the 65% accuracy rate with care, recognizing that what it really indicates is that
MYCIN is 65% accurate for the test suite within the part of the knowledge base that
was actually executed by the test cases. Without some proof of adequate rule-base
coverage it cannot be concluded that MYCIN is 65% accurate overall. And while it is
ture that MYCIN performed better than did all the human experts, that fact should
not dispel concern about the unused sections of the rule-base for two reasons. First,
consider the situation in which the unused sections of the rule-base have never been
executed because they, in fact, do contain some errors and can never be used by well-
formed test cases. If we do not identify those sections and correct or eliminate them
then there is some risk that incorrectly entered data will cause those sections of the
system to execute, giving unpredictable and potentially incorrect results. Second, con-
sider the situation in which the unused sections are no longer a necessary part of the
rule-base and represent unreachable conclusions. While it may do no harm to retain
those sections, in the sense that they will not lead to any incorrect results, it reduces
the efficiency of the system to have extra rules which are never used.

Evaluating functional testing with rule-base coverage measures can help in the iden-
tification and correction of these kinds of situations. Furthermore, monitoring the
rule-base coverage achieved by the test cases will clarify exactly which sections of the
rule-base perform at the indicated accuracy. The accuracy figure obtained by compar-
ing the system’s results on a set of known cases with an expert’s results on those same
cases is reflective of the kind of cases included in the test set. If the test data has been
biased by using many “standard” cases, the accuracy of the system would seem very
high. In this situation, evaluating rule-base coverage would immediately show that the
sections of the rule-base that evaluated more difficult cases had not been exercised by
the test data.

2.3 Control Flow Analysis

There are several ways in which rule-based systems differ from programs written in a
procedural programming language. Many rule-base languages do not have any control
structures for looping, and there are also no structures equivalent to procedures or
functions in programs. Therefore we avoid the problems of infinite program paths, due to looping, and parameter passing, due to procedure and function calls. These two simplifications should make it easier to analyze rule-based systems. However, there is a significant complication with rule-based systems. In a procedural program we can clearly describe the possible control flow through the program (usually using a Control Flow Graph (CFG)) because the sequential nature of computations in the program is completely determined by the order of statements in the code. By comparison, in a rule-based system the execution order of the rules is not determined by the structure of the rule-base, but rather by the data present at the time of system execution. This makes it much more difficult to analyze a rule-based system. It is still possible though to use control flow analysis of procedural programs, and associated complexity metrics, to develop some useful analysis tools for rule-based systems.

Control flow analysis of a program is a method for generating information about the run-time behavior of the program without actually executing the program [MJ81]. This information can be used subsequently to guide program optimization, to create invariant assertions which can be used in program verification, to generate program documentation, or to automate some parts of program debugging.

The first step of control flow analysis is the breakdown of the program into basic blocks, each of which is a sequence of consecutive instructions which the program must enter by executing the first instruction and from which the program can exit only by executing the final instruction (that is, there are no branches into or out from the middle of a basic block) [Ken81]. Figure 2.1 shows a simple program (adapted from [Ken81]), separated into basic blocks. Generally basic block construction and control flow analysis are carried out on an intermediate code representation of the source code (generated as part of the compiling process). After the basic blocks have been constructed, the control flow graph (CFG) can be built. The control flow graph \( G = (N, E, n_0) \) is based on

- \( N \) – the set of basic blocks which are the nodes of the graph

- \( E \) – the set of edges which represent control flow between blocks, or block-to-block
rept: scanf("%f %f %f", &A, &B, &C); Block 1
if (A == 0)
    return; Block 2
Disc = B*B - 4.0*A*C; Block 3
if (Disc >= 0)
    { Block 4
        Droot = sqrt(Disc);
        R1 = (-B + Droot) / (2.0 * A);
        R2 = (-B - Droot) / (2.0 * A);
    }
else
    { Block 5
        Droot = sqrt(-Disc);
        R1 = -B / (2.0 * A);
        R2 = Droot / (2.0 * A);
    }
printf("%f \%f \%f\n", Disc, R1, R2); Block 6
goto rept;

Figure 2.1: Simple program, showing basic blocks.

transfers

• n₀—the element of N which is the program entry point.

Figure 2.2 shows the CFG of the program presented in Figure 2.1.

Once the CFG has been created, it can be used to carry out numerous analyses of
the original program. Usually these analyses are based on data flow analysis, in which
we are interested in the precise relationships between variable definitions and variable
uses that are allowed by the control flow structure of the program [Ken81]. For example,
in an optimizing compiler we would be interested in identifying and eliminating dead
code and common subexpressions [Hec77]. In order to identify dead code, we want
to look for statements that define a variable (compute a value for it) which is never
used subsequently in the program. We can accomplish this by using the definition-use
relationships to help us mark “useful” instructions, instructions that define variables
that are used subsequently in the program according to the control flow graph. That
is, a variable definition might not be used subsequently on every path through the
program, but if it is used on any path then its definition is a necessary part of the program. Any instructions left unmarked by this process can be eliminated from the program.

2.4 Control Flow Metrics

McCabe [McC76] introduced a complexity metric for procedural programs, based on the control flow graph representation. His goal was to develop a measure that would quantitatively identify the parts of a software system that would be difficult to test and maintain. His research showed that, in procedural programs, a limitation on module size (lines of code) is not sufficient to limit the complexity of the module, as it is the structure of the program which leads to the number of possible control paths, which in turn determines the complexity of testing and maintaining the program.

Because of the possibility of loops, and the resulting difficulty in counting the total number of possible paths through a program, McCabe defined a complexity metric which measures basis paths, which can be combined to form every possible control path through the program. The metric he uses is based on the cyclomatic number \( V(G) \) for a graph \( G \). This is defined in graph theory for a graph \( G \) with \( n \) vertices, \( e \) edges, and
$p$ connected components, as

$$V(G) = e - n + p$$

Based on this value, McCabe defined the **cyclomatic complexity** of a graph $G$, $CV(G)$, as being the number of linearly independent program paths, which can be computed by $V(G) + p$ or

$$CV(G) = e - n + 2p$$

McCabe applied this to program analysis by computing the cyclomatic complexity of the control flow graph for a program. It is then possible to choose $CV(G)$ linearly independent circuits from the control flow graph. McCabe argues that this set of circuits forms a basis set for all circuits in the graph, and any path through the CFG can be expressed as some combination of circuits from this set.

McCabe proposed that, during software development, the complexity of a program or a module be controlled by setting an upper bound on $CV(G)$, rather than focusing on just the number of lines of code. When the program is being tested, the cyclomatic complexity can also be used as a mechanism for judging the adequacy of the actual number of paths tested. If the number of paths tested is less than the cyclomatic complexity then it is possible that more testing has to be done. However, it would also be possible that the program, and therefore the CFG, can be simplified by taking out decision points which are unnecessary and are increasing the complexity.

### 2.5 Data Flow Testing of Procedural Programs

Data flow testing of procedural programs is based on both the control flow and data flow possible within a program. The main principle of data flow testing is: if there is a value assignment to a variable that is not used in some subsequent computation or predicate in the program during the program test, then we cannot be sure that the steps leading up to the assignment were correct. Data flow testing involves tracking input variables through a program, following them as they are modified, until they are ultimately used to produce output values. Ideally the program tester would like to provide as many different test cases as are necessary to "exercise" all paths between
variable assignments and variable uses.

Rapps and Weyuker [RW85] developed a family of data flow testing criteria, the strongest of which requires that the test set include data which causes the program to execute every path connecting a variable definition with a subsequent use of that variable in a computation or a predicate. ASSET [FW85] is a software tool for Pascal programs which incorporates these data flow testing criteria, thereby evaluating the usefulness of test data and helping the user to select test data. The user provides a module to be tested, along with test data and the specified criterion. ASSET then tells the user which paths (as required by the chosen criterion) were not traversed by the given test data. The user can then provide additional test data until either all required paths are exercised or the user can see that some path may not be executable. In addition, ASSET serves as a debugging aid by indicating what path through the program the user should study if a particular test case gives an incorrect result.

In the next chapter we introduce our approach to rule-base testing that applies the control flow and data flow concepts to the rule-base structure.
Chapter 3

Testing with Rule-Base Coverage

There are obvious analogies we can make between data flow testing and the testing of expert systems. The relationship between a variable assignment and a subsequent use of the variable is analogous to the relationship between a rule which generates a particular consequent and a subsequent rule in the reasoning chain which uses that consequent in its antecedent. We propose that the degree of coverage attained by the test data run through a rule-base be based, in part, on a determination of whether each finding and sub-class conclusion resulting from a rule firing is subsequently used in the antecedent of another rule in a reasoning chain that correctly culminates at a class of the system.

Therefore, a “data flow” testing method for expert systems must define an analog of the path testing used for procedural programs. An immediate problem is that we do not have the same notion of paths in a rule-based system [Rus88]. In this chapter, after an overview of the process of testing with rule-based coverage measures, we introduce a representation for rule-bases that will allow us to define paths in the context of a rule-based system. We then discuss the five Rule-Base Coverage Measures (RBCMs) and the process of static analysis with our representation and dynamic analysis using the rule-base coverage approach. We discuss how this approach can help us identify the types of knowledge base errors mentioned in Chapter 2. This is followed by discussion of heuristics for test data selection, based on coverage information. We then introduce additional information that is available about the system and as a result of running test cases on a rule-base which is useful for identifying class dependencies and sections of the system which can be pruned. We also introduce a metric for rule-based systems based on the control flow metric for procedural programs, explaining why the control
flow analysis methods used for procedural programs break down for rule-bases. Finally, we compare a set of data flow testing criteria for procedural programs with the criteria we have proposed for rule-based systems.

3.1 Overview of Testing with Rule-Base Coverage Measures

The first step in rule-base testing with coverage measures is to build a graph representation of the rule base. In particular, our method is based on a directed acyclic graph (DAG) representation. During construction of the DAG, redundant rules, simple contradictory rules and potential contradictions (ambiguities) can be identified. After DAG construction is complete the user can carry out static analysis (verification) of the rule-base. This will report dangling conditions, useless conclusions, and cycles in the rule-base. At this point the user could modify the rule-base to eliminate or correct any problems identified during static analysis.

The static analysis phase is followed by dynamic analysis of the rule-base using test cases. It is in this area that our approach differs from that of other expert system analysis tools. As test cases are processed, one or more of several rule-base coverage measures (RBCMs) can be reviewed in order to determine the quality of the test data supplied thus far. Additional information about the rule-base and its testing can also be used by the system tester to guide the selection of future test data. Presumably the tester would start by providing sufficient test data to satisfy the simplest functional measure (conclude each class of the system) and proceed to the more difficult structural measures (such as coverage of all execution paths). Finally, if the user is not able to provide sufficient data to attain the desired degree of rule-base coverage (according to the criterion), the user could use the graphical representation of the rule-base to generate synthesized data, which can then be reviewed by an expert to determine if the data represents a valid case. Figure 3.1 shows the general steps of the approach we are suggesting.
3.2 Rule-base Representation

In order to judge the rule-base coverage, we first must have the rule-base in a form which will allow us to identify which sections have been covered, or exercised, by the test data and which have not been. The representation we propose using in this work is based on the AND/OR graph implicit in a rule base ([Mes91]). This representation for rule-based systems is a directed acyclic graph (DAG). It contains a source node, a sink node, and nodes for each finding and each class of the system. The source node corresponds to working memory, the source of findings supplied to the system during a consultation. The sink node corresponds to success in reaching one of the classes of the system. There are also two additional types of interior nodes: sub-class nodes (SUBs) and operator nodes. Sub-class nodes represent non-class rule consequents. Operator nodes are used to represent the allowable operators AND, OR, and NOFM ¹. These operator nodes allow us to represent the fact that the conjunction and/or disjunction of multiple components of an antecedent must be true in order for the conclusion of a rule to be entered into working memory. There are edges from the source to each finding and from each class to the sink. The antecedent of each rule is represented

¹NOFM nodes represent the construction “if N of the following M things are true, then ...”, which is a feature of EXPERT [WKKU87] rule-bases. The most comprehensive testing would be achieved by creating graph nodes representing each of the $\binom{n}{m}$ combinations. However that would create a graph that was completely unwieldy and impossible to test. Furthermore, it seems that when experts provide this sort of construction in a rule, they do so in order to avoid delineating the combinations. Therefore it seems that to test the system with the NOFM nodes is in keeping with how the experts solve problems, at the risk of losing some accuracy in the test process.
Figure 3.2: DAG representation of a simple rule

Figure 3.3: DAG representation of two related rules.

by a subgraph which connects findings and sub-class nodes to operators as indicated by the antecedent. Each antecedent-consequent connection represented by a rule is also represented by an edge from the subgraph for the antecedent to the node for the consequent.

For example, the representation of the rule

\[ \text{If P1 \& P2 then R1} \]

is shown in Figure 3.2. P1 and P2 could be finding or sub-class nodes, while R1 is either a class or a sub-class node.

The complete graph of a rule-base is constructed by linking together the individual structures for successive rules. For instance, if there is also a rule in the rule-base

\[ \text{If R1 \& R2 then Q} \]

then the two rules will be represented by the subgraph shown in Figure 3.3. In this framework we can easily represent rule-bases that encode reasoning processes that do

---

2In the literature there are often references to the left hand side and right hand side of rules. However this terminology can be confusing as some rule formats use left-to-right implication (EXPERT) while others use right-to-left implication (Prolog). To avoid confusion, wherever possible we refer to the antecedent and the consequent of the rule, instead of the left hand side and the right hand side.
not rely on dynamic computation of certainty factors, including rule-bases in which the certainty factors are hard coded within the consequent definitions. In Chapter 6 we discuss the extension of our testing approach to rule-bases that involve the dynamic computation of certainty factors.

### 3.3 Rule-base coverage measures

Using the DAG structure described above, we can make an analogous definition of execution paths in the representation of a rule-based system. Each reasoning chain through the rule-base corresponds to some sub-DAG of the DAG representation of the rule-base. That is, if we were to mark all nodes and edges in the DAG that correspond to the rules fired during a particular chain of inferences, we would mark the source node, the sink node, all nodes corresponding to the findings involved, the class concluded, all nodes corresponding to antecedent components, operators, rule consequents, and all edges involved in antecedent-consequent links formed by the rule connections used in the reasoning chain. We can think of a rule firing as involving all edges and nodes in the graph that corresponds to the rule. An execution path is, therefore, the sub-DAG that corresponds to all the rules fired along a particular reasoning chain executed due to a specific set of findings.

Therefore, the idea of path coverage, from procedural program testing, is analogous to execution path coverage in the DAG representation of the rule-base. Ideally, we would like to provide sufficient test data to cause every possible execution path of the DAG to be traversed, which corresponds to the firing of all rules in every combination possible. Since this is usually not reasonable in the testing environment, we propose the following hierarchy of rule-base coverage measures (RBCMs) for rule-based systems in order to guide the selection of test data and give an objective measure of how well a test suite has covered the rule-base.

1. **Each-class** Provide test data to cause traversal of one execution path to each class of the system (i.e. to each node that connects directly to the sink node). This is equivalent to providing, for each class, one test case that concludes that class.
This is clearly a very minimal coverage measure, which we hope is already satisfied by all expert systems developers, regardless of their overall testing strategy.

2. **Each-hypoth** Provide test data to cause traversal of execution paths such that each sub-class is reached, as well as each class. Note that we are not stipulating where the reasoning chains go from each sub-class, only that some reasoning chain can get to each sub-class. From some sub-classes there are reasoning chains that continue on to the classes. This shows that each sub-class is actually reachable from the source and appears to be a relevant part of the system. The set of test data which satisfies this coverage measure is clearly a superset of that which satisfies Each-class.

3. **Each-class-every-sub** There may be many execution paths that connect each sub-class to each class (and there may be no execution paths for some sub-class to class combinations). Provide data such that for each sub-class to class combination connected by some execution path, at least one execution path which includes the combination is executed. This criterion is stronger than Each-hypoth because it requires that we must have data that executes paths that do not just reach each sub-class node, but that also continue on from each sub-class to all classes to which it is connected in the graph.

4. **Each-class-every-finding** There may be many execution paths that connect each finding to each class (and there may be no execution paths for some finding-class combinations). Provide data such that for each finding-class combination connected by some execution path, at least one execution path which includes the combination is executed. This requires data which is a superset of the data needed by both Each-class and Each-hypoth. We can see that Each-class exercises only one execution path to each class, and therefore will only satisfy finding-class combinations which are involved with that single execution path. Furthermore, while Each-hypoth exercises partial execution paths to each sub-class, since we make no stipulation that each of the partial execution paths be part of an execution path that concludes a class, Each-hypoth can be satisfied without
5. **All-edges** Provide data to cause traversal of a set of execution paths such that every inference relationship (every edge in the graph) is utilized along some inference path. While this will not guarantee that all rules will be used in every combination possible, it will guarantee that every rule is used at least once along some execution path. In a rule-base which has no NOFM nodes, the data that satisfies this coverage measure will be a superset of the data necessary to satisfy **Each-class-every-finding** and **Each-class-every-sub**. We can see this by noting that there may be multiple execution paths along which a given finding or sub-class can help lead to the conclusion of a class. In **Each-class-every-finding** and **Each-class-every-sub** we stipulate only that one of these execution paths be traversed, whereas in **All-edges** we stipulate that all must be traversed. In systems that have NOFM nodes it is possible that every edge could be used along at least one execution path without traversing more than one execution path for a finding-class combination. Consider the graph in Figure 3.4. A test case which has only F4 and F5 true will cause the 2-of-5 node to be true and the path from 2-of-5 to G to execute. Next consider a test case with F1, F2, and F3 set to
true. Such a case will satisfy both the 3-of-5 and 2-of-5 nodes. We see that the finding-class relationships between G and all five findings have been satisfied, but only one execution path from F4 and F5 to G was executed. In order to satisfy the All-edges coverage measure, we would also need a test case which had F4, F5 and one additional finding set to true. This would cause G to be concluded on a path through the 3-of-5 node, using the edges from F4 and F5 to 3-of-5.

Referring to the Sample TRUBAC Session in Appendix D (discussion of TRUBAC is in Chapter 4) and Test Data (Appendix E), we can demonstrate how the coverage measures would apply to testing a simple rule-base. Figure 3.5 shows the DAG representation of a simple EXPERT rule-base which suggests treatments to follow when a car will not start. We can see that the first five test cases executed (1, 2, 3, 4, 7) cause the system to conclude each of the eight classes at least once (the class NSTR is concluded by two of the cases). This satisfies the Each-class coverage measure. In this case it also satisfies the Each-hypoth and Each-class-every-sub coverage measures as well. When the Each-class-every-finding measure is considered, we learn that there are 3 finding-class combinations that have not yet been exercised by the test data. Two additional test cases (cases 5 and 6) take care of two of these case, with the third one resolved by test case 8.

Figure 3.6 shows the lattice of relationships that exist among these coverage measures. If a coverage measure higher in the lattice is satisfied, it guarantees satisfaction of the coverage measures below it in the lattice. We note that the coverage measures Each-class-every-finding and Each-class-every-sub are incomparable, where satisfaction of one does not guarantee satisfaction of the other. We can see this by considering the graph in Figure 3.7. This figure represents a section of a rule-base in which a particular finding F can lead to the sub-class I1 in two distinct ways, and F can also lead to the sub-class I2. Furthermore, both I1 and I2 are on paths to a class G (We use the dotted and dashed lines to indicate that the edges represent paths not direct links). In this graph there are 3 execution paths, all of which involve using F to conclude G. The Each-class-every-finding coverage measure would require that we traverse (provide test data to execute) only one of these paths. Assume that we satisfy
Figure 3.5: DAG of simple rule-base.
Figure 3.6: Lattice of coverage measures.

Figure 3.7: Multiple paths from a finding to a class.
Each-class-every-finding by executing one of the paths that goes through \( I_1 \). In order to satisfy All-edges we would need to execute the remaining two paths. However, we can satisfy the Each-class-every-sub criterion by executing only one more path, the path through \( I_2 \). So in this case satisfying Each-class-every-finding does not guarantee satisfaction of Each-class-every-sub.

As another example, consider the graph in figure 3.8. In this case satisfying Each-class-every-finding will guarantee satisfaction of Each-class-every-sub. We can satisfy Each-class-every-sub with a test case that has both \( F_1 \) and \( F_3 \) set to true, concluding \( I \) and whatever sub-classes are on the path from \( F_1 \) to \( G \). However we cannot satisfy Each-class-every-finding unless we also provide a test case which has \( F_2 \) set to true.

We can easily see from these simple examples that in a very complex rule-base, or one for which there is very little test data, using these criteria will help the tester determine the ways in which the data has been deficient in testing portions of the rule-base.

We assume for this testing approach (as in ASSET) the existence of an oracle that
determines if the result given by the rule-base for a test case is correct\(^3\) or not. In this work we do not consider the issue of specifications and how we determine if an answer is correct or not. If an answer is wrong, then the execution path that led to its conclusion is suspect and must be studied for errors. If the answer is right then the path is only of interest to the extent that it helps satisfy the coverage measure(s) chosen by the tester.

The intent of the coverage measures is that they provide the user with information that will lead to the selection or development of additional test cases, or will lead to the discovery of errors in the rule-base. However, we recognize that one of the difficulties often faced by those who test expert systems is a paucity of test data. There is an additional feature of this approach to rule-base testing that can help the user gain insight into the correctness of the system even when little test data is available. In addition to evaluating the five coverage measures, the DAG framework allows for the generation of test data which would lead to traversal of any untraversed execution paths. This synthesized data can then be shown to one or more experts in order to determine if each synthesized test case, which is a reflection of the logic embodied within a section of rule-base (corresponding to an execution path), makes sense in the context of the problem domain which the rule-base was designed to handle. If the expert does not agree that the data represents a plausible case in the problem domain then the section of the rule-base represented by that section of the graph must be reviewed for errors.

3.4 Rule-base Analysis

3.4.1 Graph Construction

The first step of DAG construction is the creation of source and sink nodes. Then the rule-base is processed, starting with the findings and classes. A node for each finding and class is put into the DAG with a link to the source or sink, respectively.

\(^3\)We recognize that for a given test case the experts may not agree on one answer, in which case the system will be considered correct if its answer is in the set of answers agreed on by the experts as possible for that case. There are a number issues that are raised by the desire to have a "gold standard" \cite{BE85} against which to measure the expert system's answers. For example, if the system's answer agrees with that of the expert, should we consider the system to be correct, even if we subsequently learn that both were wrong.
After the findings and classes have been processed, the rule base is processed, one rule at a time, and the interior components of the DAG are created. Rule order is unimportant. For each rule a subgraph is built for the antecedent, and that subgraph is linked appropriately to the node for the consequent of the rule.

Using this approach we are able to determine rule redundancy and conflict (or ambiguity) while graph construction is taking place by identifying that an antecedent is common among more than one rule, and comparing the consequents of those rules. (The details of this are in Chapter 4).

3.4.2 Static Analysis

Some aspects of static analysis of the rule-base will be carried out during graph construction, while others are carried out after graph construction is complete. The situations we would like to identify are (see Chapter 2):

1. redundancy
2. conflict
3. potential conflict (ambiguity)
4. circular rules
5. dangling conditions
6. useless conclusions

As the DAG is being built, redundant rules, directly contradictory rules (pairs of rules with the same antecedent but opposite consequents), and potentially contradictory or ambiguous rules can be identified. (Two rules are contradictory if they have the same antecedent and inverse consequents. Two rules are ambiguous if they have the same antecedent and different, but not inverse, consequents). One of each pair of redundant rules can be left out of the final rule-base, and hence out of the final DAG as well. The developer should be given the opportunity to specify which of two contradictory rules
should be used. This allows us to keep the size of the graph as small as possible, which will reduce the complexity of subsequent analysis over the structure.

In the actual implementation (see Chapter 4) information about redundancy, conflict, and ambiguity is obtained by using a hash table of antecedent and consequent information which is updated and referenced as the rules are processed and the graph is built.

After construction of the DAG is complete, the remainder of the static analysis of the rule-base can be performed. The goal of static analysis is to identify cycles, dangling conditions, useless conclusions, and isolated rules. (A conflict within a single rule, in which the consequent conflicts with the antecedent, will be identified in this portion of the rule-base analysis, as it will cause a cycle in the graph). The key to the static analysis is a depth-first traversal of the DAG. The depth-first traversal provides the following information about the rule-base:

1. If there are any back-edges in the traversal, then the graph is in fact not a DAG and there are cycles in the rule-base.

2. If the traversal produces a depth-first forest, not a tree, then there are dangling conditions (an antecedent component that is not defined as a finding and is not found as the consequent of another rule).

3. If there is a rule consequent which is not known to be a class and is not used in the antecedent of a rule which is on a path to a class, then it is a useless conclusion.

4. If the depth first traversal generates a depth-first forest which also contains useless conclusions then there are isolated rules (rules which are not on some path from the source and also are not on some path to the sink).

In addition, the static analysis can be used to determine if there are classes of the system which appear in no rules of the rule-base.
3.4.3 Dynamic Analysis

Dynamic analysis (validation, dynamic test) of a rule-based system is carried out by executing actual test data. Dynamic methods for testing rule-based systems are applied while the system is operating on some test case(s). Typically, dynamic methods are used to compare the system's line of reasoning and conclusions with that which an expert would give in the same situation. Dynamic methods, therefore, usually correspond to testing a program by running it on data and checking the result against that which is expected. These methods can usually tell the user if each rule has been used or not, but the user has no way of knowing if a rule is not used because it is unnecessary in the system, or because the test cases were too limited. While traditional dynamic methods (e.g. TEIRESIAS [Dav84]) can contribute important information about the consistency and completeness of the system, they do not consider interactions among rules during possible runs of the system. Knowledge refinement systems (e.g. SEEK [PW84] and SEEK2 [GWP85]) consider particular rules in a reasoning chain, but do not consider the absence of rule combinations in the executed reasoning chains. It is because of the weaknesses identified in existing dynamic testing methods (explained in greater detail in Chapter 5) that a data flow analysis approach to testing rule-based systems seems warranted.

There are two main parts to dynamic analysis in our approach: percolating data through the DAG, and evaluating the rule-base coverage measures. Data must be percolated through the DAG for each test case processed by the system, and the coverage measures can be evaluated as often as is desired by the user. Each test case not only contributes to satisfaction of the coverage measures, but it should also generate the result expected from that case. If a test case produces an incorrect result then the portion of the rule-base that it exercised should be examined for errors. After the rule-base is corrected then the test cases should be re-run so that the coverage measures will reflect only inference chains that lead to correct results.
for (each class G) 
{
    if (G is in the graph )
    if (class is unmarked after running test cases)
        Each-class not satisfied

Figure 3.9: Pseudo code for Each-class

3.4.4 Evaluating the Rule-Base Coverage Measures

At various points during execution of the test cases, the user will want to evaluate some or all of the rule-base coverage measures. A good testing strategy is to try to satisfy the RBCMs in an order which increases coverage. As suggested by the lattice of Figure 3.6, one should start by presenting data which satisfies Each-class, which could then be followed by data to satisfy Each-hypoth and then work in parallel on Each-class-every-finding and Each-class-every-sub. We discuss here how we determine whether or not the test cases presented actually were sufficient to satisfy the coverage measures. Pseudo-code for RBCM evaluation is in Figures 3.9 - 3.13.

Each-class

For each class in the rule-base that is also in the graph (if not in the graph then it was in no rules and will be identified during static analysis), if it was not concluded during any test case then Each-class has not been satisfied.

Each-hypoth

For each node in the graph, if it is a class node or a sub-class node then it is examined to see if it was concluded during some test case. If any class or sub-class node is found that was not concluded then Each-hypoth has not been satisfied.

Each-class-every-finding

For each class in the rule-base which is also in the graph, report any finding which could have been used to conclude the class but never was used on a successful path to
for (each node in the graph)
  if (the node is a class or a sub-class and is unmarked
      after running test cases)
    Each-hypothesis not satisfied

Figure 3.10: Pseudo code for Each-hypothesis

for (each class G)
{
  if (G is in the graph)
    for (every finding F which could contribute to the class)
      if (F did not contribute)
        {
          report F-G as unused finding-class combination
          Each-class-every-finding is not satisfied
        }
}

Figure 3.11: Pseudo code for Each-class-every-finding

that class. Each-class-every-finding is satisfied if every finding which could possibly contribute to any class was involved in some test case that concluded the class.

Each-class-every-sub

For each class in the rule-base which is also in the graph, report any sub-class which could have been used to conclude the class but never was used on a successful path to that class. Each-class-every-sub is satisfied if every sub-class which could possibly contribute to any class was involved in some test case which concluded that class.

for (each class G)
{
  if (G is in the graph)
    for (every sub-class I which could contribute to the class)
      if (I did not contribute)
        {
          report I-G as unused sub-class to class combination
          Each-class-every-sub is not satisfied
        }
}

Figure 3.12: Pseudo code for Each-class-every-sub
for (every node V in the graph)
    if (the node is not the source node) /* edges from source to findings
        are never marked -- skip them */

    for (every edge U which points to a neighbor of V)
        /* only care if the edge is not from a class to the sink,
            and the edge wasn't marked during dynamic test */
        if (the edge was not marked yet and it doesn't point to the sink)
            {
                if (V is a finding)
                    report V

                if (U points to a class or a sub-class node)
                    report the class or sub-class

                if (U points to an AND)
                    walk up graph from the AND to find all findings which
                    contribute to positive conclusion of the AND

                    from the node pointed to by U, walk down to an AND and
                    then walk up to find other findings which are necessary to
                    cause execution of the path containing the unexecuted edge
            }

Figure 3.13: Pseudo code for All-edges

All-edges

The Each-class-every-finding (Each-class-every-sub) criterion determines whether
one execution path was traversed by the test data for each finding-class (or sub-class to
class) pair for which at least one execution path exists in the DAG. However, there may
be more than one way in which a finding (sub-class) can contribute to the conclusion
of a class. In order to find execution paths that have not been traversed, we have to
look at edges in the graph. Each unmarked edge is part of one or more basis paths that
were not traversed. Therefore we want to expand all such paths. We do so by starting
a traversal at each unmarked edge, reporting on all finding and hypothesis nodes which
could contribute to a path which contains the unmarked edge. (The complexity of
coverage measure evaluation is discussed in Chapter 6).
3.5 Error Identification

In Chapter 2 we discussed a number of sources for knowledge base errors. At this point we look briefly at whether each of those errors can be identified during analysis by the method outlined above.

- Typographical mistakes – these errors will show up during the static analysis phase, since an erroneous name will appear as an unused finding, a useless conclusion, or a dangling condition.
- Inadvertent errors – if part of a rule is left out then this should show up during dynamic testing since the rule-base should give an incorrect result for one or more test cases. If implication is reversed then there may be a cycle in the rule-base which would be found during static analysis.
- Errors due to not thinking through the application problem domain, and errors due to changes made in an attempt to enable the system to have wider application coverage – as with the previous error, the coverage measures do not really help, as they only check what is in the rule-base, but the errors would still show up during dynamic analysis due to disagreement between the result generated during testing and the expected result.
- Errors introduced when rules are added or changed – these errors could be found during both static and dynamic analysis, depending on the exact nature of the error
- Inability to handle cases that are more complex than those on which development of the system was based – dynamic analysis could show this by generating an incorrect result, but enforcing rule-base coverage alone will not force exposure of this error.
- Errors due to order-dependent rules, with problems caused by unforeseen rule interactions – in our approach we carry out testing of all rule interactions that are possible based on the rule-base, so unintended or unforeseen rule interactions
will be exposed by the dynamic analysis phase, particularly when stronger rule-
base coverage measures are enforced (weaker rule-base coverage measures may
not guarantee that the part of the rule-base that contains the error is actually
exercised).

- Errors caused by incorrectly assuming an input set is unrealistic, when in fact it
  is possible – dynamic analysis can show this, though the coverage measures will
  not force its exposure. The complementary situation, allowing in the rules an
  input set that is impossible in reality, will be exposed by the rule-base coverage
  measures, as the user will be alerted that there is an untraversed execution path
  that would only be executed by a data set that cannot, in reality, occur.

- Errors caused by under-qualified rules, i.e. rules that should cover mutually exclu-
  sive conditions but instead cover overlapping conditions – the coverage measures
  should expose these errors, since erroneous results should be generated for some
  execution paths as the coverage measures are satisfied.

We emphasize here that the coverage measures employed in this testing approach
operate only on the rules that are actually in the rule-base. In general, when we test
a rule-base we are testing the knowledge that is embodied within it. Errors due to an
absence of information or a system which is too narrow or too wide for the problem
domain may not be exposed by the coverage measures alone, although they should
be exposed by a judicious choice of test cases based on the problem domain. As is
the case in traditional software testing, errors due to missing logic are invisible to
any structurally based technique, since testing techniques based on the structure of a
program can only test what is actually in the program. It is for this reason that we do
not suggest that all test data selection be based solely on the structure of the rule-base.
As we will see in the next section, test case selection should include cases that are
relevant to the problem domain, chosen without particular concern to their coverage
qualities, as well as cases that are chosen because of their contribution to rule-base
coverage.
3.6 Heuristics for Test Data Selection

If we can assume that the test suite is complete, then any lack of coverage is caused by rules that are not necessary in the rule-base, and we can perform rule-base pruning (see section 3.8). However, if we remove the assumption of completeness then lack of coverage may provide information about inadequacies in the test set as well as problems in the rule-base. If we know that the test set is or may be incomplete then we must consider two scenarios.

If an incomplete test set leads to complete coverage of the rule-base then we know that the rule-base is also incomplete and is not capable of handling precisely those types of cases that are absent from the test suite. In the second scenario the incomplete test set leads to incomplete coverage of the rule-base. In this case there are sections of the rule-base that have not been tested and our analysis of the testing results and future testing actions will be determined by the degree of coverage which has been obtained.

If the test set satisfies the Each-class-every-sub criterion then we can be confident that the major relationships within the rule-base have been tested. In this situation we might judge the incompleteness within the test set to be fairly minor and we could evaluate the remaining finding to sub-class relationships by reviewing the relevant rules. If we have not obtained this level of coverage then the implication is that there are rather substantial areas of the problem domain for which the rule-base has not been adequately tested. In this case we cannot predict how the rule-base will run for situations which involve the untested relationships. If no more data is available then the most we can say about the rule-base in this situation is that its behavior and performance is unpredictable for cases that would cause it to carry out inferences in the untested sections.

In reality, we do not expect to have complete test sets, in part due to the difficulty in obtaining enough actual cases to completely test an expert system. It is also sometimes the case that there is a large pool of available test data, of which some must be selected for the test suite due to the lengthy time necessary to run all the available cases. However, a poor selection of cases will lead to an incomplete test set. Therefore we
address the question of how to use coverage information to guide test case selection in a way that will maximize rule-base coverage.

Before presenting the heuristics for test data selection, it is useful to have a simple model of a classification system. Let us assume that we are dealing with four findings, three sub-classes, and two classes, as shown by the graph in Figure 3.14. In reality the rules will involve combinations of findings which lead to sub-classes and combinations of findings and/or sub-classes which lead to classes. If we use a binary logic, in which each finding, sub-class, and class can be either True or False, then the full rule-base for such a model would contain all possible combinations of the binary values for the findings, the sub-classes and the classes. In the case shown in Figure 3.14, each of the possible 16 combinations of findings values will lead to one of the 8 combinations of subclass values, which in turn lead to one of the 4 combinations of class values. Usually a rule-base, in essence, represents a subset of this complete model. However, it can be helpful to think of each case in the complete population of available test cases as representing one of the possible combinations of findings which leads to a vector of sub-classes and a vector of classes.

The set of heuristics for data selection which we propose is as follows:
1. For each class, select a test case which uniquely concludes it (a findings vector for which the resulting class vector has only one element true).

2. Select test cases which will conclude unused sub-classes.

3. Select test cases which will cover sub-class to class relations, and direct finding to class relations for findings which do not lead to an interim sub-class.

4. Select test cases which will cover finding to class relations for findings which represent alternative ways to conclude sub-classes.

As we will see in the discussion below, the use of the coverage information, along with meta-knowledge about the pool of available cases, is critical to selecting the test cases in a way that contributes to a useful test of the rule-base.

Generally there are four kinds of meta-knowledge that can help with the selection of a test set that is representative of actual data. We may have information about the (domain-specific) significance of the findings to the classes and sub-classes, as well as information about the probabilistic relationships between findings and classes and sub-classes. We may also have information about types of cases for which it is critical that the system work correctly because of the particularly harmful impact of an incorrect classification. We can select test cases that include findings that are deemed by the expert to have relatively greater significance or that are known to have greater probability of being present for certain classes. (We may also have this kind of information about the relationships between sub-classes and classes). Finally, depending on how the test data was collected and recorded, we may actually have direct information for each test case about what sub-classes and classes are concluded by the test case. By using this information to construct the test set, we will obtain a test suite that covers those sections of the rule-base that are representative of the sections of the logic that are "most important" by some measure (significance to the classes, probabilistic relationship, criticality). If this test set does not provide the level of coverage that we desire, it will cover those sections that reflect the most common and most critical situations. Therefore, it gives us greater assurance that the system has been tested and works correctly for the situations that we expect to encounter most often, as well as
clearly identifying those parts of the system that still require more examination, testing, or refinement.

While we have not yet automated this process, the following series of steps outline the process of test data selection guided by the given heuristics. As a first step we choose, for each class, one test case which concludes only that class, if possible. In general, while we expect rule-based systems to handle ambiguous cases which present more than one possible classification, we would like the testing process to emphasize the ability of the system to discriminate between classes. Therefore we must make sure that both aspects of the system’s reasoning abilities are tested.

Once we have run one test case for each class, we evaluate the coverage measures. Certainly the test set will have satisfied the Each-class criterion. However, Each-hypoth may not be satisfied, with a number of sub-classes that have not been concluded by any of the test cases executed so far. It is often the case, particularly in a medical problem domain, that the test cases will include some information about what sub-classes are involved with the case. If this meta-knowledge is available, then cases should be selected which will involve the conclusion of the as yet unused sub-classes. This corresponds to selecting from our population those test cases which map to sub-class vectors in which the sub-classes of interest are true. This may not completely satisfy Each-hypoth however, as there may be some sub-classes which are involved in no test cases.

The next step is to evaluate the Each-class-every-sub criterion, in order to identify sub-classes that can lead to a class but have not been used to do so by the test data. We would like to select test cases which will satisfy these sub-class to class relationships. However, we would like to do so in a way that still emphasizes the discriminatory capability of the system. Therefore, we look at the unused sub-classes across the different classes. Let us assume we have two classes, $C_1$ and $C_2$, each of which has a number of sub-classes which have not been used by the test data thus far. Let the two groups of sub-classes be $SC_1$ and $SC_2$ respectively. If there is a high degree of overlap between $SC_1$ and $SC_2$ then it may be relatively easy to find a test case which concludes both $C_1$ and $C_2$ and will satisfy many of the sub-class to class relationships for both classes.
However, we first try to find cases for each class which concluded only that class. In order to do this, we look for a test case for C1 which uses the portion of SC1 which is not in the intersection of SC1 and SC2, and similarly for C2 with the unique portion of SC2. If no such test cases can be found then we gradually include the sub-classes which are in the intersection. It may, in fact, be possible to find distinct cases for the two classes even with tremendous overlap in the sets of related sub-classes. However, if necessary, we will select a test case which concludes multiple classes.

Finally, we carry out the same kind of test case selection based on the unused finding to class relationships, determined by the Each-class-every-finding criterion. Again we emphasize discriminatory cases but accept cases that test ambiguous situations. If the appropriate meta-knowledge is available, we will first try to satisfy “critical” finding to class relations, in which the finding is used directly in conclusion of the class, with no intermediate sub-class.

While these heuristics are not yet automated, in section 4.6 we present an example in which the heuristics are applied, by hand, to test case selection for one of the AI/RHEUM prototype systems, with 71 rules and 5 classes, using a pool of 127 test cases.

### 3.7 Class Dependence

Another aspect of rule-base analysis or evaluation is the identification of class dependencies, in which the rules that lead to the conclusion of one class are also highly involved in the conclusion of another class. We would like to know about this situation for two reasons. If two classes, C1 and C2, are dependent, and we have a large number of test cases that we know will be classified as C1, it may be that some of those cases will also be classified as C2. In this way, testing for one class will also help us achieve coverage for the other. The second reason we would like to identify class dependencies is so that we know what aspects of the system’s performance may be affected by system modifications. If C1 and C2 are dependent classes and we change rules for C1 then we should not only rerun test cases which we know conclude C1, but we should also rerun
test cases which we know conclude C2 in order to verify that changing rules for C1 did not inadvertently affect the system’s ability to properly classify C2 as well.

At first we might consider looking at common elements in the finding foundations of pairs of classes, in order to determine dependency. We can quickly see that this will not be a useful strategy. Sharing of findings does not imply that there is any sort of rule sharing. This is very clear in the medical diagnosis situation in which many findings, representing basic symptoms and test results, will be shared by the different classes. However there may be no rules in common at all. To determine if two classes do have rules in common we look instead at sharing or overlap among the sets of sub-classes that can lead to a class. If there is a high degree of overlap among sub-classes then it does imply overlap among the rules and a degree of dependency between those classes.

We can obtain this information quite immediately from data which can be collected while the DAG representation is built. If we accumulate for each class a list of all the sub-classes which can help to conclude the class, and then compare these lists for pairs of classes we will get the desired information. Table 3.1 shows the overlap figures for a small prototype of the AI/RHEUM system for rheumatology diagnosis [Kin85].

<table>
<thead>
<tr>
<th></th>
<th>PM</th>
<th>PSS</th>
<th>SLE</th>
<th>MCTD</th>
<th>RA</th>
</tr>
</thead>
<tbody>
<tr>
<td>PM</td>
<td>0</td>
<td>0</td>
<td>22.22</td>
<td>22.22</td>
<td></td>
</tr>
<tr>
<td>PSS</td>
<td>0</td>
<td>16.67</td>
<td>16.67</td>
<td>16.67</td>
<td></td>
</tr>
<tr>
<td>SLE</td>
<td>0</td>
<td>9.09</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MCTD</td>
<td>18.18</td>
<td>9.09</td>
<td>0</td>
<td></td>
<td>36.36</td>
</tr>
<tr>
<td>RA</td>
<td>28.57</td>
<td>14.29</td>
<td>0</td>
<td></td>
<td>57.14</td>
</tr>
</tbody>
</table>

Table 3.1: Overlap of sub-classes for class pairs.

The figures of interest are those for RA and MCTD (rheumatoid arthritis and mixed connective tissue disease). What these figures indicate is that 57% of the sub-classes which lead to RA also lead to MCTD, while 36% of the sub-classes which can lead to MCTD also lead to RA. The lack of symmetry in these figures results from the fact that the absolute number of sub-classes which can lead to the classes is different. These figures would indicate that modifications made to the rules for RA would likely also affect the performance of the system on cases that should be classified as MCTD, while
there would be a lesser affect on the system's classification of cases as RA if the rules for MCTD were modified.

These results were also borne out by [Ind91] in work on Monte-Carlo simulation-based rule-base evaluation. In the Monte-Carlo approach, the rules for one class are actually modified by random perturbations and then the performance of the test data for all the other classes is evaluated. If there is degradation in system performance for a second class, then there must be a dependence among the class pair made up of the class for which the rules were changed and the class for which the performance degraded. The Monte-Carlo approach involves iteratively perturbing rules for one class and then evaluating performance over all other classes, until rules for all classes have been perturbed and their effect evaluated. In our approach, using a graph representation of the rule-base, we can obtain the equivalent information without actually executing the rule-base.

### 3.8 Rule-Base Pruning

Once a rule-based has been constructed, it may be possible that not all the rules are actually necessary for the rule-base to perform correctly. It is possible, for example, that during an incremental development process some early rules were supplanted by rules added later, but they were never removed from the system. If the system can be pruned, with the removal of some rules or components within rules, and the performance on the test cases is not affected, then it is possible that there were unnecessary rules or that the test cases are not adequate to evaluate the entire rule-base [Ind91].

One approach to pruning is made in [Ind91] in which the rule-base is pruned randomly and its performance evaluated after the pruning. A number of different pruned rule-bases are formed and the one with the best performance over the test suite is selected as the best pruned version of the rule-base.

We can take another approach to pruning, in which the pruning steps are focused by the coverage information. In particular, we can focus on sections of the rule-base which have not been executed by the test data. If a section of the rule-base is never
used or executed during evaluation of the test cases, then either that section contains unnecessary rules or the test suite is not sufficiently rich and additional test cases are necessary which will cause use of the inference paths which go through the unexecuted section of the rule-base. It will usually be up to the developer and/or the expert to decide whether the proper approach is to prune or to add test cases. If we believe that the test suite is truly representative of cases that will be found in the application environment, and the rule-base performs correctly on the test set, then it is more likely that the uncovered portion of the rule-base is in fact unnecessary and can be pruned. If the test set is complete and the rule-base performs incorrectly then the uncovered sections of the rule-base are candidates for rule refinement. The coverage measures will provide information about why the rules were never used, based on findings and subclasses which appear in rule antecedents but are not present in any test cases. Removal of these antecedent components from the rules may generalize them sufficiently that they will correctly handle some of the test cases and will be covered by some portion of the test suite.

3.9 A Metric for Rule-Based Systems

While the coverage measures can be very helpful once we are actually engaged in the process of testing a rule-based system, we would also like to have some way of predicting or measuring the complexity of the system and of the testing process before we start testing. We would also like to be able to compare the complexity of different rule-bases, particularly if we have multiple rule-bases that handle problems within a common problem domain. This raises the interesting issue of whether there is a reasonable analog to the control flow graph and, more importantly, a complexity metric for rule-bases which would reflect the relationship between rule-base structure and system complexity. As a first attempt we consider rule-bases in an if-then rule form. This rule form will allow us to construct basic blocks, as we can do for a procedural program. The problem with this approach, (also identified in [Kip92], discussed in Chapter 5), is that the graph built out of the basic blocks will impose an execution order on a system which normally does not specify execution order. We also have a difference in the nature of control flow
in a rule-base, compared with procedural programs. In a procedural program, each
decision point has two outgoing paths, one for true conditions and one for false conditions.
However, our rule-base representation and other graph based representations, and in
rule-bases in general, we are only interested in situations in which the conditions in
an antecedent evaluate to true. There is no continuing control flow down an execution
path if the conditions evaluate to false.

The graph representation we have proposed is a suitable model of the control flow
within a rule-base. It imposes no particular execution order on the rules, and it does
represent all the logical relations that are inherent within the rule-base. The remaining
issue to address is that of a suitable complexity metric over this graph structure. We
first consider application of the cyclomatic complexity to the graph representation we
are using. Recall that McCabe's cyclomatic complexity for a graph is computed by
\[ CV(G) = e - n + 2p. \]
Considering the graph in Figure 3.15, which represents the rule

\[ \text{If } P1 \land P2 \text{ then } R1 \]

we can see that the graph has 3 edges, 4 nodes, and 1 connected component (which will always be the case for our graphs after the static problems have been corrected), giving a cyclomatic complexity of 1. Clearly this is correct, as there is only one execution path that covers this rule. However, this metric breaks down if we consider a rule with the disjunction of two findings, rather than the conjunction. The graph in Figure 3.16
represents the rule

\[ \text{If } P1 \lor P2 \text{ then } R2 \]

In this case the graph also has 3 edges, 4 nodes, and 1 connected component, giving a cyclomatic complexity of 1. However we can see clearly that there are 2 execution
paths that would have to be traversed in order to adequately test the rule. It seems clear that the cyclomatic complexity metric cannot adequately determine the number of basis execution paths in a rule-base as represented by a DAG. However, this simple example points out that the overall complexity of the rule-base is determined by the kind of logical relations that exist.

We propose a method for counting the actual number of basis execution paths within a rule-base that is based on the kinds of logical relationships in the rule-base as evidenced by the kinds of nodes in the DAG representation. Based on the structure of rule antecedents, we can count possible basis execution paths within a rule-base using the following mechanism:

- For each class node we count one path for each edge into the class node. Clearly each edge into a class node represents at least one distinct path through the rule-base that will lead to conclusion of that class. In Figure 3.17 there are 2 paths to the class $G$.

- For each OR node we count one additional path. In our representation all OR nodes have two parents, and therefore each OR node represents two paths to the associated consequent. We will have already counted one of these paths when we considered the classes, so we count just one more when we consider the OR node. In Figure 3.18 there are a total of 3 paths to $G$. We count 2 paths from
the in-edges at $G$ itself and add 1 more at the OR node.

- For each SUB node, we consider the number of parent edges. As with an OR node, each parent edge represents a possible path which concludes the sub-class. One of these paths will be counted when the classes are considered, so we count additional paths for the additional parent edges. In Figure 3.19 there are a total of 5 paths to $G$. As before, we count 2 at $G$ and 1 more at the OR node, plus 2 more at the SUB node.

- For NOPM nodes, we do not want to enumerate all possibilities, but we do need to minimally cover use of all antecedent components that can lead to the consequent. In order for the node to have a true value, we need at least $N$ true items among the $M$ that are parents of the node. In order to cover all possible antecedent-consequent combinations we will need $\lfloor M/N \rfloor + 1$ distinct data sets, or that
Figure 3.20: Graph of rule-base with OR, SUB, and NOFM.

number of basis execution paths. However, we are already counting 1 path through the NOFM node, so the node contributes an additional \( \frac{M}{N} \) execution paths. (If \( \frac{M}{N} = \frac{N}{M} \) then we add an additional \( \frac{M}{N} \) - 1 in order to avoid over counting). In Figure 3.20 we can count 7 paths to \( G \). We identify 2 of these at \( G \) itself, 2 more at the SUB node, one additional path at the OR, and 1 additional at the NOFM node. This node requires 7 of 10 inputs be true for the node to evaluate to true. We will need at least 2 data sets of 7 true items at the NOFM node to satisfy all 10 antecedent-consequent relationships. One of these data sets is already counted when we consider the paths through the OR, and one more is counted when we consider the NOFM node itself.

- for AND nodes we add no additional paths.

As an example, consider the graph (Figure 3.21) of a simple EXPERT [WKKU87] rule-base, which suggests actions to take when a car will not start. With 44 nodes and 63 edges, the cyclomatic complexity would give a value of 21 for the number of basis paths in the graph. However, by our basis execution path metric we compute only 11 execution paths, as follows:

- 8 edges into the classes (1 edge into each of the 8 classes).
Figure 3.21: Complete DAG of CAR rule-base.
• There is 1 NOFM node, NOFM0, which contributes 1 additional execution path (it requires that one of two items be true, so it will take a total of two paths to minimally cover the edges into the NOFM node, one of which has already been counted).

• There are 7 SUB nodes (filt, strt, batd, flod, chock, empt, cab). Of these, two have two incoming edges, and each of these contributes an additional execution path.

• There are no OR nodes in this graph (no OR relations in the rule antecedents) so there are no additional execution paths due to OR nodes.

This gives us a total of 11 basis execution paths, which is much better than if we viewed the graph as the control flow graph of a program and applied the McCabe complexity metric to it. This implies that we could get complete coverage of the rule-base (All-edges) with at most 11 data sets, one for each basis path.

The execution path metric described here can serve a number of purposes in rule-base development and analysis. If the system tester or developer is shown how each type of node contributes to the number of paths, this information allows the user to see immediately which aspects of the knowledge organization within the rule-base contribute to the overall complexity of the rule-base. For example, consider the information in Figure 3.22, which is generated by TRUBAC (see Chapter 4) for a rheumatology rule-base. We can see immediately that the NOFM nodes contribute considerably (approximately 80%) to the overall number of basis execution paths. As considerable use is made of the NOFM nodes, it might be worthwhile to have the expert review those parts of the rule-base in order to determine if all the items present in each NOFM node are necessary. The temptation might be to replace simple (2 of 3, 3 of 4) NOFM nodes with the explicit representation of the logical relationships that are implicit in the NOFM nodes. However, we caution that this will increase the overall number of execution paths (except in the case of NOFM nodes based on 1 of 2 items, in which case the number of execution paths is unchanged). In general, any sort of disjunctive relationship (either OR, SUB, or NOFM nodes) will increase the complexity of the rule-base, while
rule-bases with a high degree of conjunctive connectivity (lots of AND nodes) will have lower complexity and fewer basis execution paths.

The total number of basis execution paths also represents the maximum number of test cases we would need to get complete coverage of the rule-base according to our strongest rule-base coverage measure (All-edges). However, often the actual number of data sets needed will be less than the number of execution paths, since often, particularly in diagnosis systems, one test set may cover a number of execution paths to different diagnoses (see the TRUBAC session for the CAR rule-base, in Appendix C).

3.10 Comparison of Testing Criteria

We end this chapter with a look at one set of data flow testing criteria, which have been applied to procedural programs, and compare them to our rule-base coverage measures. Specifically, we consider the Rapps-Weyuker data flow criteria [RW85] which are implemented in ASSET [FW85], a data flow testing tool for Pascal programs. The data flow testing criteria were intended to supplement path selection criteria based on control flow analysis, so that the associations between variable definitions and uses would be examined during program testing. Each of the criteria determines if it is
satisfied by the set of complete paths exercised by the test data (complete paths over the control flow graph are paths that start at the entry node and end at the exit node). Specifically, a set of complete paths

- satisfies the \textit{all-defs} criterion if for every variable defined in some node of the graph, there is some path along which the variable is used subsequently in a decision (predicate) or computation.

- satisfies the \textit{all-p-uses} criterion if for every variable defined in some node of the graph there are paths that include each use of the variable in a predicate.

- satisfies the \textit{all-c-uses/some-p-uses} criterion if for every variable defined in some node of the graph there are paths that include each use of the variable in a computation. Should a variable be used in no computations, then there will be a path that includes at least one use of the variable in a predicate.

- satisfies the \textit{all-p-uses/some-c-uses} criterion if for every variable defined in some node of the graph there are paths that include each use of the variable in a predicate. Should a variable be used in no predicates, then there will be a path that includes at least one use of the variable in a computation.

- satisfies the \textit{all-uses} criterion if for every variable defined in some node of the graph there are paths that include each use of the variable in a predicate and each use of the variable in a computation.

- satisfies the \textit{all-du-paths} criterion if for every variable defined in some node of the graph the set of paths contains every path that includes a use of the variable. That is, there may be more than one path from a particular definition of a variable which includes a particular computation or predicate use of that variable and this criterion requires that all such paths be selected (or tested).

A comparison of these data flow path selection criteria to the rule-base coverage measures further illustrates the differences between procedural programs and rule-based systems, as well as highlights the usefulness of the coverage measures we have introduced in this thesis.
We first consider the all-defs criterion, and possible corollaries in the rule-base environment. The rule-base corollaries of a variable definition or assignment is the assignment of a value to a finding when a particular test case is considered and the conclusion of a sub-class during execution of a test case. Some portion of these “definitions” would be covered by an all-findings criterion and an analogous all-subs criterion. An all-findings criterion would require that for each finding we execute some path that has the finding in a rule antecedent and concludes a class of the system. We consider two issues, whether we get the all-findings coverage from our existing method, and whether all-findings represents particularly relevant information in the context of a rule-based system.

Any finding which is found in no rule antecedent will be reported on during the static analysis and, presumably, removed from the system. Paths which only lead to subclasses but structurally cannot continue on to final classes will also be reported during static analysis, and findings which only contribute to those paths will be removed from the system. So we assume that the all-findings criterion would only apply to findings which are actually used along paths from the source node to the sink node in the DAG representation of a rule-base. We can see that Each-class-every-finding actually gives us more than the all-findings information. This is because Each-class-every-finding requires that each finding be used to get to all the classes it can influence, whereas all-findings would require that each finding be used to conclude only a single class. So satisfaction of Each-class-every-finding actually gives us a stronger test of the system than would be required by an all-findings criterion.

Would all-findings be a useful criterion to add to a rule-base analysis scheme? To answer this we have to consider what all-findings really means in the context of a rule-base. After static analysis we can remove any findings that do not lead to the conclusion of system classes. Therefore we can presume that all other findings are necessary for conclusion of the classes of the system, that all remaining findings lie on execution paths to the classes. Showing that we can find data to traverse a single path from a finding to a class provides little new information. We gain much more information by showing that we can use each finding to attain each class for which it is relevant. Therefore
Each-class-every-finding, which subsumes an all-findings criterion, is more useful in the context of testing rule-based systems.

The other form of definitions in a rule-base, that of sub-classes, is addressed by the Each-hypoth and Each-class-every-sub coverage measures. We can see that satisfying Each-hypoth ensures that each sub-class is assigned a value, but does not guarantee that the assigned value is subsequently used to conclude a class. As with the Each-class-every-finding measure, Each-class-every-sub actually gives us better coverage than would be required by the all-defs criterion. All-defs would require only that each sub-class be used on one path to a single class, whereas Each-class-every-sub considers whether each sub-class has been used to conclude every class to which it can contribute. Following the argument that was used above for all-findings versus Each-class-every-finding, we can see that the Each-class-every-sub criterion is also a stronger criterion which subsumes an allsubs criterion.

Next we consider the data flow criteria which examine variable uses. In the rule-base languages to which we have applied our approach thus far there are no computational uses of findings and sub-classes. Findings and hypotheses are used only in logical expressions, which correspond to predicate uses in the data flow analysis terminology. The criterion of all-c-uses/some-p-uses is interesting in that it reduces to some-p-uses in the absence of computational uses of the definitions, which is essentially the same as the all-findings and allsubs criteria discussed above.

Next we see that the three criteria all-p-uses, all-uses, and all-du-paths all require the same path set, which is the all-p-uses set. We recall that the difference between all-defs and all-p-uses is that the former requires just a single use of a definition, whereas the latter requires that we exercise every predicate use of a definition. In the rule-base case this would require that for each use in an antecedent of each finding and sub-class we traverse one execution path that covers that use. In the context of a requirement that the paths of interest are only complete paths, all-p-uses is the same as All-edges, requiring complete coverage of all execution paths.

Based on this review of analysis techniques for procedural programs we can see that there are many ways in which the analysis of rule-bases is less complicated than is
the analysis of procedural programs, particularly because of the emphasis on predicate
uses in rule-bases. However, we have also seen that the lack of a predictable execu-
tion sequence presents a difficulty that does not exist when dealing with procedural
programs. The method presented in this thesis is useful because it allows us to eval-
uate the rule-bases using an approach which is comparable to a data flow analysis of
procedural programs, but takes into account the ways in which rule-bases differ from
procedural programs.
Chapter 4

The TRUBAC Tool

In this chapter we discuss TRUBAC, a C language implementation of the rule-based coverage testing method proposed in Chapter 3. We include a discussion of the application of TRUBAC to several rule-bases. We look at both simple and more complex rule-bases, without dynamic computation of uncertainty. We also discuss the translation programs that are used to put the rule-bases into the input form that TRUBAC expects. By establishing a common input format it is possible to use TRUBAC to test rule-bases written in a variety of rule-base shells or languages, as long as a front-end is written that first translates the rule-base into the proper format. We also present examples of the use of information generated by TRUBAC to guide rule-base pruning and test case selection. In addition, we have used the coverage measures in TRUBAC to evaluate the quality of testing achieved by random test case selection. Finally, we end the chapter with a brief analysis of TRUBAC as a verification and validation tool.

In order to create a tool that could be used to evaluate rule-bases which originated in different rule-base languages, we selected a generic rule-base language in which TRUBAC expects input rule-bases to be written. Before a rule-base can be evaluated it must be translated into this language. The overall picture of the rule-base testing process changes slightly from that presented in Chapter 3. In figure 4.1 we can see that before the actual rule-base analysis can be performed first the rule-base must be translated into the language expected by TRUBAC. The translation is carried out by a program which is customized for the input language. Our experience with two different rule-base languages is that once we had built one working translation program, it was quite easy to treat that translator much like a compiler with the desired back end. We developed the second translator by modifying the sections that parse the input rules so
that they were suitable for rules in the second rule-base language.

The user interface of TRUBAC is quite straightforward and the actual tool should be easily used by rule-base developers outside of research settings (see the script of a TRUBAC session in Appendix D).

4.1 Rule-base Language

TRUBAC expects the rule-base to be in the following format:

*FINDINGS
  <list of findings, one per line>
*CLASSES
  <list of classes, one per line>
*RULES
  <series of rules>

Rules are in the form “if Expression then Consequent”, where the consequent is a single name of either a class or a sub-class, and the antecedent expression conforms to the grammar shown in Figure 4.2.

We can see from the grammar that rule antecedents can include disjunction, conjunction, and negation of findings and sub-classes. Furthermore, there are two constructions
Figure 4.2: Grammar for antecedent expressions accepted by TRUBAC

in this language that deserve particular discussion, the between and the nofin constructions. Both of these were initially created to facilitate translation of EXPERT [WK84, WKKU87] rule-bases so that they could be tested with TRUBAC. The between construction is used to represent an EXPERT rule with a range of confidence values for a hypothesis, such as

$$H(\text{FLOD, .2:1}) \rightarrow H(\text{WAIT, .9})$$

which says that if the certainty factor associated with sub-class FLOD (engine is flooded) has a value between .2 and 1, then we can make the conclusion WAIT (wait 10 minutes before starting the car again or depress accelerator to floor while starting) and associate a certainty factor of .9 with it. In the TRUBAC rule language this rule becomes

if between(flod, 0.20, 1) then wait(0.90)

The nofin construction is used to represent an EXPERT feature which allows the rule-base designer to specify that if a specified number (numneeded) of a list of findings or sub-classes are true then the entire nofin construction is true. In essence this construction allows the designer to avoid expanding a list of relevant findings and hypotheses into all possible sub-groupings. For example, in EXPERT this construction would be used in a rule such as

$$F(\text{TEMP, 0:50}) & [1:F(\text{SCRN, T}), F(\text{OCRN, T})] \rightarrow H(\text{CHOK, .6})$$
This rule says that if the finding TEMP (temperature) is between 0 and 50 and at least one of the findings SCRN (starter is cranking slowly) or OCRN (the starter is cranking normally) is true then we can conclude the sub-class that the choke is stuck, with an associated certainty factor of .6. In the TRUBAC rule language this rule becomes

\[
\text{if between(temp, 0, 50) AND nofm(1, scrn, ocrn) then chok(0.60)}
\]

The terms within the `nofm` list can also include negation or the `between` construction.

### 4.2 Graph Construction

The first step of DAG construction is the creation of `source` and `sink` nodes. Then the rule-base is processed, starting with the lists of findings and classes. Each finding is put into the hash table, as well as into the DAG. The graph node created for the finding will have the source as its parent, and will appear on the neighbor list of the source. Classes are not put into the DAG immediately, but instead are put onto a separate list. This allows us to determine, for each rule processed, if its consequent is a class or a sub-class. Furthermore, it makes it extremely simple to determine, after the DAG is constructed, if there are any classes for which no rules appear in the rule-base. Creation of this extra class list does not require a significant amount of extra storage space, since the number of classes in a system tends to be very small, relative to the number of findings and sub-classes.

After the findings and classes have been processed, the rule base is processed, one rule at a time, and the interior components of the DAG are created. Rule order is unimportant. As each rule is processed, the following steps are carried out:

1. Parse the antecedent, constructing a graph representation of it. In this graph the findings and sub-classes appear as leaf nodes, and the AND, OR, NOT, NOFM operators appear as interior nodes, with one as the root of the graph. For example, the antecedent of the rule

\[
\text{if a or not (b or c) then d}
\]

will result in the graph shown in Figure 4.3.
2. Convert the antecedent to Conjunctive Normal Form (CNF). This is done by modifying the graph created in step 1, rather than working with the rule text directly. For our example rule, Figure 4.4 shows the graph after the first conversion step, with the not applied to its argument so that any not in the graph will have only a single finding or sub-class node as its child. Figure 4.5 shows the graph which results from the completed conversion to CNF. Note that the node for the finding (or sub-class) \( a \) has been replicated in the process of converting to CNF. While in general conversion to CNF can cause a combinatorial explosion in the number of terms (precisely because of the sort of node replication that was necessary with node \( a \) in the example), we do not expect a problem in this...
application due to the reasonable size of antecedents (on the order of 5 terms). Furthermore, when we connect the antecedent graph to the larger rule-base graph we use the existing nodes for findings, sub-classes, and classes so that we will see multiple edges to a node (such as to the node \( a \) for the example) but we will not actually store two nodes for \( a \) in the graph. See Chapter 6 for a further discussion of the complexity of the graph construction algorithm.

3. Create a hash table entry for the full antecedent, in the form of an ordered list of conjuncts, in which each conjunct is also represented by an ordered list. Look up the antecedent in the hash table.

4. If the antecedent is already in the hash table then note that a redundancy check will have to be done after the consequent is seen. Otherwise the antecedent is put into both the hash table and the DAG.

5. Get the consequent. If it is in the hash table already, then retrieve its graph location. Otherwise put it into both the hash table and the DAG, making appropriate connections to the sink node if the consequent is a class.

6. Make the necessary graph connections between the antecedent and the consequent by making the consequent a parent of the antecedent and the antecedent a child of the consequent. We do this by using the root of the graph containing the CNF representation of the antecedent as the node to and from which connections to the consequent will be made. For our example rule, the graph with the consequent added is shown in Figure 4.6.
7. Up to this point the graph of the rule has been constructed as a binary tree, with each node having left and right children. Now we must reverse the pointers to integrate the rule's graph into the larger DAG structure. We can think of this process as one in which the tree is essentially turned upside down. Basically, every child in the tree becomes a parent of its "tree parent", and every parent in the tree becomes a neighbor (child) of its "tree children". Furthermore, we eliminate all NOT nodes by moving negation information onto the appropriate edge. Finally, wherever a finding appears in the antecedent tree, we connect instead to the node for the finding which is already in the DAG. For our example rule, the relevant section of the final DAG is shown in Figure 4.7 (assuming that \(a, b, c\) are all findings of the system, and are therefore connected to the source).

In TRUBAC we are able to determine rule redundancy and conflict (or ambiguity) while graph construction is taking place by using the hash table to find antecedents that are common among more than one rule. Ideally one of each pair of redundant or conflicting (ambiguous) rules will be left out of the final rule-base. Although leaving redundant rules in the rule base does not significantly increase the size of the final DAG, since common antecedents will not be repeated in the DAG, redundant rules can cause serious maintenance problems, and conflicting or ambiguous rules can cause erroneous results when the expert system is used.

The specific situations TRUBAC can identify are:

1. Redundancy – same antecedent in each of two rules leads to the same conclusion in each case.
2. Contradiction – same antecedent in each of two rules leads to the opposite conclusion in each case; rule consequent conflicts with (element of) rule antecedent.

3. Potential Contradiction or Ambiguity – same antecedent in each of two rules leads to two seemingly independent conclusions, or to two related conclusions which are not contradictory. This could be due to incompleteness of the rule-base, or to implicit uncertainty within the rules. The user should be made aware of this situation, but it is not necessarily an error.

As an individual rule is evaluated, TRUBAC carries out a series of checks in the hash table, and it may enter antecedent information into the hash table. First, TRUBAC checks to see if the individual components of the antecedent are in the table. All components previously identified by the user as findings, or concluded as consequents of previously processed rules, will be in the table already. Any components not already in the table will be added. TRUBAC then does a table look-up to see if there is an entry for the complete antecedent of the rule. The location of each antecedent in the table is computed based on the elements found in the antecedent. If there is not already an entry for the complete antecedent, then it is entered into the table. The table entry includes a list of the CNF clauses which make up the antecedent and a pointer to the node in the DAG for the antecedent. (While the antecedent actually corresponds to a subgraph, the hash table entry will point to the node from which the antecedent connects to the node for the rule consequent. This is the same node that had been the root of the CNF representation of the antecedent, before the edges were reversed). Similarly, after the hash table is checked for the antecedent, and modified if necessary, TRUBAC looks up the consequent to determine if it is in the table already, and adds it, if necessary, to both the table and the DAG.

In order for the hash table look-up to provide us with the information we need about the rule-base, the location computation mechanism must be based on the components of the antecedent, but be independent of the relative position of components in the antecedent. For example, consider the two rules:

Rule 1. If \( P_1 \) \& \( P_2 \) \& ... \& \( P_{n-1} \) \& \( P_n \) then \( R \)
Rule 2. If \( P_n \& P_1 \& \ldots \& P_{n-2} \& P_{n-1} \) then \( R \)

found in different parts of the rule-base. When Rule 1 is processed an entry is made in the table for the antecedent \( P_1 \ldots P_n \) and the antecedent will have \( R \) as a neighbor in the DAG. When Rule 2 is processed, we would like to have the same table location computed for its antecedent. We accomplish this by basing the hash function on all of the names of the terms in the antecedent expression, so that the order of the antecedent terms is irrelevant. In this case, when the hash table lookup is carried out for Rule 2, there will already be an entry in the computed location. However, because TRUBAC uses a chained implementation for the hash table, we cannot assume that an entry at the antecedent's hash table location means that the same antecedent has already been seen in another rule. TRUBAC has to compare the current antecedent with all antecedents that hang off the particular hash table location computed for the antecedent. In order to facilitate this comparison, each antecedent is stored as an ordered list of clauses, with each clause in turn stored as an ordered list of terms. TRUBAC does a simple sequential search of the elements hanging off the hash location to see if any match the current antecedent. We chose sequential search for this because of our belief that the size of the hash table is sufficiently large relative to the number of antecedents expected, so that we do not expect to find more than two or three antecedents (or antecedent components) hanging off any one hash location.

If TRUBAC determines that, in fact, the antecedent of the current rule does already appear in the hash table, and therefore in the DAG, then it checks for redundancy or conflict. It does this by sorting the list of neighbors of the antecedent (which is really the list of consequents which result from the truth of the antecedent) and comparing the list elements to each other. If TRUBAC finds two consequents which are the same, then it reports the corresponding rules as redundant. If it finds two consequents which are the same but one was negated in its rule, then it reports a conflict between the corresponding rules. (Note that, since all NOT nodes were removed from the final graph and negation information is carried on the edges, conflicting rule consequents will
appear next to each other in the sorted list of neighbors). Finally, if TRUBAC finds two consequents which are different then it reports the corresponding rules as having a potential conflict or ambiguity. The user must then examine the rules in question and determine if there is in fact a problem with the rules or if they can remain, unchanged, in the rule-base.

4.3 Static Analysis

As we saw in Chapter 3, after construction of the graph has been completed, static analysis of the rule-base can be carried out by using a depth-first traversal of the graph. TRUBAC carries out the depth-first traversal with the following modification. TRUBAC starts the traversal at the source node, marking nodes as visited in the usual fashion. However, if the traversal arrives at a dead-end at any node other than the sink node, nodes on the dead-end path which have no other neighbors are re-marked as “dead” as the traversal backs up, in order to show that there is no path to a class from this point. The reason for this is as follows: as the traversal proceeds it will arrive at nodes that are already marked. Since the traversal does not continue to visit previously visited nodes, it has to be able to determine whether a previously visited node leads to a class or not. Using two kinds of marking allows us to differentiate between paths that lead to a class and paths that do not lead to a class.

4.4 Dynamic Analysis

As we saw in Chapter 3, there are two aspects to dynamic analysis that must be implemented in TRUBAC: data percolation through the DAG and evaluation of the rule-base coverage measures. In order to percolate the data associated with a test case, first the input data is read and each finding node is given a value if there is one for it in the input data. In order to distinguish between true, false, and unknown data, we actually set two fields in each node. One field contains the value provided by the user, if there was one. The other field indicates whether or not the user supplied a data value for this finding. This scheme allows us to distinguish between a finding which is false
because the user knew that information and a finding which is false because the user
provided no data for it.

After the finding values are set, the nodes are considered individually in topological
sort order. In this way we can be sure that by the time we consider an interior node in
the DAG, all values necessary to determine its state will be available. For the different
node types, the action taken is as follows:

- For the source, sink and finding nodes, do nothing.
- For class nodes, OR nodes, and sub-class nodes, if any single parent node is true
  then give an appropriate value to the node (could be a specific value or a discrete
  Boolean value such as "true")
- For AND nodes, if all parents are true then give an appropriate value to the node.
- For NOFM nodes, if at least the specified minimum number of parents are true,
  then give an appropriate value to the node.

We continue to indicate whether a node’s value is the direct result of user data or not.
This is particularly important in a situation in which there is negation on an edge.
A false value at a node which results from user information (directly or indirectly)
should be converted to true along an edge with negation. However, a false value which
represents no information will remain false, even upon negation. (In the future a simple
command line switch in TRUBAC will allow the user to select whether nodes which are
false because the user provided no data should be considered as false or as unknown
values during data percolation).

At each node we use both a value field, to indicate the effect of the current test case
on the node, and a dynamic mark field, which is used to indicate whether the node has
had its value set during execution of any test case. In addition to having a value which
may be set during a test case, each node has associated with it two lists of findings
and two lists of sub-classes. The all-findings and all-subs lists are established after the
graph is built, indicating all the findings or sub-classes which can possibly contribute
to conclusion of the node. The other lists are the pass-along and sub-pass-along lists
that are updated during execution of all the test cases, indicating which findings and sub-classes have actually been used to conclude the node in the course of test case execution. For example, consider the graph in Figure 4.8. Let us assume that we have a test case in which A and B are false, while C and D are true. Since A and B are both false, the result at node AND1 is false, and no findings are added to the pass_along list at that node. When we consider node AND2, even though C is true the result of AND2 is false, so ultimately no findings will be added to the pass_along list of node P. Meanwhile, since C and D are both true they will be passed along by node AND3 and will be placed on the pass_along list of node Q. In general, as a series of test cases is processed, some findings may be passed along (in relation to a particular class node) in one case but not in another case. This information is ultimately used to determine which finding-class combinations have and have not been traversed by the set of test cases which have been executed by comparing the pass_along and all_findings lists at the class nodes. Similar information is gathered for use in evaluating the Each-class-every-sub coverage measure by comparing the sub_pass_along and all_subs lists.

After reporting on untraversed execution paths, TRUBAC will offer to generate data sets which would lead to coverage of the untraversed paths. If the user has more data available, at this point the user could opt to continue to provide test cases, chosen so as to exercise the as yet untested section of the rule-base. Alternatively, the user could have TRUBAC generate data sets, which it does by carrying out the same traversal used to evaluate All-edges and reporting the findings that are involved in the execution path. These data sets can then be reviewed by an expert in order to determine if they, and the corresponding sections of the rule-base, reflect realistic situations within the problem domain. If the experts agree that the generated cases represent cases within the
4.5 Applications of TRUBAC

For this dissertation TRUBAC was applied to four rule-bases written in two different languages, for two different inference mechanisms. Three of the rule-bases were originally written for use with the EXPERT [WK84, WKKU87] system, while one was written in a generic language based on PROLOG [Pre91]. In each case the rule-base was translated into the form accepted by TRUBAC (see section 4.1), and then the rule-base and various test cases were input to TRUBAC. In this chapter we discuss test cases in the two languages separately, as different translation and implementation issues are raised by each language.

4.5.1 EXPERT Rule-Bases

An EXPERT rule-base (or consultation model) contains hypotheses, findings, and rules which relate the findings and hypotheses. Appendix A shows the basic format of an EXPERT system.

Classes

Translation of EXPERT rule-bases into the TRUBAC input language is fairly straightforward. An EXPERT rule-base starts with a Taxonomy section and an optional Treatments section (a simple EXPERT rule-base is shown in Appendix B). If both sections are present then the identifiers listed in the Treatments section are classes of the system and will be listed as such in the TRUBAC input, whereas the identifiers listed in the Taxonomy section are sub-classes and can be ignored initially since they will be found again when the rules are processed. If there is no Treatments section
then the identifiers listed in the **Taxonomy** section will be listed as classes in the TRUBAC version of the rule-base.

**Findings**

The **Treatments** and **Taxonomy** sections are followed by the **Findings** section. This contains the names of all the findings for which the user can specify values when the system is executed. In **EXPERT** **Findings** are categorized according to the various ways by which **EXPERT** can get information about the findings. **EXPERT** can ask the user for information about the findings by one of four mechanisms (checklist, multiple choice, numerical, or yes/no) or it can compute it using **functional findings**. However, regardless of how the user provides the information to the system, each finding will either be set to true or false, or to a numeric value. Each identifier present in the **Findings** section will be included in the **Findings** section of the TRUBAC input.

**Functional findings** consist of simple arithmetic equations that use the values of certain findings to compute values for new findings. For example

\[
f(d_{c3}) = c_{3l} - c_{3p}
\]

uses the values of \(c_{3l}\) and \(c_{3p}\) (numeric values provided by the user) to compute the value of the finding \(d_{c3}\). In the subsequent rules \(d_{c3}\) will be used, whereas \(c_{3l}\) and \(c_{3p}\) will not appear at all. In order for TRUBAC to have the proper values for these **functional findings** it will also have to compute them for each test case after the test data is obtained from the user. The following steps allow **functional findings** to be handled in TRUBAC much as they are in **EXPERT**.

If **functional findings** are found during translation of the **EXPERT** rule-base, they are converted to postfix expressions and output to a functions file which will be associated with the translated rule-base. When the rule-base translation is complete, each line of the functions file will contain one postfix expression which is the translation of one **functional finding** from the **EXPERT** rule-base. The dynamic test phase of TRUBAC consists of a 4 step loop:

1. initialization – all findings are set to false, if Boolean, or -100, if numeric
2. read test case data

3. interpret functional findings

4. percolate data through the DAG

In the interpreter step TRUBAC checks to see if the functions file exists. If it does, then each line of the file is interpreted, using the values provided as input for the test case, and the resulting values are stored in the appropriate graph nodes which correspond to the computed findings.

Rules

Finally, the Rules section of the EXPERT rule-base is processed. EXPERT rule-bases can contain three different kinds of rules: FF (finding to finding rules, which are optional), FH (finding to hypothesis) and HH (hypothesis to hypothesis). Each of these is handled in a somewhat different way during the translation process.

FF rules are intended to describe implied values for some findings [WKKU87]. That is, based on the value given by the user for one finding, an FF rule will direct that current values for other findings be kept or those findings be reset to "unknown". Given the structure of dynamic testing in TRUBAC, we assume that all finding data for a particular test case will be given by the user or computed via functional findings but not inferred from other findings. Therefore we do not translate FF rules at all, but rather require the user to explicitly present all finding values relevant to each test case.

FH rules combine findings in order to confirm or deny a sub-class. The format for FH rules is [WKKU87]

\[ X \land X \land \ldots X \rightarrow H(MNE, CF), -1 \leq CF \leq 1 \]

where \( H(\ldots) \) refers to a hypothesis, CF is a certainty factor in the range \([-1, 1]\), and the conjuncts in the antecedent are in one of the two forms:

\[ X = F(MNE, TVAL) \text{ or } X = \{n:F(MNE_1, TVAL), F(MNE_2, TVAL)\ldots\} \]
In these terms \( F(\ldots) \) refers to a finding and its value, MNE is a finding or hypothesis mnemonic, and TVAL is a truth value (T, F, U\(^1\), or a numeric range in the format \textit{low:high}). The second form is the formal representation of the NOFM construction, where there would be a total of \( m \) terms, \( n \) of which must be true for the expression to be true. We translate the various parts of an FH rules as follows:

- A term in the form \( F(\text{MNE}, \text{T}) \) translates simply to the mnemonic (MNE).
- A term in the form \( F(\text{MNE}, \text{F}) \) translates to the negation of the mnemonic (not MNE).
- A term in the form \( F(\text{MNE}, \text{low:high}) \) translates to a \textit{between} construction
  
  \[ \text{between(} \text{MNE, low, high} \text{)} \]

- A term in the form

  \[ [n:F(\text{MNE}1, \text{TVAL}), F(\text{MNE}2, \text{TVAL})\ldots] \]

  will translate to an \textit{nofm} construction (see section 3.1).

  \[ \text{nofm}(n, f(mne1, tval), f(mne2, tval)\ldots) \]

The right hand side (consequent) of an FH rule translates to the mnemonic followed by the certainty factor. That is, a consequent of \( \text{H(}\text{MNE, } .9) \) will be translated into \( \text{MNE}(0.90) \). The conjunctions in the antecedent of an FH rule are retained and an \textit{if} and \textit{then} are used to indicate the overall rule structure for the TRUBAC version of the rule, rather than the \( \rightarrow \) used in the EXPERT rules. Two FH rules, from a small system that diagnoses car trouble, and their translation into the TRUBAC format, are shown in Figure 4.9.

HH rules differ from FH rules in several ways. First, there can be both findings and hypotheses in the rule antecedent. Secondly, in EXPERT, HH rules are grouped within a high level IF-THEN structure. That is, as can be seen in Figure 4.10, there is an IF clause that determines whether all the rules in the corresponding THEN section can be evaluated. Each rule within the THEN section also has an implied if-then structure,

\(^1\)U represents unknown in this context.
if between(temp, 0, 50) AND nofm(1, scrn, ocrn) then chok(0.60)
if scrn AND dim then batd(0.70)

Figure 4.9: Two rules in EXPERT and TRUBAC forms.

*HH Rules
*IF
F(FCWS,T)
*THEN
H(FLOD,.2:1)→H(WAIT,.9)
H(CHOK,.2:1)→H(OPEN,.5)
H(EMPT,.3:1)→H(GAS,.9)
H(FILT,.4:*)→H(RFLT,.8)
H(CAB,.5:*)→H(CLEN,.7)
H(BATD,.4:1)→H(GBAT,.8)
H(STRT,.4:1)→H(NSTR,.9)

Figure 4.10: Sample HH Rules
\[ F(RNP,T) \land H(ENASH,.9\text{:*}) \land F(SM,F) \rightarrow H(RNESH,1.) \]
\[ H(RNEH,.9\text{:*}) \land \\
[1: H(RAYES,.9\text{:*}) \land H(MYOSS,.9\text{:*}) \land H(COPHV,.9\text{:*})] \\
\rightarrow H(RR201,1.) \]

if \( rnp \) AND \( \text{between}(enash, 0.90, 400000) \) AND not \( \text{sm} \) then \( \text{rnesh}(1.00) \);

if \( \text{between}(rneh, 0.90, 400000) \)
AND \( \text{norf}(1, \text{between}(rayes, 0.90, 400000)) \),
between(\( myoss, 0.90, 400000 \)) ,
between(\( cophv, 0.90, 400000 \)) then \( \text{rr201}(1.00) \)

Figure 4.11: Rules using \textbf{between} construction.

with the left hand side of the rule representing the antecedent and the right hand side representing the consequent. It would seem that the logical thing to do would be to fold the IF clause for the section into the antecedent of each rule within the THEN part of that section. In fact, it turns out that this is not necessary in many cases. For example, a technique used in EXPERT rule-bases is to group the HH rules for a particular class within an IF-THEN structure for which the IF clause is simply \( H(\text{class}, -1:1) \). Since this will always be true when TRUBAC processes test cases for the rule-base (all non-finding nodes are set to false (0) before each test case is processed), it is not necessary to include it in each rule within the section. Once the IF section is translated, translation of the rules within the THEN section is largely like the translation of FH rules. Two HH rules, from a rule-base that does diagnosis of rheumatoid arthritis diseases, and their TRUBAC versions, appear in Figure 4.11. Note that the EXPERT numeric range allows a * in either the low or high positions. In the low position it constrains the identifier in question to have a value less than or equal to the upper range value. In the high position it constrains the identifier in question to have a value greater than or equal to the lower range value. In the TRUBAC version we replace the * with either 400000 (if it is in the high position) or 0 (in the low position).

Finally, the order in which the rules appear in the original EXPERT rule-base and the TRUBAC version is irrelevant, as the same graph will be built regardless of
<table>
<thead>
<tr>
<th>RULE-BASE</th>
<th>FINDINGS</th>
<th>CLASSES</th>
<th>RULES</th>
<th>GRAPH NODES</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAR</td>
<td>14</td>
<td>8</td>
<td>17</td>
<td>43</td>
</tr>
<tr>
<td>RHEUM</td>
<td>151</td>
<td>8</td>
<td>76</td>
<td>333</td>
</tr>
<tr>
<td>RH0184</td>
<td>864</td>
<td>28</td>
<td>940</td>
<td>2623</td>
</tr>
</tbody>
</table>

Table 4.1: Size of EXPERT rule-bases and associated graphs.

rule order. Appendix C contains the TRUBAC version of the rule-base presented in Appendix B.

Results of using TRUBAC on EXPERT rule-bases

We translated 3 EXPERT rule-bases into the input language expected by TRUBAC, and then used TRUBAC to test the rule-bases. One rule-base (CAR) is very small (17 rules) and suggests steps that may help when a car will not start. The other two rule-bases (RHEUM and RH0184) represent two early versions of the AI/RHEUM [Kin85] system that diagnoses rheumatoid arthritis diseases. Table 4.1 shows the sizes of these rule-bases and the corresponding graphs created by TRUBAC. For each rule-base we used TRUBAC to carry out both the static and dynamic analyses. In the case of these three EXPERT rule-bases there is sample data available which dates from the original development of these systems.

In the case of the CAR rule-base (shown again in Figure 4.12), the graph is small enough that we would be able to get all the information provided by the static analysis by simply looking at the graph. The static analysis shows that three findings (mgas, ngas, foth) are never used in the rules. We can see this clearly in the graph as each of these nodes has an incoming edge from the source but no outgoing edges. On the other hand, in a larger rule-base, such as the RHEUM rule-base, the graph is too large to take in visually as a whole. Therefore the messages provided by TRUBAC are very useful.

While the graph is being constructed TRUBAC informs the user of rule redundancies.

---

2Unfortunately there is no record of the systems as they were modified during development, and which test cases led to the discovery of what kind of errors.
Figure 4.12: Complete DAG of small rule-base.
and (potential) rule conflicts, if any exist. For example, TRUBAC reports on a potential
conflict between the following two rules from the RHEUM rule-base (checking for DNA
antibodies):

\[
\begin{align*}
\text{if not dnap AND not dnaci then } & \text{dnal(-1)} \\
\text{if not dnap AND not dnaci then } & \text{dnam(-1)}
\end{align*}
\]

TRUBAC sees that the antecedent of these rules is the same and the consequent is
different. Therefore this may reflect a possible conflict. (Had the consequent of one
rule been the exact negation of the consequent of the other rule then TRUBAC would
have identified it clearly as a conflict).

In the static tests on the Rheum rule-base we are informed about a number of
other problems with the rule-base. For example, TRUBAC finds that renal (renal
involvement) is a useless conclusion. That is, it is concluded by a rule but is not a class
and is not used in the antecedent of any other rule. Therefore any path through the
graph that ends at the renal node will not be able to proceed to a class of the system.
We are also told that it is rule 27 in the rule-base which concludes renal, allowing us
to look back at the rule-base and determine if rule 27 has an error in it and should
be corrected, is unnecessary and can be removed from the system, or is in a part of
the system that has not yet been completed. We also find out that a large number of
findings are used in no rules of the system. Again this could show us that either we plan
to ask the user for more information than is necessary, or we have not implemented
the entire expert system. This latter point would be born out by the fact that the
static analysis also informs us that there are a number of classes which appear in no
rules. Table 4.2 shows the size of the three EXPERT rule-bases, and the corresponding
graphs, after correction of all problems identified by the static analysis. Of course, one
could choose to proceed with dynamic analysis even if the statically identified problems
have not been corrected. In this case the dynamic analysis may take more time since the
rule-base, and therefore the graph, may be larger than it would be were the statically
identified problems removed.
<table>
<thead>
<tr>
<th>RULE-BASE</th>
<th>FINDINGS</th>
<th>CLASSES</th>
<th>RULES</th>
<th>GRAPH NODES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Car</td>
<td>11</td>
<td>8</td>
<td>17</td>
<td>40</td>
</tr>
<tr>
<td>Rheum</td>
<td>66</td>
<td>5</td>
<td>71</td>
<td>241</td>
</tr>
<tr>
<td>RH0184</td>
<td>415</td>
<td>26</td>
<td>911</td>
<td>2110</td>
</tr>
</tbody>
</table>

Table 4.2: Size of revised EXPERT rule-bases and associated graphs.

We constructed additional test cases for dynamic testing of the CAR rule-base, and used existing test cases for both the RHEUM and RH0184 rule-bases. Given the small size of the CAR rule-base, it was quite easy to construct test cases that would give complete coverage (satisfying All-edges). Five test cases were sufficient to satisfy Each-class and Each-hypoth, and an additional 3 cases provide complete coverage. (Certainly we could have satisfied the coverage measures with fewer test cases in which more findings were provided in each test case. However, we attempted to construct test cases that reflect situations that might realistically occur in the problem domain). Appendix D contains a script of the TRUBAC session, and Appendix E shows the test cases which were used. In running the dynamic analysis of the RHEUM and RH0184 rule-bases we found problems in both rule-bases due to the fact that there were sections of the rule-bases for specific disease classes which had not yet been completely implemented. These were eliminated when the rule-bases were edited to remove all problems identified during the static analysis phase, resulting in a rule-base which could handle those diseases which had been fully implemented. In addition, TRUBAC exposed an error in the RH0184 rule-base when the result generated by TRUBAC for a test case disagreed with the expected result. The result generated by EXPERT was incorrect and depended on the order in which certain information was presented to EXPERT. Since TRUBAC makes all possible connections based on the data presented, the result is free of order dependencies and it could expose this error.

4.5.2 PROLOG Rule-Bases

We also tested a rule-base which was written in a generic PROLOG-based rule language [Pre91]. The intention when this language was created was that it would serve not as
a language for writing expert systems, but instead as a language into which other rule-
base languages could be translated. Therefore it could potentially serve well as an
intermediate point between other rule-base languages and the language expected by
TRUBAC.

There are four types of basic expressions in the PROLOG-based language: rules, goal
declarations, askable declarations, and constraint declarations. Appendix F contains
the complete grammar for the language (from [Pre91]). Translation of rule-bases in the
PROLOG-based language into the language required by TRUBAC proceeds as follows.

Each line of the PROLOG version of the rule-base starts with a keyword that
identifies what kind of statement it is, and therefore determines how the statement
should be translated and what role it will have in the TRUBAC version of the rule-
base. Possible keywords are goal, rule, askable, and constraint. We consider the first
two of these in turn (the constraint declarations could be translated into semantic
checks at certain nodes in the DAG, but this is not done currently in TRUBAC).

**GOAL:** Goals in the Prolog rule-base serve a somewhat different function than
classes in the EXPERT rule-base. In an EXPERT rule-base we would have, for example,
several classes, each of which represents one possible disease with which a patient could
be diagnosed. An equivalent rule-base in the Prolog-based language would have one
goal statement, with disease listed as the goal. The particular classes of disease which
comprise the actual diagnostic categories can be determined from the rules where a
value is assigned to the goal. To create the TRUBAC version of the PROLOG rule-
base, we need to determine the possible instances of the goal term. Therefore, when
the goal statement is found, the term listed there is saved so that actual instances of
the goal can be determined subsequently.

**ASKABLE:** An askable item in the Prolog rule-base indicates an item, the value
of which is asked of the user while the rule-base is executing. The format of an askable
statement is `<item name> / <type> / <values>`. For example, the statement

```
askable nearfood / yesno / _
```

means that when the system asks for the value of fact nearfood it expects only a yes
or no response, and so no possible values must be indicated. On the other hand, the statement

\[
\text{askable loadtype / category / [t, s, c].}
\]

means that the variable loadtype can take on one of the three values specified (t, s, or c). We recall that at this point in the development of TRUBAC we restrict variables to either be Boolean valued or to take on a numeric value. In order to handle variables which can take on one of a number of distinct values we took the approach of creating a distinct Boolean valued variable representing each of the possible values of the corresponding Prolog variable. So for the above example the translation program would create three Boolean valued variables in the TRUBAC version of the rule-base: loadtype-t, loadtype-s, and loadtype-c. Each of these will be included in the list of findings that is part of the TRUBAC version of the rule-base.

This may seem to present an unnecessary increase in the complexity of the graph representation, since each new finding corresponds to a new node in the graph. However, as we will see in Chapter 7, the major contributor to the complexity of the system is the number of edges in the graph. If we have a node for each version of an askable item this does not introduce additional edges except for the edges from the source to the finding nodes. The total number of edges from finding nodes to their uses in antecedents will be the same as it would be if we had one node for the finding and put the value information on the edges to antecedents.

**RULE:** The format of a rule is

\[
\text{rule <rule id> :: <rule body>}
\]

Upon seeing a rule, indicated by the keyword rule, the first thing the translation program does is get past the keyword, the id, and the :: symbol. Then it translates the rule body. The rule body can appear in one of two equivalent forms: either

\[
<\text{consequent}> \text{ if } <\text{antecedent}>
\]

or
if_ <antecedent> then <consequent>

In either case the real work of the translation program is to process the antecedent and the consequent. The TRUBAC rule will be in an if-then form, where the antecedent of the TRUBAC rule will be derived from the antecedent of the Prolog rule, and similarly for the consequent. The grammar allows antecedents in disjunctive normal form, with negation only of simple expressions, not of complex expressions containing Boolean operators. The antecedent is copied largely as it appears in the Prolog version, with a few changes. Recalling the way we handle askable items, with the creation of new findings, a clause of the form \texttt{subsl hasValue absacryl} will become the identifier \texttt{subsl.absacryl}, which will be true on input to the TRUBAC version of the rule-base in those test cases for which \texttt{subsl} has the value \texttt{absacryl} on input to the Prolog version of the rule-base. A clause of the form \texttt{"amnt is less than 1.600"} will become the identifier \texttt{amnt.is-less-than-1.600}, while a clause of the form \texttt{"amnt hasValue 'less than 1.600"} will become the identifier \texttt{amnt.less-than-1.600}. All Boolean operators are copied as they are in the Prolog version of the rule.

A consequent in the Prolog version of the rule-base can be either a finding or an assignment which starts with a finding, such as \texttt{outcome hasValue t4004}. If the consequent consists of an assignment to a goal then we show just the assigned value as a class in the TRUBAC version of the rule. That is, if a Prolog rule has the consequent \texttt{"outcome hasValue t4004"} then the TRUBAC version of that rule will have just \texttt{t4004} as its consequent. If the consequent consists of a finding, then the finding will appear in the TRUBAC rule, with negation on the edge if the finding is negated in the Prolog version of the rule. Figures 4.14 and 4.15 show a rule in both its Prolog version and the TRUBAC version generated by the translation program.

Results of using TRUBAC on Prolog rule-bases

We translated one Prolog rule-base into the input language expected by TRUBAC. The rule-base contained 101 rules and makes suggestions of what kind of tape to use based on the environment in which the tape will be used. Translation into the TRUBAC format and subsequent graph construction results in 373 nodes, based on the 101 rules,
rule '022' ::

outcome hasValue t4008

if subs1 hasValue absacryl
and surf2 hasValue c
and 'amnt is less than 1 600'
and temp_ext hasValue no.

Figure 4.13: Prolog version of rule

if subs1-absacryl
and surf2_c
and amnt.is_less_than.1.600
and temp_ext_no then t4008

Figure 4.14: Corresponding TRUBAC version of rule

79 findings, and 21 classes. TRUBAC was then used to carry out both the static and dynamic analyses. Unfortunately, in this case we did not have available test data from the original rule-base developer or much information about the problem domain. Instead we actually used TRUBAC to guide construction of some test cases, thereby mimicking the steps a rule-base tester might actually follow when using TRUBAC to evaluate a rule-base in a situation in which there is little available test data.

During graph construction no conflicting or redundant rules were reported. The static analysis reported on 21 findings that were used in no rules. These unused findings actually are instances of the new findings which were constructed during the translation process. As such they do not indicate findings that were never used in the original rule-base, but rather indicate particular values of findings that are never used in the original rule-base. That is, if TRUBAC lists subs1-lam_chip as an unused finding, then we can conclude that no rule in the original rule-base used the value lam_chip of the finding subs1. After correcting the static problems by eliminating the unused findings, graph construction resulted in 352 nodes, based on 58 findings, while there were still 21 nodes and 101 rules.

Since we had little knowledge of the real meaning of the rule-base, we started by
creating one test case for each class, based on one rule that concludes each class. Because
of the way in which the test cases were constructed it turned out that all 21 classes
could be achieved with only 20 test cases. At that point we considered Each-hypoth,
which showed only 5 sub-classes which had never been concluded during consideration
of the first 20 test cases.

An additional 4 test cases allowed us to satisfy Each-hypoth. While these tests
were manually constructed by studying the structure of the rule-base, we also expect
them to give the correct result as well as enable us to satisfy Each-hypoth.

Unfortunately, the tapes rule-base is very flat, with few sub-classes (only 11 rules
have a consequent which is not one of the 21 classes). Therefore there are numerous ways
by which each rule can be concluded, and many finding-class combinations are possible.
Table 4.3 gives a sense of the magnitude of the problem, showing how many rules there
are which conclude each class and how many findings can possibly be involved in the
conclusion of each class.

After Each-hypoth has been satisfied, Each-class-every-finding still shows 178
unsatisfied finding-class combinations. This means that the execution paths that in-
clude these particular 178 finding-class combinations were not traversed during execu-
tion of the first 24 test cases. On closer examination we find that 58 of the 178
finding-class combinations involve the findings that lead to the few sub-classes that
are part of the system. For example, consider the rules shown in Figure 4.16. We
can see that each of the basic findings amnt_less_than_0.155, amnt_less_than_0.205,
etc. can lead to the sub-class amnt_is_less_than_1.600. Therefore, any class which
has amnt_is_less_than_1.600 on the execution path will also have on the execution path
all the basic findings which lead to the sub-class. It would appear that we could, in the
absence of any other test data, create test cases to cover these finding-class combina-
tions by using the one successful test case we have so far for each class and replacing
the basic finding used to conclude the sub-class. For example, our original test case
which led to the conclusion t4008 was

\begin{align*}
\text{subs1_absacryl} & \quad \text{T}
\end{align*}
<table>
<thead>
<tr>
<th>CLASS</th>
<th># of Rules</th>
<th># of Findings</th>
</tr>
</thead>
<tbody>
<tr>
<td>t4004</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>t4008</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>t4016</td>
<td>3</td>
<td>14</td>
</tr>
<tr>
<td>t4032</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>t4416</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>t467</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>t4912f</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>t4930</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>t4939</td>
<td>10</td>
<td>22</td>
</tr>
<tr>
<td>t4945</td>
<td>8</td>
<td>17</td>
</tr>
<tr>
<td>t4959</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>t4969</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>t584</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>t9308</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>t9469</td>
<td>7</td>
<td>24</td>
</tr>
<tr>
<td>t9482</td>
<td>10</td>
<td>24</td>
</tr>
<tr>
<td>t95c</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>t9528</td>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>t9529</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td>t96c</td>
<td>6</td>
<td>14</td>
</tr>
<tr>
<td>none</td>
<td>20</td>
<td>35</td>
</tr>
</tbody>
</table>

Table 4.3: Relationship of classes to findings and rules in Tapes rule-base.

if amnt_less_than_0.155
    then amnt_is_less_than_0.155
if amnt_less_than_0.205 or amnt_is_less_than_0.155
    then amnt_is_less_than_0.205
if amnt_less_than_0.305 or amnt_is_less_than_0.205
    then amnt_is_less_than_0.305
if amnt_less_than_0.350 or amnt_is_less_than_0.305
    then amnt_is_less_than_0.350
if amnt_less_than_0.550 or amnt_is_less_than_0.350
    then amnt_is_less_than_0.550
if amnt_less_than_1.600 or amnt_is_less_than_0.550
    then amnt_is_less_than_1.600

Figure 4.15: Rules from PROLOG based system
which will cover one finding-class combination for each of the 4 findings and the class \( t_{4008} \). We can use this same test case to satisfy additional finding-class combinations by simply replacing \( \text{amnt}_{<1.600} \) with each of the other basic findings which can also lead to the intermediate conclusion \( \text{amnt}_{<1.600} \). Alternately, it would seem that we could put all the findings under consideration into a single test case. However, without more detailed knowledge about the problem domain we have no way of knowing whether this would be in keeping with realistic situations or not. In devising test cases to cover the finding-class combinations which arise from the sub-classes, we may also cover additional finding-class combinations, since the initial test case for a class may not have included the sub-class in its antecedent at all. In creating a test case to cover rules with the sub-class in the antecedent we may also cover additional finding-class combinations reflecting the additional findings in the rule antecedent. For example, our original test case which concluded \( t_{4912f} \) was based on the simple rule

\[
\text{if subs1-other-rubbr and not amnt}_{<0.305} \\
\text{then } t_{4912f}
\]

and contained just the single finding \( \text{subs1-other-rubbr} \). In order to satisfy finding-class combinations which arise from the use of the sub-class \( \text{amnt}_{<1.600} \) we turn to the rule

\[
\text{if subs1-\text{absacryl} and surf2-c and amnt}_{<1.600} \\
\text{and temp-ext-yes and high}_{<92.5} \\
\text{then } t_{4912f}
\]

which will lead us to also satisfy finding-class combinations involving the class \( t_{4912f} \) and the findings \( \text{subs1-\text{absacryl}}, \text{surf2-c}, \text{temp-ext-yes} \) as well as all findings which lead to the sub-class \( \text{high}_{<92.5} \).
Our steps to devise test cases which would cover more of the finding-class combinations, focusing on those based on the sub-classes, left us with 124 finding-class combinations on execution paths untraversed by any test data. Of these, 24 are related to one of the sub-classes, while the other 100 combinations involve other findings. Of the 54 finding-class combinations we were able to cover, 33 of them were related to the sub-classes while the remaining 21 reflect additional findings found in the antecedents of the rules we used as the structural basis of the test cases. In a real testing situation, where test data was created in this way, all the case should then be reviewed by an expert to ensure that they make sense in the context of the problem domain, rather than satisfying coverage of an erroneous system.

4.6 Use of Heuristics for Test Data Selection

We demonstrate the heuristics for test data selection by applying them to one of the AI/RHEUM prototype systems, with 71 rules and 5 classes. We have available 127 test cases, which we will treat as the total case population, even though we know it to be incomplete. We begin our test case selection by picking one case for each of the 5 classes, PM, PSS, SLE, MCTD, and RA\(^3\). Executing the 5 initial test cases satisfies the Each-goal coverage measure but leaves 10 sub-classes which have not been concluded by any of the test cases (rd205, rx105, rd203, rd103, hcmp, neph, dnah, anem, myoss, myosm). As a next step we try to select test cases which will conclude these 10 sub-classes. For 4 sub-classes (rd205, dnah, anem, myosm) there are no test cases which actually conclude the sub-class as part of concluding a final class, though there may be cases which conclude the sub-class but do not subsequently use the sub-class to conclude a class. Of the remaining 6 sub-classes, for each of two sub-classes (hcmp, myoss) there is only one test case which will conclude it. Both of these test cases conclude multiple classes, but we must select them in order to get coverage of the sub-classes, and possibly get coverage of sub-class to class relations. The remaining sub-classes will be covered by the two test cases already selected or can be covered by

\(^3\)Polymyositis, Progressive Systemic Sclerosis, Systemic Lupus Erythematosus, Mixed Connective Tissue Disease, Rheumatoid Arthritis
<table>
<thead>
<tr>
<th>PM</th>
<th>PSS</th>
<th>SLE</th>
<th>MCTD</th>
<th>RA</th>
</tr>
</thead>
<tbody>
<tr>
<td>dnah</td>
<td>enash</td>
<td>enash</td>
<td>anem</td>
<td>enah</td>
</tr>
<tr>
<td>enah</td>
<td>myosm</td>
<td>rneh</td>
<td>enah</td>
<td>myosm</td>
</tr>
<tr>
<td>ex1pm</td>
<td>rnevh</td>
<td>enavh</td>
<td>myoss</td>
<td></td>
</tr>
<tr>
<td>rd205</td>
<td>myosm</td>
<td>rash</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rneh</td>
<td>rd102</td>
<td>rayes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rd102</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rr202</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.4: Unused sub-class to class relationships.

selecting two additional test cases, each of which concludes only a single class.

When we check **Each-hypoth** again after running the 4 additional test cases, we see that there are only 2 sub-classes (anem, myosm) which now have been concluded by no test case. This implies that there are cases which conclude some sub-classes but the sub-classes are not used subsequently to conclude a class. This becomes apparent when we examine the next criterion.

Checking the **Each-class-every-sub** criterion shows (Table 4.4) the sub-class to class relationships that have not been covered by the test data. We apply the third heuristic to the selection of test cases from the pool of available cases which will satisfy some of these sub-class to class relationships. In this situation the selection is quite easy, as for most of the sub-classes there are no test cases at all which use the sub-class to conclude a class, or the test case which does conclude the sub-class has already been executed and did not satisfy the particular relationship represented in the table. In fact, there are only two sub-classes, **ex1pm** and **rayes**, for which there are test cases which could be useful. For **ex1pm** there are two cases which involve it in leading to the class **PM**, one of which concludes 2 classes and the other of which concludes 3 classes. Given our interest in focusing on the discriminatory power of the system, we select the test case which concludes only 2 classes. For the **rayes-RA** relationship we use meta-knowledge about the system to determine that this is an unexecutable relationship. The sub-class **rayes** can be used to conclude the classes **MCTD** or **PM**, or it can be used to conclude an intermediate hypothesis, **ed101**, which serves to exclude **RA**. That is, if **rayes** is concluded and leads to the subsequent conclusion of **ed101** then **RA** will
never be concluded. Therefore, even though there is a structural connection between rayes and RA in the graph, the relationship between them will never be executed by any test case. Therefore we do not consider the rayes-RA relationship to be one that must be satisfied in order to obtain complete coverage of the system. Therefore, based on this step of data selection, we run one additional test case.

Next we evaluate the Each-class-every-finding criterion. TRUBAC reports that there are a number of unused finding to class relationships for each of the 5 classes: 12 for PM, 11 for PSS, 20 for SLE, 17 for MCTD, and 22 for RA. As stated above, the heuristic we would like to employ is to select cases which involve the least overlap between the sets of findings for each class. However, even if a test case concludes more than one class, it may not use the same precise subset of findings to make each conclusion. More likely is the scenario in which a test case concludes multiple classes but each class uses a subset of the case plus some findings which are used only to conclude that class.

In this particular system the overlap between these sets of unused findings is relatively low. The largest overlap is between PM and PSS, with 5 findings in common. However, for two of those findings there are no relevant test cases and for a third we have already run all relevant test cases. For each of the remaining two findings, the test case which uses the finding does not conclude either of these classes, so the finding to class relationship will remain unused for both classes. Focusing on the remainder of the findings for PM, there are 4 additional test cases that are suggested which should satisfy some of the finding to class relationships, while minimizing the number of cases which conclude multiple classes. After running these 4 new test cases we find that we have also eliminated 1 finding from the list for PSS and 2 findings from the list for RA. In both cases these findings were not unused by PM, but the construction of the test cases caused the finding to class relationships to be satisfied for PSS and RA.

Continuing to select test cases in this fashion, we select 1 case to take care of one more unused finding for PSS. When we consider cases which will satisfy findings for SLE we see that for two findings we have a choice between a case which concludes only SLE and a case which will also conclude another class. For another finding we must
choose the case which concludes two classes as it is the only relevant case. We opt to keep both the cases which conclude only SLE as well as the case which concludes two cases in order to test both the discriminatory aspect of the system as well as its reasoning in ambiguous cases.

For MCTD we select two cases which use relevant findings and conclude MCTD, though it turns out that they apparently do not use the findings to conclude MCTD and the particular finding to class relationships are left unresolved. Consideration of RA leads to the selection of 2 additional cases. At this point we see, by checking the number of unexecuted basis paths, that we have 103 unexecuted paths of 173 total paths. Were we to run all the available test cases we would still have 91 of 173 paths left unexecuted. This difference (103 versus 91) exists because the Each-class-every-sub and Each-class-every-finding criteria check that each sub-class or finding has been used along at least one path to each relevant class. In fact, it may be possible for the sub-classes or findings to reach a class along multiple paths. If there are test cases which exercise these additional paths then the path count will drop, although no new relationships are being tested.

We note that this large number of unexecuted paths is largely due to the high number of NOFM constructions (64) in the rule-base. Many of these are of the 1-of or 2-of form, representing a large number of alternative paths. Over 60 of the 91 unexecuted paths (over 66%) result from NOFM nodes for which numerous relevant findings are found in no test case.

Overall, we found that by using the heuristics to guide our test data selection, assuming sufficient meta-knowledge about the available pool of cases, we achieved a level of coverage that was almost equal to that obtained by running all the test cases. By running only 24 of 127 available test cases, we equaled coverage for the Each-class, Each-hypoth, Each-class-every-sub and Each-class-every-finding criteria, and almost equaled the total number of paths covered by the complete pool of cases. In evaluating the results of running the 321 available test cases on the Rh0184 system, we found that the test set did not cover all sub-classes, with 39 of 414 sub-classes never concluded by the test data. When we look more closely at the sub-class to class
<table>
<thead>
<tr>
<th>CLASS</th>
<th>% findings used</th>
<th>% subs used</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANKSP</td>
<td>75.00</td>
<td>88.24</td>
</tr>
<tr>
<td>BACT</td>
<td>61.90</td>
<td>85.19</td>
</tr>
<tr>
<td>CPDD</td>
<td>82.35</td>
<td>85.00</td>
</tr>
<tr>
<td>CTS</td>
<td>17.65</td>
<td>24.56</td>
</tr>
<tr>
<td>DJD</td>
<td>26.53</td>
<td>18.75</td>
</tr>
<tr>
<td>ENTAR</td>
<td>53.33</td>
<td>70.59</td>
</tr>
<tr>
<td>FIB</td>
<td>3.61</td>
<td>2.47</td>
</tr>
<tr>
<td>GCART</td>
<td>60.00</td>
<td>79.31</td>
</tr>
<tr>
<td>GCR</td>
<td>8.43</td>
<td>5.63</td>
</tr>
<tr>
<td>JR1</td>
<td>7.74</td>
<td>10.19</td>
</tr>
<tr>
<td>JR2</td>
<td>41.27</td>
<td>61.76</td>
</tr>
<tr>
<td>JR3</td>
<td>12.94</td>
<td>22.39</td>
</tr>
<tr>
<td>MCTD</td>
<td>88.46</td>
<td>95.00</td>
</tr>
<tr>
<td>PAN</td>
<td>46.00</td>
<td>60.00</td>
</tr>
<tr>
<td>PM</td>
<td>53.33</td>
<td>66.67</td>
</tr>
<tr>
<td>PMR</td>
<td>13.24</td>
<td>13.64</td>
</tr>
<tr>
<td>PRAYN</td>
<td>0.82</td>
<td>2.86</td>
</tr>
<tr>
<td>PSART</td>
<td>22.22</td>
<td>66.67</td>
</tr>
<tr>
<td>PSS</td>
<td>72.22</td>
<td>83.33</td>
</tr>
<tr>
<td>RA</td>
<td>41.27</td>
<td>57.14</td>
</tr>
<tr>
<td>REIT</td>
<td>73.17</td>
<td>85.19</td>
</tr>
<tr>
<td>RHF</td>
<td>7.78</td>
<td>11.36</td>
</tr>
<tr>
<td>SJ</td>
<td>100.00</td>
<td>100.00</td>
</tr>
<tr>
<td>SLE</td>
<td>78.26</td>
<td>66.67</td>
</tr>
<tr>
<td>TBART</td>
<td>58.14</td>
<td>72.73</td>
</tr>
</tbody>
</table>

Table 4.5: Use of findings and sub-classes to conclude classes.

relationships we see that the test set has not done particularly well at all, as evidenced by Table 4.5. This table shows the percentage of both findings and sub-classes which can lead to each class that were actually involved in inferences resulting from the test data. We can see that, for a majority of the classes, the test data has covered very few of the ways in which the class can be concluded. If more data were available, or if we were able to request additional data from doctors involved in rheumatology, we would use the heuristics to select cases for those classes for which few sub-class to class relationships had been tested, as well as any finding to class or sub-class to class relationships that we knew must be handled correctly by the system because of the serious nature of an incorrect classification for those cases. If we have no additional
test data then we would discount the results from the rule-base in actual cases which were classified as belonging to one of the relatively untested classes such as FIB or GCR, unless we knew, based on our meta-knowledge about both the system and test data, that the actual case corresponded to the tested section of the rule-base.

4.7 Example of Rule-Base Pruning

In rule-base pruning we would like to remove from the rule-base rules which are not necessary for the rule-base to perform correctly. In general, we expect that a smaller rule-base will run more efficiently. We have already mentioned (Chapter 3) work that has been done [Ind91] using a Monte-Carlo simulation approach to determine which of a number of randomly pruned rule-bases most closely functions as the original did. In the context of rule-base coverage we can use the coverage measures to identify sections of the rule-base that are candidates for pruning.

A first phase in the pruning process can actually be carried out based on the static analysis process. Referring back to Tables 4.1 and 4.2, we can see that the static analysis information alone allowed us to remove 5 rules from the RHEUM rule-base (6% of the rule-base) as well as 85 findings and 3 classes. Since TRUBAC is graph based, removing the findings and classes significantly reduces the size of the graph, since we eliminate one node for each finding and each class. The total reduction in the graph size is 92 nodes, or 27%. Similarly, for the RH0184 rule-base we removed 449 findings, 2 classes, and 29 rules, for a 3% reduction in the rule-base and an overall 20% reduction in the number of nodes in the graph.

We can follow this initial pruning by further pruning based on the information provided by the coverage measures. For example, we consider the RHEUM rule-base, which contains 71 rules after correction of all the problems identified by the static analysis. As a first step, we run all available test cases through the system and evaluate the All-edges coverage criterion. TRUBAC identifies for us each rule that is the source of a graph edge that has not been used during execution of the test data. While a rule may be the source of an unexecuted edge, this does not directly imply that the rule
<table>
<thead>
<tr>
<th>Stage</th>
<th>FINDINGS</th>
<th>CLASSES</th>
<th>RULES</th>
<th>GRAPH NODES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>151</td>
<td>8</td>
<td>76</td>
<td>333</td>
</tr>
<tr>
<td>Static</td>
<td>66</td>
<td>5</td>
<td>71</td>
<td>241</td>
</tr>
<tr>
<td>1st Prune</td>
<td>61</td>
<td>5</td>
<td>52</td>
<td>204</td>
</tr>
<tr>
<td>2nd Prune</td>
<td>60</td>
<td>5</td>
<td>50</td>
<td>197</td>
</tr>
</tbody>
</table>

Table 4.6: Reduction in size of RHEUM rule-base.

...itself is unnecessary. Consider, for example, that we have a rule which involves a 7-of-10 clause. If we have a test case in which 7 of the 10 findings or sub-classes are true then the rule consequent will be concluded. However, there will be 3 unexecuted edges, 1 for each of the 3 findings or sub-classes that were not true. We would not want to prune this rule from the rule-base.

For the RHEUM rule-base, let us assume that the available test cases are a comprehensive set of test cases, covering all types of cases which could be encountered in the actual diagnostic setting. Under this assumption, any uncovered section of the rule-base is indicative of either an error in the rule-base or unnecessary rules which can be pruned. So we begin the next step of the pruning process by examining each rule which is identified as leading to some untraversed edge in the graph. Of the 72 rules, TRUBAC indicates that 52 rules involve one or more untraversed edges. On further manual examination, we remove any of these rules that do not have an NOFM clause, allowing the immediate removal of 15 rules.

This prune step may lead to new static problems, which we can identify by running the static analysis portion of TRUBAC on the pruned rule-base. This leads us to the further removal of 3 rules, 5 findings and 11 components of rule antecedents. We also can eliminate 2 NOFMs because they each became 1-of-1 constructions, which can be simplified to an antecedent with a single component. At this point the rule-base contains 52 rules and we can iterate the pruning process, removing an additional 2 rules, 1 finding, and 4 antecedent components. Table 4.6 summarizes the changes made to the RHEUM rule-base as a result of both the first static analysis phase and the subsequent pruning steps. We note that, even though we did not focus our pruning solely on the unexecuted paths but looked more broadly at all untraversed edges, we do
see an improvement in coverage. If we run all available test data through the system before pruning (but after static analysis) there are 91 unexecuted paths out of a total of 173 paths (52.60%). After the pruning there are 76 unexecuted paths out of a total of 156 (48.70%). Overall performance, in terms of results for the 127 test cases, is unchanged by the pruning steps.

The Monte-Carlo based pruning in [Ind91] actually generates a rule-base of 26 rules for RHEUM. This is accomplished by automatically carrying out random pruning steps, removing both rules and antecedent components, and comparing the performance of the pruned rule-base to the performance of the original. While the Monte-Carlo based method certainly produces a better prune, it is really only useful in a situation in which the test suite is considered a comprehensive collection of possible types of test cases. Were we not to make this assumption then it is critical that we have some oversight of the deletions that are made during the pruning process.

Realistically, we should assume that the test suite may not be adequate, in which case there are numerous points in the pruning process at which we would consult with the expert prior to actually pruning a rule. For example, if a rule antecedent has a conjunct which is never concluded by any rule and is therefore always false, then it would seem that the rule could be pruned since it would never conclude its consequent. In actuality, consulting the expert would be indicated in such a situation as it could be that there was an error, either in the construction of the rule or in the construction of rules that should lead to the conjunct in question.

After the pruning steps outlined above it would be possible to continue to prune based on information that TRUBAC provides about sub-class to class and finding-class combinations that have never been executed. For example, if the finding coma is never used in a test case for *Systemic Lupus Eryth.* then we can prune the finding out of the appropriate antecedents. This is a more tedious approach, but it will eventually lead to the direct deletion of some rules. A rule might also be pruned because the removal of antecedent components caused it to become redundant with another rule in the rule-base.
4.8 Evaluation of Random Test Selection

Using TRUBAC we are able to evaluate how well a complete test suite covers a rule-base. In addition, by reporting on the number of unexecuted paths remaining after each test case is run, we can see which test cases actually contribute to coverage and which cases duplicate execution paths which have been exercised by a previously executed test case. In general we would expect there to be overlap among the test cases, particularly for those portions of the system that handle the more common situations, since it is much easier to get test data representative of those situations. Of the available data for the RHEUM rule-base only 23 of the 127 test cases, or 18%, are necessary to attain the coverage level of 47.40% (82 of 173 paths) we reach by running all 127 cases.

With such a low number of cases actually covering new execution paths (i.e. reducing the number of unexecuted paths), and the remainder of the cases duplicating already covered paths, we would expect that randomly selecting cases out of the available test suite should also be a good testing strategy. In general, intuition would lead us to believe that increasing the number of test cases we use should improve coverage. In fact, we can use TRUBAC to determine if this is in fact the case. We set up a program which would, for each class in the rule-base, randomly select a test case for that class which had not already been run during the testing process. We specified how many unique test cases we wanted to select for each class, as well as how many times we wanted to run that number of test cases in order to get an average value for the coverage attained. Note that, while we never duplicate test cases for a single class, a specific test case which concludes more than one class may be executed more than once as it may be selected for each of the classes which it can conclude.

We can see from Table 4.7 precisely how many of the 127 available test cases do conclude each class in the RHEUM rule-base. If the number of test cases that we want to randomly select and run for each class is greater than the number of cases available for that class then we will run all available cases for that class. Knowing that we can get maximum coverage possible with 23 test cases, we would expect to get reasonably close to this coverage figure by selecting 10 cases per class, which should give us close
Table 4.7: Number of test cases which conclude each class.

<table>
<thead>
<tr>
<th>CLASS</th>
<th># of Test Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA</td>
<td>82</td>
</tr>
<tr>
<td>PSS</td>
<td>36</td>
</tr>
<tr>
<td>SLE</td>
<td>30</td>
</tr>
<tr>
<td>PM</td>
<td>8</td>
</tr>
<tr>
<td>MCTD</td>
<td>29</td>
</tr>
</tbody>
</table>

to (though not necessarily exactly) 50 test cases total. In fact, when we tested the system with 10 cases per class we were left with an average (over 10 runs with 10 cases) of 108.90 unexecuted paths, almost 20% more than the number we can get in the best case. Table 4.8 shows the number of unexecuted paths remaining after running the program with increasing numbers of randomly selected test cases for each class, and the average number of unique test cases used. We should note that while selecting 30 test cases for each node guarantees the maximum coverage possible for the three classes SLE, PM, and MCTD, it only approaches the best overall coverage possible. It is also important to note that while the coverage improves over large steps as we increase the number of test cases per class, coverage does not always improve when we make a single increment in the number of test cases. For example, the coverage achieved was better when we used 15 cases per class than when we increased to 16 cases per class.

While the final coverage figure obtained after running 30 cases per class is nearly as good as we can get (92.80 unexecuted paths versus 91), we had to run over 80 unique test cases to get that result. Furthermore, in using random selection of test cases we had to run over 50 test cases to obtain the same degree of coverage (103 unexecuted paths remaining) that we obtained with just 24 test cases when we applied the heuristics for test data selection. This shows that randomly selecting test cases does not assure complete or maximum possible coverage, even if the selection is spread across the classes of the system.
<table>
<thead>
<tr>
<th># of Test Cases Per Class</th>
<th>Unexecuted Paths Remaining</th>
<th>Ave # Unique Test Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>108.90</td>
<td>40.60</td>
</tr>
<tr>
<td>11</td>
<td>111.10</td>
<td>43.70</td>
</tr>
<tr>
<td>12</td>
<td>107.70</td>
<td>46.40</td>
</tr>
<tr>
<td>13</td>
<td>105.10</td>
<td>47.60</td>
</tr>
<tr>
<td>14</td>
<td>102.60</td>
<td>50.70</td>
</tr>
<tr>
<td>15</td>
<td>102.50</td>
<td>53.20</td>
</tr>
<tr>
<td>16</td>
<td>103.00</td>
<td>53.30</td>
</tr>
<tr>
<td>17</td>
<td>101.00</td>
<td>57.10</td>
</tr>
<tr>
<td>18</td>
<td>101.00</td>
<td>60.70</td>
</tr>
<tr>
<td>19</td>
<td>99.90</td>
<td>61.40</td>
</tr>
<tr>
<td>20</td>
<td>99.90</td>
<td>65.30</td>
</tr>
<tr>
<td>21</td>
<td>96.10</td>
<td>66.90</td>
</tr>
<tr>
<td>22</td>
<td>96.60</td>
<td>69.20</td>
</tr>
<tr>
<td>23</td>
<td>96.80</td>
<td>70.60</td>
</tr>
<tr>
<td>24</td>
<td>95.80</td>
<td>72.30</td>
</tr>
<tr>
<td>25</td>
<td>95.20</td>
<td>72.70</td>
</tr>
<tr>
<td>26</td>
<td>94.60</td>
<td>75.60</td>
</tr>
<tr>
<td>27</td>
<td>93.20</td>
<td>78.00</td>
</tr>
<tr>
<td>28</td>
<td>92.80</td>
<td>80.80</td>
</tr>
<tr>
<td>29</td>
<td>93.10</td>
<td>79.30</td>
</tr>
<tr>
<td>30</td>
<td>92.80</td>
<td>82.90</td>
</tr>
</tbody>
</table>

Table 4.8: Coverage results after random test case selection.
4.9 Discussion of TRUBAC as a V&V Method

Discussion among verification and validation researchers of different V&V approaches and tools has led to the preliminary development of criteria for objective comparison of tool functionality ([GKLP92]).

While TRUBAC clearly provides features not present in many other tools, most notably the ability to do both verification and validation in a single tool, it can be compared to other tools that carry out either verification or validation. The functions commonly found among verification and validation tools are:

- ability to check knowledge base logic for anomalies and errors (redundancy, conflict, circular inference chains)
- ability to check semantics of knowledge base (for example, for type conflicts and violated semantic constraints)
- ability to find missing knowledge and other instances of incompleteness in the knowledge base
- ability to generate test cases based upon the structure and function of the knowledge base

TRUBAC can carry out many of the functions listed above. Certainly it is able to find redundancy, conflict, and cycles in a rule-base. It is also able to generate test cases based on the structure of the rule-base, a feature which is actually available to the user in conjunction with evaluation of the coverage measures. TRUBAC is not set up to be able to check for semantic errors in the knowledge base because semantic errors must be determined according to the language in which the rule-base was developed originally. Since TRUBAC is designed to work with a generic rule-base input format, a check for semantic errors in the original rule-base would have to be done in the translation programs and not in TRUBAC itself. While TRUBAC does not find missing rules, it can identify other kinds of incompleteness, such as isolated rules, dangling conditions and useless conclusions.
Finally, while the capabilities of TRUBAC may seem unnecessary when one looks at a small rule-base, with a large rule-base it is impossible for the user to view the entire rule-base at once in order to identify static problems. Lastly, the dynamic analysis portion of TRUBAC allows the user to determine whether the rule-base “gives the right answer”, while at the same time testing the overall structure of the rule-base. By using coverage measures TRUBAC serves as an objective judge of the quality of the testing that the user carries out.
Chapter 5

Related Work

This chapter presents a discussion of related work in the area of VV&T, explaining the types of analyses that are carried out by other VV&T methods, as well as comparing our approach to some of the other VV&T methods discussed.

The methods discussed are grouped into three categories. Verification, or static, methods are discussed first, followed by discussion of validation (dynamic) methods which are not graph-based. A number of the dynamic methods discussed are, in reality, knowledge refinement tools whereby the user is directed to changes that can be made to improve the functional behavior of the expert system. The information provided to the user by our approach can be used to guide a knowledge refinement and test cycle. In attempting to satisfy a coverage measure, the user may find errors in the rule-base. After correcting the errors, our method can be used again with the revised rule-base and the original set of test cases in order to determine if the original error was fixed and no new errors introduced. (See also the discussion in Chapter 7 on the use of rule-base coverage testing during incremental rule-base development). The chapter closes with a particular discussion of research which incorporates either a graph-based representation or an analysis based on execution paths and a comparison of certain of the graph-based methods to our approach.

This division of existing tools facilitates comparison to our method, since we use separate static and dynamic phases which can be run at distinct times by the system tester. There is no discussion of metrics as applied to expert systems since the majority of such approaches seem to focus only on static metrics (an exception is found in [Mes91], which we discuss in the section on graph-based methods).
5.1 Verification Methods

One of the earliest static rule checking programs was developed as part of the ONCOCIN system\(^1\) [SSS82]. ONCOCIN's verification capabilities are divided into checking for inconsistencies (defined as conflict, redundancy, and subsumption) and incompleteness (defined as missing rules). An automated rule checker is used which displays potential errors, allowing the expert to make the ultimate decision about what are actual errors. ONCOCIN examines the rule set to determine which condition parameters are used to conclude a given action parameter (equivalent to the "path-expression" approach of KB-Reducer [Gin87, Gin88a] which is also equivalent to collecting antecedent information during traversal of a path to a class in our approach). These parameter combinations are put into a table and used to determine conflicts, redundancies, and missing rules. (We achieve the same effect by storing antecedent and consequents in a table as the graph is constructed and doing a look-up as each new antecedent/consequent combination is encountered). Missing rules are identified when missing parameter combinations are determined. A weakness of the ONCOCIN approach is that it is limited to identifying problems at the rule level. That is, it cannot identify problems that are the result of longer reasoning chains, such as redundancies that are the result of several inference steps. Furthermore, it cannot determine if all the rules it tests are actually reachable in the context of the entire system. Finally, the overall complexity is \(C(2^n)\), where \(n\) is the number of distinct condition parameter values (found in the antecedents of the rules), so it is only really useful if the set of parameters is small ([Pre89]).

Another example of static analysis of rule-based systems is CHECK, which was developed [NPLP85, NPLP87] to verify the consistency and completeness of knowledge based systems (without uncertainty) developed using LES, the Lockheed Expert Systems development environment. The algorithms used in CHECK could, presumably, be applied to rule-bases developed in other environments, but the CHECK program itself could not be. Operation of CHECK is based on the construction of a dependency chart,

\(^1\)ONCOCIN is a rule-based consultation system designed to advise doctors on management of patients on experimental treatment programs.
which shows the dependencies among rules and between rules and classes. CHECK identifies the following rule-base problems (or potential problems) consistency checks (conflict, redundancy, subsumption, unnecessary IF conditions, circular rule chains); completeness checks (unreferenced attribute values, illegal attribute values, unreachable conclusions, dead-end goals, and dead-end IF conditions). For identification of cycles in the rule base, CHECK uses a cyclic graph detection algorithm based on an adjacency matrix graph implementation. For identification of other problems in the rule-base, CHECK iterates over rules and clauses within rules. While the complexity is not given in the papers on CHECK, it appears to be \( \mathcal{O}(nk^2) \), where \( k \) is the maximum number of clauses in any rule in the knowledge base. While this complexity is better than that of ONCOCIN, the tool is not one for general use since it depends on the rules being set up in a particular fashion.

Finally, we consider ESC (Expert System Checker) [CS87] which uses decision-tables to check the completeness and consistency of a knowledge base. ESC looks for discrepancies, ambiguities (including conflict), redundancies, and missing rules. There is an automated mapping of rules to decision tables, with the conditions and consequents (actions) forming the table rows and the rule numbers forming the columns. Conflicts and redundancies can be determined in quadratic time, followed by a completeness check. If there were no conflicts or redundancies then the completeness check can be carried out in linear time. However, if conflicts or redundancies were found then the completeness check requires exponential time \( \mathcal{O}(2^n) \) where \( n \) is the number of rules. Note that the rules are divided into subtables, and the check is therefore performed several times, but over a subset of the rules each time).

### 5.2 Validation and Dynamic Analysis Methods

Perhaps the earliest dynamic analysis method was TEIRESIAS [Dav84], developed for use with the MYCIN system. TEIRESIAS allows the rule-base developer to find errors in the rule-base that had led to incorrect conclusions. TEIRESIAS is in reality both a debugging tool and a knowledge acquisition tool, allowing the alteration, deletion or addition of rules in order to fix an error. TEIRESIAS shows the user the reasoning
that was used by the system to reach a conclusion, and the user can then approve of the rules used or indicate that there was an error and make corrections.

There are two aspects of TEIRESIAS which diminish its utility for expert systems developers. First, using TEIRESIAS is an iterative process. Upon discovery of an error during a consultation, TEIRESIAS helps the user build new rules to fix the error, but it does not then automatically test the impact of the new rules on test cases which have already run successfully. The user has to rerun those test cases to ensure that the changes did not create new errors and did, in fact, fix the old errors. Secondly, effective use of TEIRESIAS depends on having a knowledge base tester who is sufficiently expert in the problem domain to know that there is an error in the reasoning used to reach a conclusion. This can limit its effectiveness in the usual expert systems development environment in which the expert provides information to the systems developer but is not necessarily actively engaged in the development and testing process.

The development of TEIRESIAS was followed by EMYCIN [vMSB85], which fixes spelling errors, checks that rules are semantically and syntactically correct, and points out interactions among rules that could lead to errors. EMYCIN uses a trace of the system's reasoning process, an interactive mechanism for reviewing and correcting the system's conclusions (like TEIRESIAS), and a facility that compares the system's results with stored correct results for the test cases.

There are also a number of dynamic analysis tools which carry out (or assist) knowledge base refinement. Among these are SEEK [PW84] and SEEK2 [GWP85, Gin88b]. SEEK is an interactive rule refinement system which uses stored cases to provide guidance for rule modifications. SEEK makes suggestions of possible rule generalizations and specializations. A generalization weakens a rule so that the antecedent will be satisfied in additional situations, whereas rule specialization strengthens the rule so that the antecedent will be satisfied in fewer situations than originally. The user then decides which refinements to try out and which to incorporate into the rule-base. SEEK is designed to work with rule bases that use a criteria table format as the rule representation. A criteria table is presented for each conclusion, containing major and minor findings that are significant for reaching the conclusion, along with the rules used to reach the
conclusion. As each test case is run, SEEK matches the expert's conclusion for that case with the system's conclusion, creating a performance summary. Once the performance summary is complete the user can refine the rules, weakening or strengthening rules with the guidance of heuristics built into SEEK.

SEEK2 represents an advance beyond the work done in SEEK in a number of ways. SEEK2 is not limited to knowledge bases that can be converted into criteria tables, but rather can work with any rule base written in the EXPERT language (see section 3.5.1). Furthermore, SEEK2 automates many of the knowledge base refinement tasks which are left to the user in SEEK. For example, SEEK2 automatically decides for which conclusions rules should be evaluated. It also decides which rule refinements to try, and which of those to keep in the final knowledge base as indicated by improvements when the refined knowledge base is used to evaluate the test cases.

For both SEEK and SEEK2 the quality of the refinements produced will be determined by the breadth of test cases used. There is no aspect of these systems that guides the initial selection of the test cases, or judges how well the test cases cover the range and domain of the system being evaluated. SEEK and SEEK2 both base their analysis on the functional performance of the rule-base, though they then suggest structural modifications.

5.3 Graph or Network Based Methods

5.3.1 CASNET

Graph-based methods for rule-base analysis involve the creation of a graph representation of the rules. As an early example of such a representation (though not specifically used for verification and validation), we consider the Causal-Associational Network (CASNET) for glaucoma diagnosis [WKAS78]. In this model, states (or nodes) correspond to observations of a patient. The edges of the graph represent causal links or causal relationships between different states. Each complete path through the network (from the starting node to a terminal node) is the representation of a particular disease.
process. A partial path, from the starting node to some nonterminal node, is the representation of some degree of development within the disease process, with increasing seriousness of the disease as the path progresses.

There are some differences between the graph structure we have proposed and that of CASNET. The source of these differences lies in the roots of our graph in the rule base of the expert system. For CASNET the network serves as a representation of the knowledge, whereas we are using a graph which is a representation of the rule-base, while the rule-base is the initial representation of the knowledge. As a result we see that interior nodes in CASNET networks represent intermediate stages of disease progression. However, in our graph an interior node will represent such an intermediate disease stage (assuming for the moment a medical application) only if the rules were originally developed in such a way that the intermediate disease stage is also an intermediate hypothesis of some rule in the rule-base. That is, if the knowledge engineer does not use intermediate disease stages as part of the information organization for the problem domain, then those stages will not appear as nodes in our graph.

Our approach is similar to CASNET in that the complete graph is a static structure, as is the initial state network in CASNET. In CASNET a sub-net of interest is delineated dynamically based on a series of observations, much as we consider the sub-DAGs (execution paths) which result from particular inference chains.

5.3.2 KB-REDUCER

We include discussion of KB-Reducer [Gin87, Gin88a] in this section because of the implied network of rules which KB-Reducer employs to do rule-base analysis. KB-Reducer is a verification tool which can check rule-bases for inconsistency and redundancy, including contradictions and redundancies that are the result of inference chains, not just pairs of rules.

The process of knowledge base reduction involves calculation of all possible logically independent and minimal sets of inputs under which the knowledge base will conclude each assertion (each class). In order for the reduction process to work, the rules of the knowledge base must form an acyclic network under the depends-on relation, defined as
follows ([Gin87]): a rule \( r \) depends-on a rule \( r' \) iff \( r' \) asserts a literal \( l \) in its consequent, such that either \( l \) or its negation appears in the antecedent of \( r \). That is, \( r' \) must fire in order for \( r \) to be able to fire. KB-Reducer computes the transitive closure of the \( \text{depends-on} \) relation. If the pair \( < r, r' > \) is not in the transitive closure, then the network is acyclic.

If the network is acyclic, then KB-Reducer proceeds to label each hypothesis \( H \) with the set of environments that lead to the assertion of \( H \), where an environment is itself a set of findings. (This labeling is the same as the logical expression formed by following paths through the induced AND-OR graph to the node representing the assertion of the hypothesis.) KB-Reducer carries out the labeling process on the rules in an order such that no rule is processed before any rules on which it depends. As each rule is processed, KB-Reducer updates the partial label for the hypotheses the rule asserts, and checks for redundancy and contradiction.

The chief cost of KB-Reducer is in the computation and update of the labels. In the worst case, the cost is exponential in \( ld \), where \( l \) is the length of the longest chain of \( \text{depends-on} \) relations, and \( d \) is the average number of rules on which an arbitrary rule depends. However, in practice this worst case is often not found.

KBR3 ([DW93]) is a system based on KB-Reducer that has been developed to assist in the maintenance of large knowledge bases. KBR3 is based on an "application-neutral" language into which the knowledge base must be translated before it can be processed. As with KB-Reducer, the label of each rule conclusion must be computed as part of the testing of the system. Operation of KBR3 is computationally prohibitive on large knowledge bases, as the complexity is a function of the amount of rule interconnectivity, the length of inference chains, the number of explicitly stated values, and which values are referenced in rule conclusions. Therefore usually other kinds of testing are carried out on the rule-base before KBR3 is run, since it is so time consuming.

5.3.3 COVER

COVER (COmpleteness VERifier) for verification of rule-based systems ([Pre89, PS92, GPSS93]) is a tool which carries out the following seven verification checks: redundancy.
conflict, subsumption, unsatisfiable conditions (rules which cannot be fired, missing values), dead-end rules, circularity and missing rules. The rules must either be written in or converted to a language based on first-order logic, and COVER must be given the set of final hypotheses (classes), as well as information about any semantic constraints.

The first phase of testing involves checking single rules in order to find unsatisfiable conditions and dead-end rules (linear in the number of rules ([PS92])). Then redundancy, conflict, and subsumption among rule pairs are detected by symbolic comparison of pairs of rules (quadratic in the number of rules). While these checks are carried out a dependency graph of the rules is constructed, in which each node corresponds to a rule and edges connect rules (nodes) which reference one another. That is, the node for a rule \( R \) is connected to the nodes for all rules which use the consequent of \( R \) in their antecedent (equivalent to Ginsberg’s depends on relation ([Gin87])). A cyclic graph detection algorithm is used on the dependency graph in order to detect cyclic inference chains. Finally, inference chains are traced to find more general cases of ambivalence, circularity, redundancy, and missing rules. The complexity of this final step is \( O(b^d) \), where \( b \) is the average number of literals in a rule, and \( d \) is the average length of a reasoning chain. Note that if the inference chains tend to be short (which seems to be common in realistic expert knowledge bases) then exhaustive checking is feasible.

The representation we have chosen for this work is of a finer granularity than that used in COVER. In COVER the nodes of the graphs represent rules, and the edges represent dependencies or relations between rules, whereas in the DAG representation we have proposed here each rule is itself represented by a small sub-graph, with nodes for each element of the antecedent, each logical operator in the antecedent, and the consequent.

5.3.4 Pr/T Nets

The graph representation we are using is closer in some ways to the Pr/T net representation of a rule-base than it is to any of the other graph-based methods. In the majority of the graph-based methods a rule from the rule-base is equivalent to a node in the graph, while in our graph nodes correspond to individual findings or hypotheses,
not to entire rules. Similarly, in the Pr/T net representation [ZN89] the level of detail is
the findings and hypotheses, not the rules. Pr/T nets are made up of places (predicates)
and transitions. With respect to a rule-base, we can think of a place as corresponding
to a finding or sub-class, and a transition as corresponding to the inference relation of
an antecedent causing a hypothesis. Therefore, a transition corresponds to the relation
between the antecedent components and the consequent(s) of the rule.

The work carried out by Zhang and Nguyen ([ZN89]) involves static analysis of a
rule base, as follows. First a Pr/T net is constructed from the rule-base, including a
transition for each rule, and places for each antecedent and consequent. Then there
is an attempt to find various sub-net patterns in the Pr/T net, where the sub-net
patterns are indicative of inconsistency and incompleteness in the net (and therefore in
the rule-base). If matches for these patterns are found, then the particular sections of
the rule-base are studied to see if, in fact, they involve an actual error in the rule-base.
The conditions that can be found by this method are redundant rule pairs, subsumed
rules, circular rules, conflicting rule pairs, dangling conditions, useless conditions, and
isolated rules. However, this work does not take certainty factors into account, and it
also does not handle negated information in the rule-base.

5.3.5 Hypergraph representation

Valiente [Val93] presents a method for redundancy and subsumption detection which is
based on a hypergraph representation of the rule-base. His premise is that hypergraphs
provide a more compact representation for the rule-base than do graphs, and they
also facilitate use of graph transformations based on graph grammars and algebraic
graph transformation. His basic process involves translating the knowledge base into
its hypergraph representation, detecting and removing all redundant and subsumed
rules by use of graph productions, and then translating the resulting hypergraph back
into a knowledge base. His method is restricted to antecedents with just conjunctions,
and any disjunction in the rules will have to be rewritten using multiple rules.
5.3.6 Path Hunter/Path Tracer

The VV&T approach based on the execution path model, incorporated in Path Hunter and Path Tracer [GPC+93, PGCR93], shares the fundamental premise upon which our work is based: functional validation may show that the system performs well on the test cases, but there may still be problems in portions of the rule base that were never exercised during testing. The goal of Path Hunter/Path Tracer is the selection of a set of test cases that exercise the structural components of the rule-base as exhaustively as possible. This involves firing all rules, and also firing every “causal sequence” of rules. The model used to identify all possible dynamic causal rule firing sequences is the rule execution path (equivalent to a sub-DAG in our representation).

Path Hunter uses structural path analysis to detect potential interactions between rules in a rule-base and to identify problems within the rule-base. Path Hunter generates paths, based on chains of inter-dependent rules, where the definition of rule dependency is essentially the same as the depends-on relation used in KB-Reducer. This is basically a path enumeration task, which could be combinatorial in nature. The complexity is controlled by precomputing the logical completion for each subproblem and by the use of equivalence classes of rules, formed by collecting redundant rules together into one class, which reduces the number of paths that Path Hunter must generate. (In our approach the step of explicit identification of redundant and ambiguous rules is unnecessary, as they will be identified through the graph construction process. Therefore, while there may be redundant rules in the rule-base, there will be no duplication of those rules within our graph representation). Through the process of applying Path Hunter to a rule-base, various rule-base anomalies can be identified, such as isolated rules and ambiguous rules.

Path Tracer is a tool for structural rule-base testing, using the paths generated by Path Hunter, in conjunction with traces of dynamic rule firings, to determine how extensively the possible execution paths are covered by the test data. The basic testing steps are
• run Path Hunter, along with specifications of the logical completions, to enumerate all execution paths.

• run test cases through the rule-base, generating trace files for each test case (presumes a rule-base shell tool with a trace capability).

• run Path Tracer to analyze the trace files and generate information about path coverage.

This final step is rather complex, since it requires finding a mapping between the abstract paths, those generated by Path Hunter, and the concrete paths, those observed at run time. This is a nontrivial task, which in turn involves finding a mapping of the concrete rule firings to the abstract rules and then using asserted sub-classes to trace causal relations among rule firings.

In actual use of Path Hunter and Path Tracer [PGCR93] a set of test cases derived from a functional analysis of the system were used as the test data. While the rule-base under analysis performed very well, when compared with human experts, in functional tests the structural analysis was able to discover that a significant portion of the rule-base was not exercised during testing, reinforcing the argument that functional testing alone is not an adequate approach to testing rule-based systems.

While there are a number of similarities in the premises which underly our approach and Path Hunter/Path Tracer, there are also significant differences between the two approaches. First, the graph we construct models the rule-base at the level of findings and hypotheses, rather than at the rule level as is done in the Path Hunter representation. While this may make the graph somewhat larger, it allows us to carry out both verification and validation with one representation. This is in contrast to COVER, which requires two representations, the first-order logic translation of the rules and the dependency graph, in order to carry out verification alone.

Second, the methods by which rule-base coverage are determined or measured are quite different. Path Tracer does its assessment of path coverage based on the number of causal dependencies observed in the trace file after the test data is run. First an attempt is made to map the concrete paths observed at run time to the abstract paths...
generated by Path Hunter. However, at times it is not possible to map a concrete rule firing unequivocally to a single abstract rule. In this case all possible mappings are recorded as an equivocal mapping. Unfortunately the existence of equivocal mappings complicates the job of tracing execution paths, raising the question of whether an abstract rule can be considered to have fired if it is a participant in an equivocal mapping. Therefore different strategies are used to trace paths, based on whether or not equivocal rule mappings are to be counted. In our approach, since the effect of concrete rule firings is indicated directly in the graph representation of the rule-base, we can determine rule-base coverage directly from the graph after the test data is run. We determine the extent of rule-base coverage by considering whether the state of the graph after the test data is run satisfies the four rule-base coverage measures.

Third, our approach can be extended in a straightforward way to handle rule-bases with dynamic computation of certainty factors (see Chapter 6). Grossner et.al.’s work does not currently handle certainty factors and they do not appear, at the present time, to be planning to add such a capability to the system\footnote{Personal discussion with Clifford Grossner.}.

### 5.3.7 Logical Path Graph Model

The logical path graph (LPG) model, developed by Kiper \cite{GKLP92,Kip92} is based on program control flow analysis. It represents an attempt to apply cyclomatic complexity and basis path testing to the rule-base environment. Kiper’s goal was to find a graphical representation of rule-bases that was equivalent to the control flow graphs of procedural programs, which could then be used to determine rule-base complexity and determine a set of paths through the rule-base that, when executed, would adequately test the rules and their interactions.

The logical path graph is a directed graph in which the nodes represent individual rules and the edges are determined by logical paths through the rule base. That is, an edge will exist from node $i$ to node $j$ if all conditions asserted on a path to node $i$ along with those of node $i$ itself will cause the rule corresponding to node $j$ to fire. Because
rule \( j \) cannot fire until rule \( i \) has fired, rule \( j \) logically follows rule \( i \), but rule \( i \) is not the only rule that must fire in order for rule \( j \) to fire.

One goal of Kiper's work is to use the LPG to determine a set of paths through the rule-base such that traversal of those paths during system testing represents an adequate test of the rules and their interactions. Since the graph is based on logical, rather than structural, relations among elements of the rule-base, it can show the logical flow of data through the rule-base. Because of this, it should be possible to apply data flow test path selection techniques to the LPG in order to get paths that satisfy various adequacy criteria (such as those discussed in section 3.3).

Kiper argues that the LPG will be strongly connected and that, therefore, the cyclomatic complexity metric can be applied to it, generating a count of the number of basis paths through the rule-base. However, there are certain problems that arise in the use of logical paths. If there are multiple edges entering a node in the graph, they can be interpreted either as an AND or an OR relation. In order to avoid the possibility of OR edges the node must be replicated, in effect creating in the LPG the kind of redundancy which we usually try to remove from rule-bases. This results in the possibility of a single rule being represented by multiple nodes. If both AND and OR edges are to be allowed then the person evaluating the LPG must know what type each edge is. In addition, because of the possibility of multiple nodes representing a single rule, each node has to be labeled not just with the rule number but also with the condition set, the set of all conditions asserted by nodes on the path leading to the node.

While there are a number of similarities between this work and our approach, there are also significant differences. The graph structure we have proposed is based on findings and hypotheses and directly models the logical relations within rule antecedents, whereas the structure used for the LPG is built at the rule level. This difference in the graph construction allows us to avoid some of the problems that arise with the LPG representation. For example, we recall that in our graphs there is an explicit representation of each operator node (AND, OR, NOFM). Therefore we avoid the need to put identifying information on edges or replicate rule representations, as is the case in the
logical path graph.

5.4 Summary

In general, based on this overview of related work, we can see that the majority of formal expert systems evaluation methods focus on verification only, or on a strictly functional approach to validation. By adding a structural analysis to the validation process, our approach enhances the information that is gained about the rule-base and improves the predictions we can make about future performance of the system.
Chapter 6
Conclusions, Analysis, and Future Work

6.1 Conclusion

In this thesis we have introduced the concept of rule-base coverage analysis, and shown a number of ways in which this analysis can be applied during the evaluation of a rule-based system. Using several sample rule-bases we have demonstrated that this approach can facilitate rule-base pruning and can guide test case selection. In general, evaluating rule-base coverage can identify ways in which rule-base testing has been inadequate and can point to specific weaknesses in the test suite as well as specific sections of the rule-base that may be incorrect.

While it is intuitively obvious that there is a relationship between completeness of the test set and the quality of the testing, we can conclude from this work that there are several additional factors involved in the testing process that affect the conclusions we can draw from testing a rule-base on a suite of test data. These factors are:

1. meta-knowledge about the kinds of cases the system should handle

2. the relative coverage of the test set relative to the more complete test set defined by the meta-knowledge

3. coverage of the rule-base resulting from execution of the test set

Without consideration of these factors, performance evaluation of the rule-base can easily be in error. However, by analyzing these factors we can improve both the test data and the rule-base, incrementally increasing the portion of the rule-base which we know to have been used during the testing process. This is demonstrated most clearly in the thesis by the use of the coverage analysis to guide data selection and rule-base
• $r$ – number of rules
• $f$ – number of findings
• $c$ – number of classes
• $l$ – length of longest reasoning chain
• $a$ – number of unique antecedent components (number of findings plus number of distinct sub-classes)
• $t$ – maximum number of terms (literals or NOFM) in an antecedent
• $m$ – maximum number of terms in an nofm construction

Figure 6.1: Variables used in complexity analysis.

pruning.

The coverage approach to rule-base analysis presented in this thesis is part of a small, but growing, body of work in the area of automated methods for rule-base verification, validation and testing. As these automated methods improve they should help to increase confidence in expert systems, particularly in sensitive application areas in which the cost of system failures can be quite high.

6.2 Analysis

The algorithmic analysis of our approach to rule-base testing and analysis must include the cost of the graph construction, as well as that of the static and dynamic analysis phases. In Figure 6.1 we define variables which will be used in discussion of the algorithmic complexity of the different parts of our method. As much of the complexity of algorithms which operate on graphs depends on the graph size, we first discuss the graph size, followed by discussion of the complexity of other aspects of our method.

6.2.1 Graph Size

Given that a single finding will often appear in the antecedent of more than one rule, we expect the number of edges to exceed the number of nodes, and therefore focus on the
number of edges as an indication of the graph size. Independent of the rules in the rule-base, there will be \( f \) edges, one from the source node to each of the findings. There will also be \( c \) edges, one from each class to the sink node. Each rule contributes an additional edge to the consequent of that rule, for a total of \( r \) edges. The remainder of the edges are in the graph representation of the antecedent of each rule. With a maximum of \( t \) terms in an antecedent, since the antecedents are converted to conjunctive normal form, each antecedent could theoretically contribute an exponential number of edges. However, this should not actually be the case when using our method for rule-base analysis.

When an antecedent is parsed the corresponding graph is built down the right spine of the parse tree. That is, if the antecedent contains a disjunction at all \(^1\) the resulting graph will take on one of the forms shown in Figures 6.2. While the initial graph structure alone is not enough to bring down the size of the CNF representation, a limit on the size of the antecedents would allow us to bound the size of the CNF representation in a very favorable way. In fact, we believe that there is a reasonable upper bound on \( t \), with a majority of expert system rules having no more than 5 terms in an antecedent. Based on this limit, consider an antecedent of the form

\[(a \text{ and } b) \text{ or } (c \text{ and } d \text{ and } e)\]

which is as follows in CNF:

\(^1\)Not all rule-base languages allow disjunction in the rule antecedents, forcing the use of separate rules in situations in which two distinct conditions can lead to the same consequents.
(a or c) and (a or d) and (a or e) and (b or c) and (b or d) and (b or e)

While we do not actually replicate nodes in the graph, we will have multiple edges to the node for each term that is shown multiple times in the CNF representation. In fact, if we assume that in the worst case each term is represented 3 times in the CNF, then we have a total of 15 edges to the 5 nodes which represent the terms of the original antecedent. However, we must also consider the case in which all the terms were NOFM nodes with the maximum of \( m \) terms in them. In that case there will be a total of \( 15m \) edges from the CNF graph representation to the distinct finding or sub-classes referred to in the terms of the antecedent. Furthermore, in this worst case of 5 original terms which are each represented in triplicate in the CNF, there will be a maximum of 14 operators, with 2 edges to each operator.

This allows us to conclude that the CNF representation (in graph form) of an antecedent with a limit on the antecedent size will have on the order of 30 nodes, of which 10 will be the original term and operator nodes. We do not have to count the finding and sub-class nodes which make up the leaves of the CNF representation, since they are counted either as findings or as the consequents of other rules. The CNF representation will also have at most \( 15m \) edges to finding or sub-class nodes, of which \( 10m \) are new, and 30 edges to operator nodes, of which 20 are new. Therefore we can conclude that each antecedent contributes at most 20 new graph nodes and \( 10m + 20 \) new edges.

Recalling that there are \( r \) rules, we can see that the number of edges, therefore, is

\[
\mathcal{O}(f + c + r(10m + 20))
\]

which is

\[
\mathcal{O}(f + c + r)
\]

assuming that the number of rules is much larger than the maximum number of terms in a single \textbf{nofm} construction.
6.2.2 Graph Construction

The cost of the graph construction results from the hash table lookups and creation of the graph edges. A table lookup is carried out for each finding before it is added to the graph, so that any finding which is listed more than once in the input file will not be duplicated in the graph. It is also carried out for the antecedents and the consequents as the rules are processed. There is a lookup for each complete antecedent (one per rule, for \( r \) lookups), one lookup for each component within each antecedent (at most \( t \) for each rule), and one for each consequent (again \( r \), since each rule has one consequent).

If we assume that there are more rules than findings and we assume simple uniform hashing [CLR90], then each lookup is of constant time and the total cost of lookups carried out during graph construction is

\[
O(f + r + r \times t + r) = O(f + r)
\]

since we expect \( r \), the number of rules, to be significantly larger than \( t \), the maximum number of terms in an antecedent.

During graph construction all of the graph edges are also created at a cost of

\[
O(f + c + r)
\]

based on the number of edges in the graph. Therefore the total cost of the graph construction is

\[
O(f + c + r)
\]

6.2.3 Static Analysis

The static analysis is carried out by doing a depth-first traversal of the graph. The complexity of the depth-first traversal is based on the number of edges in the graph, which we have already seen is \( O(f + c + r) \).

6.2.4 Dynamic Analysis

Our approach to dynamic analysis consists of the data percolation step and the evaluation of the coverage measures. We consider the complexity of these steps separately.
Running test cases

When a test case is percolated through the DAG, each node of the DAG must be considered. In addition, for each sub-class node, OR node, AND node, class node or \texttt{nofm} node we must look at all parents of the node via the edges linking a node to its parents. We do not have to consider the edges from the findings to the source node and from the sink to the class nodes. Therefore the cost of considering each test case is essentially

$$\text{# nodes} + \text{# edges} - \text{# findings} - \text{# classes}$$

which is essentially $\mathcal{O}(\text{number of edges})$, which we have seen is $\mathcal{O}(f+c+r)$. Since this is the complexity of each test case, when using this approach to test an expert system, we would like to satisfy the coverage measures with as few test cases as possible.

Evaluating the rule-base coverage measures

For each rule-base coverage measure we must consider the cost of determining whether or not it has been satisfied by the test cases input thus far to the rule-base, in its DAG representation. We consider each of the five coverage measures in turn. Note that we are not considering how much test data is required to satisfy the coverage measures, but rather how much work is involved in determining if the test data satisfies the coverage measures.

- Each-class

Much of our processing is simplified if we maintain a list of classes separate from the DAG. (In actuality this is a list of pointers to the class nodes in the graph). In order to determine if the test data has caused the conclusion of each class, we use the class list to find the classes in the graph and evaluate their status. Therefore the cost of determining if the test data satisfies Each-class is $\mathcal{O}(c)$, based on the number of classes in the expert system being tested. This check will also

\footnote{With class nodes, sub-class nodes, and OR nodes we want to consider all parents in order to maximize the certainty factor of the node, rather than stopping with the first true parent.}
report on any class for which there are no rules in the system, indicating either a typographical error in the classes section of the input or a section of the rule-base that has not been completed. (Our static analysis process will also identify classes for which there are no rules. However, since it adds nothing additional to the cost of checking whether Each-class is satisfied, and since the user might choose to skip the static analysis, it is useful to also identify this information here).

- **Each-hypoth**

  In order to determine if the test data satisfies Each-hypoth, we must look at all nodes in the DAG. However, we are only concerned with the status of nodes which have been classified as classes or sub-class nodes, and can ignore all other categories of nodes. The complexity of checking if Each-hypoth is satisfied is therefore based on the number of nodes in the DAG representation of the rule-base. There will be one node for each finding and one for each rule (this counts the rule consequents, which includes all classes and sub-classes). A maximum of 20 operator nodes are added by the conversion to the CNF representation of a rule antecedent. So, for all \( r \) rules, the maximum number of nodes in the DAG representation is \( O(f + r) \), and this is also the complexity of determining whether the test data has satisfied Each-hypoth.

- **Each-class-every-finding**

  Each class has associated with it two lists of findings. The all-findings list contains all findings which can contribute to conclusion of the class over some path in the rule-base (the finding-foundation for the class). There are \( f^c \) possible finding-class combinations in the entire rule-base, and in the worst case we would require traversal of a separate sub-DAG for each combination (in reality, each sub-DAG to a class will include a number of findings). The second list associated with each class is the pass-along list which contains all findings which have actually contributed to the conclusion of the class during some test case. In order to determine whether or not the test data has satisfied Each-class-every-finding, we consider both the all-findings and the pass-along lists of each class. These
lists have previously been sorted and all duplicates removed from each list. We check each member of the all_findings list, reporting only those elements that do not appear on the pass_along list. In the worst case each class would have all of the findings of the system on both its all_findings and its pass_along lists (though we do not expect this to ever be the case in real systems). In that case the cost of sorting the two lists would be \( O(f^2) \), using insertion sort to sort the lists. (Insertion sort was chosen precisely because we expect the lists to be relatively short). The worst case cost of looking up elements of the all_findings list on the pass_along list would also be \( O(f^2) \), making the worst case total cost of evaluating Each-class-every-finding for all classes \( O(c \times f^2) \). Certainly, should additional testing of our method show that the all_findings and pass_along lists are large, the insertion sort could be replaced with a faster sorting algorithm, giving a worst case for checking Each-class-every-finding of \( O(c \times f \log(f)) \).

- **Each-class-every-sub**

As is done to collect finding information for each class, each class also has two lists of sub-classes: all_subs and sub_pass_along. The determination of whether Each-class-every-sub has been satisfied is carried out in an analogous fashion to the determination of whether Each-class-every-finding has been satisfied. In the worst case (only \( c \) rules conclude classes) there are \( r - c \) rules which conclude distinct sub-classes. In this case the cost of evaluating Each-class-every-sub is \( O(c \times (r - c)^2) \) which is essentially \( O(c \times r^2) \).

- **All-edges**

All-edges is satisfied if every edge has been used along at least one execution path (as defined in Chapter 3) during evaluation of the test data. Satisfaction of this coverage measure will give the greatest assurance that the system being tested is correct, as it guarantees that every possible inference chain within the system has been used by at least one test case. We expect that this coverage measure will be the most costly to satisfy, since we expect it to require more test data than that necessary to satisfy the other coverage measures. It is also more
costly to determine whether or not All-edges has been satisfied, as compared to the cost incurred when evaluating the other coverage measures.

In order to determine if All-edges was satisfied, we must consider each edge in the DAG representation of the rule-base in order to see if the edge was marked (traversed) during execution of the test cases. If an edge was not marked then it is part of an untraversed execution path. Some of the edges of the execution path may have been marked by other test cases, but they will still be reported as being part of an untraversed execution path for the rule-base. Once an unmarked edge is found, we want to traverse both up and down the graph from that point, reporting on all the nodes which are not operator nodes (not AND, OR, NOFM nodes) that are part of the untraversed execution path, generating a list of the findings and sub-classes involved in the path. This will allow us to also identify the rule number of each rule which contributed to the construction of the untraversed execution path. Therefore the cost of evaluating All-edges is $O(f + c + r)$, based on the number of edges in the DAG.

Finally, in the TRUBAC implementation, if All-edges was not satisfied by the test data then the user can ask TRUBAC to generate data which would lead to traversal of all the as yet untraversed execution paths. The cost of generating the data is the same as that of evaluating the coverage measure in the first place, that is $O(f + c + r)$.

### 6.3 Future Work

The work presented in this thesis can be extended in several ways. The primary, but most difficult, area for further work is that of quantitative performance prediction. There are also a number of ways in which this work can be further developed with regard to the actual method proposed and with regard to the TRUBAC implementation.

#### 6.3.1 Extension – Performance Prediction

In the area of performance prediction, the question we would like to address is whether there is some way that we can derive a quantitative performance prediction (i.e. the
system will give a correct result for X% of cases it handles in actual use) based on

1. the degree to which the test suite is representative of the population for which
the system is intended,

2. the likelihood of occurrence of different kinds of cases in the population at large,

3. the degree of rule-base coverage,

4. correspondence between expected and actual results when the system is run on
test cases with known results.

For example, let us assume we run N test cases on a rule-base and get correct answers
for 80% of them. However, we determine that only 75% of the basis paths have been
executed, with a number of sub-class to class and fact to class relationships unsatisfied.
Furthermore, the test cases represent only 75% of the types of cases that can occur,
although we expect those types of cases 90% of the time in the general population.
We would like to be able to make a performance prediction for the rule-base which
takes into account all of these factors. In our future work we would like to pursue this
analysis, based on a statistical model of the population from which the test cases are
drawn.

6.3.2 Extension – Handling Certainty Factors

Although many tools for dynamic evaluation of expert systems have been discussed
in earlier chapters, none of the tools mentioned is designed to carry out validation of
rule-bases that normally run in an inferencing environment in which there is dynamic
computation of certainty factors. Similarly, our approach as described thus far is geared
towards rule-bases without certainty factors or those, such as the EXPERT rule-bases,
which have hard-coded certainty factors (CFs that are determined by the expert as the
system is developed, rather than computed as inferencing is carried out). In this section
we briefly discuss modifications of the rule-base coverage measures for which could, in
the future, be applied to systems with dynamic computation of certainty factors. We
follow this with a brief discussion of certainty factors in EXPERT rules and how they
are handled by our approach. We then present a simple example and discuss the use of semantic information about the problem as an aid in assessing the possibility that coverage measures can be completely satisfied.

**Rule-Base Coverage Measures with Certainty Factors**

The inclusion of dynamically computed certainty factors raises a number of issues for the application of the rule-base coverage measures defined in Chapter 3. For example, if a particular test case leads to the conclusion of some class, but only with very low certainty, should that satisfy **Each-class** for that class, or must the class be reached with high certainty in order to satisfy **Each-class**? Usually certainty factors are expressed as a value within an interval, typically either \([0,1]\) or \([-1,1]\). However, working with certainty factors over such a continuum would make it extremely difficult to incorporate them into the DAG representation. Therefore we propose dividing the uncertainty range into three subranges. For example, for the interval \([0,1]\) we propose dividing it into the subranges

- **"low"** – values in the interval \([0, .4)\)
- **"medium"** – values in the interval \([.4, .7)\)
- **"high"** – values in the interval \([.7, 1]\)

For each (intermediate and final) hypothesis node we can then extend the coverage measures to track whether the node has been concluded with certainty in each of the three ranges over the course of a suite of test cases. We present a modification of the rule-base coverage measures to guide and evaluate the testing of rule-bases with dynamic computation of certainty factors.

1. **Each-class-cf** Provide test data to cause traversal of one execution path to each class of the system for each certainty range (a total of 3 paths to each class).

2. **Each-hypoth-cf** Provide test data to cause traversal of execution paths such that each intermediate conclusion is reached, as well as each class, with certainty in each of the three ranges.
3. **Each-class-every-finding-cf** Provide data such that for each finding-class combination connected by some execution path, at least one execution path which includes the combination is executed for each certainty factor range.

4. **Each-class-every-sub-cf** Provide data such that for each sub-class to class combination connected by some execution path, at least one execution path which includes the combination is executed for each certainty factor range.

5. **All-edges-cf** Provide data to cause traversal of all execution paths for each certainty factor range.

Certainly we expect that the number of test cases necessary to satisfy this modified set of coverage measures, given the DAG representation of a rule-base, will be greater than the number needed for a rule-base without certainty factors that has a comparable DAG representation. However, the exact increase is hard to quantify for several reasons. It is possible that one test case causes a single class to be concluded multiple times over different execution paths, with a different certainty factor resulting from each execution path (which could indicate an error in the rule-base). It is also possible that one test case causes several classes to be concluded, satisfying one certainty range for each of the classes concluded. The fact that we consider certainty factors in 3 ranges does not necessarily directly imply a three-fold increase in the amount of test data required to satisfy the coverage measures. This is one area of additional research.

An additional issue raised by the inclusion of certainty is that of changes in the data generation process. If the rule-base does not have certainty factors then we can use the graph to generate data which would cause traversal of untraversed execution paths, and submit that data to the expert for review. The expert may accept that data as a valid case in the problem domain, or reject it as indicative of an error in the rule-base. In the case of dynamic computation of certainty factors, we have to consider each execution path in conjunction with the 3 certainty factor ranges, and distinguish between a path which has not been traversed at all by the user's test data and a path which has been traversed but has not made a conclusion with certainty in each of the 3 ranges. Some execution paths may not be able to ever produce a final class with the certainty factor in
a particular range, due to the relationships established in the rule-base. As discussed in Section 6.7, by using semantic information about the problem we can determine paths for which the rule-base will never conclude the class with the CF in a particular range, thereby eliminating the need to try to cover those paths in certain CF ranges.

**Data Selection Heuristics in Systems With Certainty Factors**

In testing rule-bases which use dynamic computation of certainty factors, we can apply a modified set of heuristics for test data selection. In our original model of a classification system we viewed the complete population as containing all combinations of binary values for findings, sub-classes and classes. In such a model the full set of possible conclusions which could result from a test case would be the complete set of combinations of classes. That is, if we had two classes, $C_1$ and $C_2$, then each test set could either conclude both $C_1$ and $C_2$, conclude only $C_1$ or $C_2$, or conclude neither class. In the situation in which we have dynamic computation of certainty factors, the model of a classification system is somewhat more complex, and our heuristics must be modified accordingly.

If we have dynamic computation of certainty factors, then each class can be concluded in any one of the three ranges (in general, if a particular test case can conclude a single class over multiple reasoning chains with different CF values, the rule-base will report just the maximum value for that class). Continuing with our example of two classes, in the situation in which a test case concludes both classes there are nine possible combinations of CFs, since each class can be concluded in each of 3 CF ranges. As in the case without certainty factors, we would like to focus primarily on the discriminatory power of the system under test. This leads us to focus on test cases which conclude a single class with high CF. Of cases which conclude multiple CFs, we will focus on those which conclude one class in the high CF range, and the other class in the low CF range. Test cases which conclude a single class in the medium range or two classes in the same or close ranges are ambiguous and represent examples of situations in which the system is less effective at discriminating between classes.
The revised heuristics for test data selection while evaluating a rule-base with dynamic computation of certainty factors are

1. For each class, select a test case which uniquely concludes it, or concludes it with high CF while other classes concluded by the test case are concluded with low or medium CF.

2. Select test cases which will satisfy unused sub-class/CF range and class/CF range combinations.

3. Select test cases which will cover sub-class to class relationships, focusing on concluding the class with high CF.

4. Select test cases which will cover finding to class relationships, again focusing on concluding the class with high CF if possible.

Certainly, as we will see in section 6.7, certain sub-class to class and finding to class relations will only exist in certain CF ranges. Therefore we attempt to find test cases which will discriminate between classes by concluding a class in the high CF range. However, we recognize that some findings or sub-classes may only be involved in the conclusion of a class in a lower CF range, and we will run test cases that satisfy those relations as well.

Use of Semantic Information

In order to illustrate the use of semantic information to aid assessment of the coverage actually achieved, consider the rule-based system, shown in Figure 6.6, that is similar to the CAR rule-base discussed in earlier chapters. If we assume that all findings will be known with complete certainty and we apply MYCIN type combination rules, then we can see that the most extensive coverage we can hope to get is that shown in Table 6.1. With this information we would be able to determine the extent to which we could expect to achieve coverage in each of the CF ranges. For example, we would consider Each-class to have been satisfied if we had provided test data which concluded each class in those ranges that are indicated as possible in the table.
if lgas then flod(0.80)
if between(temp, 0, 50) AND scrn then chok(0.60)
if between(temp, 0, 50) AND scrn then chok(0.60)
if egas then empt(0.90)
if cflt then filt(1.00)
if lcab then cab(0.90)
if scrn AND dim then batd(0.70)
if ncrn AND dim then batd(0.90)
if ncrn AND not dim then strt(0.70)
if grnd then strt(0.90)
if fcws AND flod then wait(0.90)
if fcws AND chok then open(0.50)
if fcws AND empt then gas(0.90)
if fcws AND filt then rflt(0.80)
if fcws AND cab then clen(0.70)
if fcws AND batd then gbat(0.80)
if fcws AND strt then nstr(0.90)

Figure 6.3: Modified car rule-base

<table>
<thead>
<tr>
<th>CLASS</th>
<th>Low</th>
<th>Medium</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait</td>
<td>x (.72)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>open</td>
<td>x (.30)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gas</td>
<td>x (.81)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rflt</td>
<td>x (.80)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>clen</td>
<td>x (.63)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gbat</td>
<td>x (.56)</td>
<td>x (.72)</td>
<td></td>
</tr>
<tr>
<td>nstr</td>
<td>x (.56)</td>
<td>x (.72)</td>
<td>x (.81)</td>
</tr>
</tbody>
</table>

Table 6.1: Possible CF range coverage for classes
We can extend the use of semantic information if we have additional knowledge of possible certainty with which findings are known when they are presented to the system. For example, the findings in our example rule-base may be known with varying degrees of certainty when they are reported to the system, as indicated in Table 6.2. Knowing that the findings will have CFs within these ranges, we can revise our expectation of what coverage we could achieve, as shown in Table 6.3. With this additional information we can now see that we can never expect to conclude wait, open, rflt, and clen in the high CF range or open in the medium CF range. In evaluating the coverage measures we would then want to take those impossible class-CF pairings into account. In actuality, in the implementation of the method, we can ask the developer for information about the possible CF range for each finding, and identify impossible class-CF combinations so that when all but those pairs have been generated by the test data the tester would be informed that complete coverage (within the particular coverage measure selected)
had been achieved. In this way we make use of semantic information about the problem being solved to help assess the coverage achieved by the testing process.

6.3.3 Additional Extensions

In addition, there are a number of ways in which the present work can be extended to make it a more useful testing approach. First, our current approach handles only rules that consider if the findings are true or not or have a value within a certain range, and then conclude that some sub-class is true (or not). We would like to extend the approach to work with variables to which a value is assigned in the consequent of some rule. For example, consider the rule (from a rule-base used in [Pre89]) of the form

\[
\text{if } a = a_1 \text{ and } b = b_1 \text{ then } g = g_1
\]

This presumes that \( a \) and \( b \) could have values other than the ones that would make the antecedent of this rule evaluate to true. Therefore, in the representation there would have to be tests on the edges of the graph (i.e. an edge from \( a \) to a use of \( a \) in an antecedent, checking for the desired value of \( a \)), or there has to be a replication of nodes (i.e. a node for value \( a_1 \) of \( a \), and one for value \( a_2 \) of \( a \) etc.).

Our second area for future work involves \textbf{All-edges} and its reporting of untraversed execution paths. Currently we list all untraversed execution paths, leaving it to the user to decide how best to go about trying to satisfy them. It would be more useful to prioritize the untraversed paths in some way so that the search for additional test cases could be focused. Possible approaches are:

- When the rule-base is first processed, include information from the user which prioritizes the findings, classes, or sub-classes. This will allow emphasis of any execution paths involving high priority elements. For example, we could modify the pseudo code for \textbf{All-edges} as indicated in Figure 6.3, in which PRIORITY is the priority level selected by the user.

- Get information from the user as to any rules which are believed to be of particular importance, and emphasize any execution paths that involve those rules.
for (every node V in the graph)
    if (V is not the source node and V's priority >= PRIORITY)
        /* edges from source to findings are never marked -- skip them */
        /* only interested in V if it is a high PRIORITY node */

        for (every edge U which points to a neighbor of V)
            /* only care if the edge is not from a class to the sink, */
            /* and the edge wasn't marked during dynamic test */
            if (the edge was not marked yet and it doesn't point to the sink)
                {
                    if (V is a finding)
                        report V
                    if (U points to a class or a sub-class node)
                        report the class or sub-class
                    if (U points to an AND)
                        walk up graph from the AND to find all findings which
                        contribute to positive conclusion of the AND
                        from the node pointed to by U, walk down to an AND and
                        then walk up to find other findings which are necessary to
                        cause execution of the path containing the unexecuted edge
                }

Figure 6.4: Pseudo code for All-edges with priorities.
Initial development

Initial testing, with creation of CASES, RESULTS, COVERAGE

When modifications/additions are necessary

Add to the system

Automatic retest, using CASES. Carry out automatic comparison of new results with RESULTS, new coverage with COVERAGE, to identify unintended changes in system behavior.

New testing, with additions to CASES, RESULTS, COVERAGE.

Figure 6.5: Stages of incremental development and testing

Thirdly, we would like to explore the use of the coverage approach for testing during incremental development of a system. In particular, we would like to save the DAG representation of a rule-base, along with the tests which have been run on it, and rerun them automatically on a modified DAG after changes have been made to the rule-base. After additions are made to the rule-base the user would only have to provide test data to cover any new rules and the interaction between new and old rules. We would also like to be able to generate information on the areas of change, i.e. the ways in which new additions to the rule-base interact with old rules. This would alert the developer to unintended interactions between rule in the rule-base. Rarely will a rule-base be a static unchanging entity. A rule-base is only as good as the knowledge that went into its construction. Even if a rule-base is tested with complete coverage and provides correct answers on each execution path, as knowledge about the problem domain increases the rule-base will become wrong and outdated. There is a need to be able to easily retest any rule-base which undergoes modification in order to reflect the current body of knowledge about the problem domain. The ability to use our approach during incremental development would greatly ease the retesting of a system that has been modified to accommodate advances in the available knowledge. Figure 6.4 shows the stages of such an approach to incremental development.
We would also like to study the relationship among the ten coverage measures. Clearly among our original five coverage measures the data that satisfies a stronger coverage measure will certainly satisfy a weaker coverage measure (as ordered in the lattice shown in Figure 3.6). This property also holds true among the extended coverage measures. However, we would like to determine what relationships hold between the two sets of coverage measures. In particular, we would like to determine an ordering of the 10 coverage measures such that satisfying a coverage measure that does not consider certainty factors will still give us useful information about and guidance for testing of a rule-base with certainty factors. This is another aspect of continued work in this area.

Finally, we would like to explore the extension of the rule-base coverage approach to rule-based systems of different types, such as systems for planning, design, forecasting, and training.
Appendix A

Format of EXPERT Rule-Bases

Major sections are indicated by **, and subsections are indicated by *. Optional sections are marked by brackets.

**HYPOTHESES
*TAXONOMY
 .
 [*INTERMEDIATE HYPOTHESES]
 .
 [*TREATMENTS]
 .
 **FINDINGS
 .
 **RULES
 [*FF RULES]
 .
 .
 *FH RULES
 .
 .
 [*HH RULES]
 .
 .
Appendix B

Simple EXPERT Rule-Base

**HYPOTHESES

*TAXONOMY

CWS Car Won't Start
FUEL .Fuel System Problems (.5)
FLOD .Car Flooded (.2)
CHOK .Choke Stuck (.2)
EMPT .No Fuel (.2)
FILT .Fuel Filter Clogged
ELEC .Electrical System Problems (.5)
CAB .Battery Cables Loose or Corroded (.2)
BATD .Battery Discharged (.2)
STRT .Starter Malfunction (.2)

*TREATMENTS

REP Car Repairs
WAIT .Wait 10 minutes or Depress Accelerator to Floor+ while Starting.
OPEN .Remove Air Cleaner Assembly and Manually Open Choke+ with Pencil
GAS .Put More Gasoline into Tank
RFLT .Replace Gas Filter
CLEN .Clean and Tighten Battery Cables
GBAT .Charge or Replace Battery
NSTR .Replace Starter

**FINDINGS

*CHECKLIST

Type of Problem:
CWS Car Won't Start
FOTH Other Car Problems

*Multiple Choice
Odor of Gasoline in Carburetor:
NGAS None
MGAS Normal
LGAS Very Strong

*Checklist
Simple Checks:
DIM Headlights are Dim
CFLT Fuel Filter Clogged
LCAE Battery Cables Loose/Corroded

*Checklist
Starter Data:
NCRN No Cranking
SCRN Slow Cranking
OCRN Normal Cranking
GRND Grinding Noise From Starter

*Numerical
TEMP Outdoor Temperature (degrees F):

*Yes/no
EGAS Gas Gauge Reads EMPTY

**RULES
*FH Rules

F(LGAS,T)->H(FLOD,.8)
F(TEMP,0:50)&[i:F(SCRN,T),F(DCRN,T)]->H(CHOK,.6)
F(EGAS,T)->H(EMPT,.9)
F(CFLT,T)->H(FILT,1.0)
F(LCAB,T)->H(CAB,.9)
F(SCRN,T)&F(DIM,T)->H(BATD,.7)
F(NCRN,T)&F(DIM,T)->H(BATD,.9)
F(SCRN,T)&F(DIM,F)->H(STRT,.7)
F(FCWS,F)->H(CWS,-1)

*HH Rules
*IF
F(FCWS,T);
*THEN
H(FLOD,.2:1)->H(WAIT,.9)
H(CHOK,.2:1)->H(OPEN,.5)
H(EMPT,.3:1)->H(GAS,.9)
H(FILT,.4:*)->H(RFLT,.8)
H(CAB,.5:*)->H(CLEN,.7)
H(BATD,.4:1)->H(GBAT,.8)
H(strt,.4:1)->h(nstr,.9)
*END
Appendix C

TRUBAC Version of Simple Rule-Base

*FINDINGS
fcws
foth
ngas
mgas
lgas
dim
cflt
lcb
ncrn
scrn
ocrn
gnd
temp
egas

*CLASSES
wait
open
gas
rflt
clen
gbat
nstr
cws

*RULES
if lgas then flod(0.80)
if between(temp, 0, 50) AND nofm(1, scrn, ocrn) then chok(0.60)
if egas then empt(0.90)
if cflt then filt(1.00)
if lcb then cab(0.90)
if scrn AND dim then btd(0.70)
if ncrn AND dim then btd(0.90)
if ncrn AND not dim then strt(0.70)
if gnd then strt(0.90)
if not fcws then cws(-1)
if fcws AND between(flod, 0.20, 1) then wait(0.90)
if fcws AND between(chok, 0.20, 1) then open(0.50)
if fcws AND between(empt, 0.30, 1) then gas(0.90)
if fcws AND between(filt, 0.40, 1) then rflt(0.80)
if fcws AND between(cab, 0.50, 1) then clen(0.70)
if fcws AND between(btd, 0.40, 1) then gbat(0.80)
if fcws and between(strt, 0.40, 1) then nstr(0.90)
Appendix D

Sample TRUBAC Session

The test case numbers used in this session refer to the test cases shown in Appendix E.

TRUBAC
Testing with Rule-Base Coverage

Enter filename to be translated, or q to quit:

Show conflict and redundancy messages? (Y or N)

GRAPH STATISTICS:
-------------
44 nodes
11 findings
8 classes
17 rules

Do you want to run the static tests? (Y or N)  y

*** the static tests inform us that three findings are used in no rules
*** and could be removed from the rule-base

finding mgas is used in no rules
finding ngas is used in no rules
finding foth is used in no rules

Do you wish to run the dynamic tests? (Y or N)  y

*** Each test case is stored in a file. File name is the rule-base
*** name with the test case number appended, e.g. car1, car2, etc.

Select: 1, 2, 3, 4, 5 for a RBCM; (L)imits or
       (C)ount; (O)verlap table; (U)pdated table; (S)elect updates;(T)ests; (Q)uit  t
Test case number:  1

Test case 1 concludes rflt with CF 0.8
Test case 1 concludes gbat with CF 0.8

Select: 1, 2, 3, 4, 5 for a RBCM; (L)imits or
       (C)ount; (O)verlap table; (U)pdated table; (S)elect updates;(T)ests; (Q)uit  t
Test case number:  2

Test case 2 concludes clen with CF 0.7
Test case 2 concludes nstr with CF 0.8
Select: 1, 2, 3, 4, 5 for a RBCM; (L)imits or
(C)ount; (O)verlap table; (U)pdated table; (S)elect updates; (T)ests; (Q)uit

Test case number: 3

Test case 3 concludes open with CF 0.5
Test case 3 concludes gas with CF 0.9

Select: 1, 2, 3, 4, 5 for a RBCM; (L)imits or
(C)ount; (O)verlap table; (U)pdated table; (S)elect updates; (T)ests; (Q)uit

Test case number: 4
Test case 4 concludes wait with CF 0.9
Test case 4 concludes nstr with CF 0.9

Select: 1, 2, 3, 4, 5 for a RBCM; (L)imits or
(C)ount; (O)verlap table; (U)pdated table; (S)elect updates; (T)ests; (Q)uit

You did not provide test data to conclude cws

You did not satisfy Each-class

Select: 1, 2, 3, 4, 5 for a RBCM; (L)imits or
(C)ount; (O)verlap table; (U)pdated table; (S)elect updates; (T)ests; (Q)uit

Test case 7 concludes cws with CF -1.0

Select: 1, 2, 3, 4, 5 for a RBCM; (L)imits or
(C)ount; (O)verlap table; (U)pdated table; (S)elect updates; (T)ests; (Q)uit

You satisfied Each-class

Select: 1, 2, 3, 4, 5 for a RBCM; (L)imits or
(C)ount; (O)verlap table; (U)pdated table; (S)elect updates; (T)ests; (Q)uit 2

You satisfied Each-hypoth

Select: 1, 2, 3, 4, 5 for a RBCM; (L)imits or
(C)ount; (O)verlap table; (U)pdated table; (S)elect updates; (T)ests; (Q)uit 4

At least one path was executed for each sub-class to class combination.

Select: 1, 2, 3, 4, 5 for a RBCM; (L)imits or
(C)ount; (O)verlap table; (U)pdated table; (S)elect updates; (T)ests; (Q)uit 3

class nstr --
No path executed for the finding(s):
grnd

class gbat --
No path executed for the finding(s)
scnr
class open --
No path executed for the finding(s)

Would you like TRUBAC to generate test data based on untraversed finding-class combinations? (Y or N) n

Select: 1, 2, 3, 4, 5 for a RBCM; (L)imits or (C)ount; (O)verlap table; (U)pdated table; (S)elect updates; (T)ests; (Q)uit t
Test case number: 5

Test case 5 concludes open with CF 0.5

Select: 1, 2, 3, 4, 5 for a RBCM; (L)imits or (C)ount; (O)verlap table; (U)pdated table; (S)elect updates; (T)ests; (Q)uit t
Test case number: 6

Test case 6 concludes gb at with CF 0.8

Select: 1, 2, 3, 4, 5 for a RBCM; (L)imits or (C)ount; (O)verlap table; (U)pdated table; (S)elect updates; (T)ests; (Q)uit 3

class nstr --
No path executed for the finding(s):

Would you like TRUBAC to generate test data based on untraversed finding-class combinations? (Y or N) y

For finding GRND and class NSTR use data

finding grnd

finding fcws

*** Now we check all execution paths. Only the path involving the *** unsatisfied finding-class combination is left. So satisfying *** Each-class-every-finding should also satisfy All-edges for this *** example.

Select: 1, 2, 3, 4, 5 for a RBCM; (L)imits or (C)ount; (O)verlap table; (U)pdated table; (S)elect updates; (T)ests; (Q)uit

Untraversed sub-DAG detected involving
sub-class or class STRT
rule number 9
finding grnd
rule number 17
finding fcws

Would you like TRUBAC to generate test data based on untraversed sub-DAGs? (Y or N) n

Select: 1, 2, 3, 4, 5 for a RBCM; (L)imits or (C)ount; (O)verlap table; (U)pdated table; (S)elect updates; (T)ests; (Q)uit t
Test case number: 8

Test case 8 concludes nstr with CF 0.9

Select: 1, 2, 3, 4, 5 for a RBCM; (L)imits or (C)ount; (D)overlap table; (U)pdated table; (S)elect updates; (T)ests; (Q)uit

At least one path was executed for each finding-class combination.
Appendix E

Test Data Used In TRUBAC Session

Test case #1:
FCWS T
LGAS F
DIM T
CFLT T
LCAE F
NCRN T
SCRN F
OCRN F
GRND F
EGAS F

Test case #2:
FCWS T
LGAS F
DIM F
CFLT F
LCAE T
NCRN T
SCRN F
OCRN F
GRND F
EGAS F

Test case #3:
FCWS T
LGAS F
DIM T
CFLT f
temp 35
LCAE F
NCRN f
SCRN F
OCRN t
GRND F
EGAS t

Test case #4:
FCWS T
LGAS t
DIM f
CFLT f
LCAE F
NCRN T
SCRN F
OCRN F
GRND F
EGAS F

Test case #5:
FCWS t
LGAS F
DIM t
CFLT F
LCAE f
NCRN f
SCRN t
OCRN F
GRND F
EGAS F

Test case #6:
FCWS t
LGAS F
DIM t
CFLT F
LCAE f
NCRN f
SCRN t
OCRN F
GRND F
EGAS F

Test case #7:
FCWS f
LGAS F
DIM t
CFLT F
LCAE f
NCRN f
SCRN t
OCRN F
GRND F
EGAS F

Test case #8:
FCWS t
GRND t
Appendix F

Prolog-Based Language

\[
\text{<knowledge_base>} ::= \text{<expressions>}
\]

\[
\text{<expressions>} ::= \text{<expression>} | \text{<expression>}\text{<expressions>}
\]

\[
\text{<expression>} ::= \text{<declaration>} | \text{<rule_expr>}
\]

\[
\text{<declaration>} ::= \text{<goal_decl>} | \text{<askable_decl>} | \text{<constraint_decl>}
\]

\[
\text{<goal_decl>} ::= \text{goal <item> .}
\]

\[
\text{<askable_decl>} ::= \text{askable <item> / <type> / <values> .}
\]

\[
\text{<constraint_decl>} ::= \text{constraint <id> :: <constr_expr> .}
\]

\[
\text{<constr_expr>} ::= \text{<conjunction>}
\]

\[
\text{<rule_expr>} ::= \text{rule <id> :: <rule_body> .}
\]

\[
\text{<rule_body>} ::= \text{<consequent> if <antecedent> | if_ <antecedent> then <consequent>}
\]

\[
\text{<consequent>} ::= \text{<fact> | not <fact> | <assignment>}
\]

\[
\text{<antecedent>} ::= \text{<disjunction>}
\]

\[
\text{<disjunction>} ::= \text{<conjunction> | <conjunction> or <disjunction>}
\]

\[
\text{<conjunction>} ::= \text{<condition> | <condition> and <conjunction>}
\]

\[
\text{<condition>} ::= \text{<simple_expr> | not <simple_expr>}
\]

\[
\text{<simple_expr>} ::= \text{<fact> | <assignment>}
\]

\[
\text{<fact>} ::= \text{<item>}
\]

\[
\text{<assignment>} ::= \text{<item> hasValue <value> | <item> includes <value>}
\]

\[
\text{<value>} ::= \text{<item>}
\]

\[
\text{<type>} ::= \text{category / set / yesno / fact}
\]

\[
\text{<values>} ::= _ | \text{<item_list>}
\]

\[
\text{<item_list>} ::= [ \text{<items> } ] | [ ]
\]
\[<\text{items}> ::= <\text{item}> | <\text{item}> , <\text{items}>\]

\[<\text{id}> ::= <\text{item}>\]

\[<\text{item}> ::= <\text{atom}> | <\text{predicate}> ( <\text{items}> )\]

\[<\text{predicate}> ::= <\text{atom}>\]
References


Vita

Valerie B. Barr


1989-96  Attended Rutgers, The State University of New Jersey, for graduate school in Computer Science.

1989-91  Teaching Assistant, Department of Computer Science, Rutgers, The State University, New Brunswick, New Jersey.

1991-92  Visiting Instructor, Department of Mathematics, Statistics and Computer Science, Mount Holyoke College, South Hadley, Massachusetts.

1994-95  Visiting Assistant Professor, Department of Computer Science, Polytechnic University of New York, Brooklyn, New York.


1996    Ph.D. in Computer Science