FAULT ISOLATION IN COMPLEX DYNAMIC SYSTEMS USING ARTIFICIAL INTELLIGENCE TECHNIQUES

Jovan Barac

Submitted for the degree of PhD at

University College London

1990

ProQuest Number: 10609093

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

ProQuest 10609093

Published by ProQuest LLC (2017). Copyright of the Dissertation is held by the Author.

All rights reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

ABSTRACT

Complex dynamic systems, such as computer networks are characterised by the occurrence of multiple faults, sensor faults and intermittent faults as well as single faults. The aim of this dissertation is to provide a general, system independent expert system shell for isolating faults in complex dynamic systems. First a new high level architecture suitable for fault isolation in complex dynamic systems is suggested. This architecture is based on the coupling of a blackboard system with an Assumption-based Truth Maintenance System (ATMS). The blackboard provides a flexible means of control for the inferencing process, while the ATMS provides the means to store and retrieve the data, a record of the inferences, support for the blackboard functions and general support for non-monotonic and multiple context reasoning. Then a new fault model is developed which combines a unified approach to single, multiple and sensor faults with a simple temporal fault logic for intermittent faults. Following that, a novel, algorithmic test optimisation mechanism, based on the calculation of the expected entropy of each test is developed. This mechanism is used to determine the next test that should be performed and completes the system independent elements of the proposed diagnostic environment. The subsequent section shows how the fault model, the test optimisation algorithm and the high level architecture are combined to provide a diagnostic shell and how the ATMS and the blackboard cooperate in solving a diagnostic problem. The final section discusses the implementation, the performance evaluation, the preliminary validation in the chosen domain of computer networks and future validation and research issues.

To my parents

ACKNOWLEDGMENTS

I wish to thank my supervisor Professor Steve Wilbur for his help and support throughout this research. My thanks are also due to Elpida Keravnou, John Washbrook, Derek Long and John Campbell who read various drafts of this manuscript and gave me many helpful comments. This research was financially assisted with an SERC/CASE award from the company CASE, Watford for which I am grateful.

Table of Contents

ABSTRACT 2
ACKNOWLEDGMENTS 4
TABLE OF CONTENTS 5
LIST OF FIGURES 9
LIST OF TABLES 10

CHAPTER I: INTRODUCTION 11
1.1 Fault diagnosis: a perspective 11
1.2 Aim of the research 13
1.3 Motivation for the research 14
1.4 Thesis outline 18

CHAPTER II: A KNOWLEDGE-BASED ARCHITECTURE FOR FAULT ISOLATION IN COMPLEX DYNAMIC SYSTEMS 20
2.1 Introduction 20
2.2 Major characteristics of fault isolation in complex dynamic systems 20
2.3 Choosing a KBS architecture for fault isolation in complex dynamic systems 23
2.3.1 Rule- based systems (RBS) 23
2.3.2 Frame based systems 26
2.3.3 The blackboard framework 29
2.3.4 Summary 31
2.4 Moving from the blackboard model to a computable framework 32
2.5 Truth Maintenance Systems (TMSs) 33
2.5.1 The concept of TMS 33
2.5.2 Benefits of TMSs 34
2.5.3 Choosing a TMS 35
2.5.4 Assumption-based Truth Maintenance System (ATMS) 36
2.6 A blackboard architecture for fault diagnosis 42
2.6.1 Overall blackboard structure 42
2.6.2 Level control modules 44
2.6.3 Overall control module 46
2.6.4 Knowledge sources format 46
2.7 Interfacing the blackboard and the ATMS 47
2.7.1 Interfacing the consumer construct to level control 47
2.7.2 Adapting the ATMS to the proposed control structure 49
2.7.3 Using the ATMS to help triggering 51
2.7.4 Using the ATMS to help scheduling 52

2.8 Comparison to other blackboard architectures 54
2.9 Summary 57

CHAPTER III: DIAGNOSING SINGLE,MULTIPLE AND SENSOR FAULTS AND REACTING TO STATE CHANGES 58
3.1 Introduction 58
3.2 Notation and definitions 58
3.3 Characteristic points of complex dynamic systems 60
3.4 Existing multiple fault strategies 61
3.4.1 Explicit enumeration of possible faults 61
3.4.2 Network Diagnostic System (NDS) 61
3.4.3 The suspect islands/ clustering approach 61
3.4.4 Geffner & Pearl 63
3.4.5 Davis 64
3.4.6 Parsimonious covering theories 65
3.4.6.1 Single fault restriction 65
3.4.6.2 Minimal cardinality approach 65
3.4.6.3 Relevancy 65
3.4.6.4 Minimal candidates 66
3.4.7 Discussion 69
3.5 Existing sensor fault strategies 69
3.6 A new integrated sensor fault/minimal candidate approach 71
3.6.1 Assumptions 71
3.6.2 The meaning of test data 72
3.6.3 Minimal candidate generation 72
3.6.4 Discussion 78
3.7 Dealing with state changes 79
3.7.1 Time information during fault isolation 80
3.7.2 State change manifestation 84
3.7.3 State change /test fault duality 84
3.7.4 A model for dealing with state changes 85
3.7.4.1 Suspecting a state change 85
3.7.4.2 Testing/confirming a state change 88
3.7.4.3 Recovering from a state change 89
3.7.4.4 Algorithm refinements 91
3.7.5 Other approaches to temporal reasoning 92
3.8 Summary 94

CHAPTER IV: TEST OPTIMISATION DURING FAULT ISOLATION 95
4.1 Introduction 95
4.2 Algorithmic versus heuristic test optimisation 96
4.3 Assumptions 98
4.4 Entropy definition and properties 99

4.5 Test optimisation using entropy 100
4.5.1 Entropy and test generation 100
4.5.2 Basic notation and equations 100
4.5.3 Candidate probabilities of a failed test 103
4.5.4 Entropy of a failed test 105
4.5.5 Entropy of a successful test 106
4.5.6 Total expected entropy of a test 107
4.6 Terminating a diagnostic session by test biasing 108
4.7 Heuristics 109
4.8 Examples 109
4.9 Discussion 113
4.10 Review of some existing test optimisation mechanisms 114
4.10.1 Arby's testing methodology 114
4.10.2 Sophie's ad-hoc half split method 115
4.10.3 Shirley's testing method 115
4.10.4 DART 116
4.10.5 General Diagnostic Engine (GDE) 116
4.11 Summary 116

CHAPTER V: MAPPING THE PROBLEM SOLUTION TO THE ARCHITECTURE 118
5.1 Introduction 118
5.2 Fault diagnosis: the ATMS perspective 118
5.2.1 ATMS reminder 118
5.2.2 Mapping the problem solution onto the ATMS 119
5.2.3 Optimising the creation of the assumptions 124
5.2.4 Optimising the mapping by creating negative assumptions only 124
5.2.5 State changes and the ATMS 125
5.2.6 Summary 125
5.3 Problem solving; the blackboard perspective 125
5.3.1 The communication level 125
5.3.2 The translation level 126
5.3.3 The filtering level 126
5.3.4 The combination level 128
5.3.5 The testing/diagnosis level 129
5.3.6 Overall control 129
5.4 ATMS and blackboard cooperation 131
5.4.1 The normal data path through the ATMS/blackboard environment 131
5.4.2 The state change data path through the ATMS/blackboard environment 132
5.5 Other issues 132
5.5.1 Obtaining continuous diagnosis 132
5.5.2 Dealing with the knowledge sources 135
5.6 Conclusion 135

CHAPTER VI: IMPLEMENTATION ISSUES,PRELIMINARY VALIDATION AND FUTURE WORK 136
6.1 Introduction 136
6.2 Shell implementation status and issues 136
6.3 Performance issues 137
6.3.1 Fault isolation algorithm characteristics 137
6.3.2 Average number of iterations 139
6.3.3 Minimal candidate generation cost (t3) 145
6.3.4 Calculating the cost of the test evaluation algorithm (t4) 148
6.3.5 Average total time taken to perform diagnosis 150
6.3.6 Issues of parallelism 151
6.3.7 Maximum time limit for a diagnosis 151
6.3.8 Time taken to deal with state change suspicions 151
6.3.9 Performance-enhancing heuristics 152
6.4 Computer networks 152
6.4.1 Network definition 153
6.4.2 Network structure 153
6.4.3 Network architecture 153
6.4.4 Network topologies 154
6.4.5 Routeing 157
6.4.6 Network testing tools 157
6.4.7 Important network characteristics 158
6.5 Deep knowledge translation of routeing failures into suspect components 158
6.5.1 Symptom information and routeing 159
6.5.2 The statics: Describing the network structure 160
6.5.3 The dynamics: Describing the routeing behaviour 162
6.5.4 Using the network structure and behaviour description to generate suspects 165
6.5.5 Illustrative runs 166
6.6 Future work on the validation of the proposed shell 176
6.7 Conclusion 178

CHAPTER VII: CONCLUSION 179
7.1 Thesis summary 179
7.2 Critical evaluation 180

REFERENCES 184

APPENDIX 1 189

GLOSSARY 192

List of Figures

1.1 Diagnostic cycle 12

2.1 Architecture of rule-based systems 23
2.2 If .. then .. rule example 24
2.3 A frame node 26
2.4 Frame representation 27
2.5 Blackboard architecture 29
2.6 TMS approach to problem solving 33
2.7 The proposed blackboard architecture 44
2.8 Level control procedure 45
2.9 ATMS/Blackboard correspondence 48
2.10 Basic ATMS 49
2.11 Split environment ATMS 50
2.12 ATMS with separate consumer environments 51

3.1 Cluster approach to multiple faults 62
3.2 Mapping to a Bayesian network 63
3.3 Relevant candidates after measurements 66
3.4 Minimal candidates after measurements 67
3.5 Typical dynamic system 81
3.6 Testing cycle 82
3.7 Possible testing sequences 82
3.8 A simple system 83

5.1 State change algorithm and the blackboard levels 130
5.2 Blackboard/ATMS interaction 133

6.1 Layers, protocols and interfaces 154
6.2 Computer network topologies 156
6.3 Black box behaviour 159
6.4 Hierarchical decomposition of a system 160
6.5 Computer network 162
6.6 Simple network topology 163
6.7 AND/OR problem decomposition 165
6.8 The simulated network 168

List of Tables

2.1 Relative merits of rule, frame and blackboard systems 31
2.2 TMS features 34
2.3 Knowledge source format 46

3.1 Combination example 1 76
3.2 Combination example 2 77

4.1 Entropy example 1 110
4.2 Entropy example 2 111
4.3 Entropy example 3 112

5.1 Label update/Minimal candidate algorithm duality 122

6.1 Required number of iterations 144
6.2 Total cost of minimal candidate operations 147
6.3 Total cost of basic test evaluation for the average test 149
6.4 Total cost of improved test evaluation for the average test 150

CHAPTER I

INTRODUCTION

1.1. Fault diagnosis: a perspective

Fault diagnosis is a process whose goal is to determine the fault or faults responsible for a set of symptoms in a particular system [Gene84]. To achieve this goal, diagnostic systems repeatedly loop through a three stage cycle [John85]:

Stage 1: Behavioral observations of subsets of the system under observation (SUO) are translated into possible explanations for that behaviour, sometimes called candidates [Davi84] , [Klee87]. Unexpected behaviour is translated into subsets of suspect elements, expected behaviour into subsets of "presumed working" elements;

Stage 2: The new explanations for the latest observed behaviour of the SUO, obtained from the symptom translation phase, are combined with the existing explanations for the previously observed behaviour [Will87], giving a new set of possible explanations for the observed behaviour of the SUO. Normally the set of possible explanations for the SUO's behaviour will be a set of alternative fault patterns, each of which can explain all the current symptoms. The combination is normally performed within the confines of a fault model;

Stage 3: The current set of possible explanations is analysed, and new measurements (tests) which discriminate between the current explanations are suggested and performed. Usually the best next measurement is the one that will on average lead to the discovery of the faulty set of components in the minimal number of steps [Klee87].

This cycle is repeated until a satisfactory explanation for all the symptoms is achieved. It is summarised in figure 1.1.

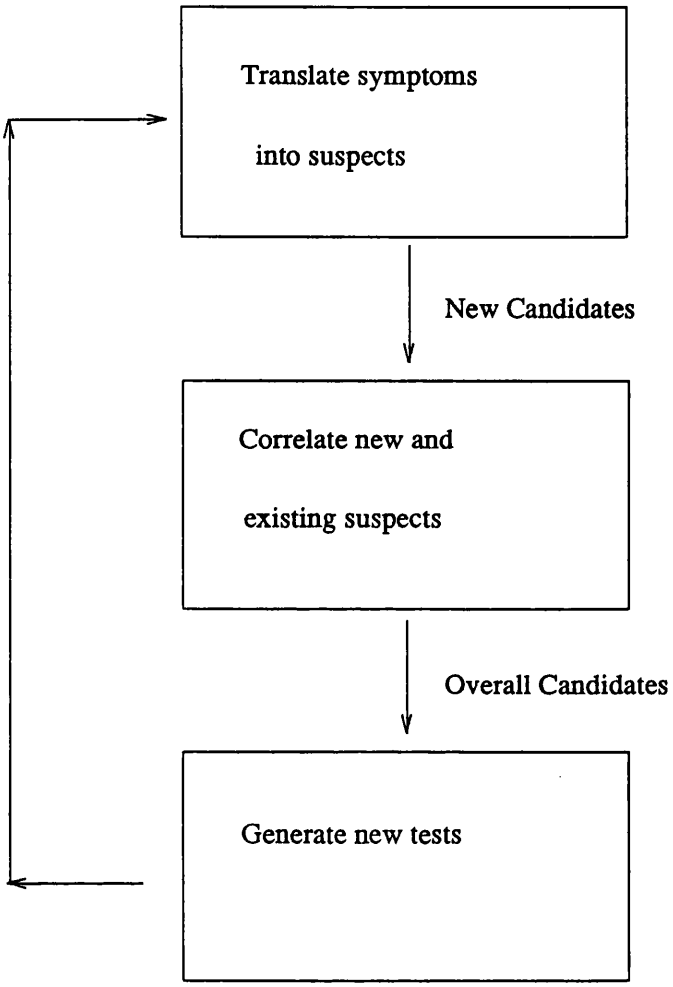


FIGURE 1.1 Diagnostic cycle

Fault diagnosis can be applied to a wide variety of systems, ranging from simple, static systems such as electronic circuits to complex, dynamic ones such as computer networks. The complex dynamic systems represent the most general category of systems; they are defined as systems consisting of a large number of interconnected components, each of which may change state while being observed. From a diagnostic point of view, a component can be either in a working state or in a faulty state. Normally, the sensors used to observe a complex dynamic system have to be considered as part of the system.

Because of the inherent complexity of diagnostic problems [Weis83], the expert system approach is particularly suitable and has been used extensively.

1.2. Aim of the research

The aim of this thesis is to provide a fault diagnostic shell for complex dynamic systems which contains all the elements of fault diagnosis which are system independent. In other words, the shell presented in this thesis supports elements of the fault isolation process which are common and can be generalised to all complex dynamic systems. This system independent shell can then be augmented with domain-specific knowledge every time it is applied to a particular system. The main elements of the dissertation include:

(a) The design of a general, high level architecture suitable for fault isolation in dynamic systems. In the development of the architecture, the main design goals were to make it flexible enough to adapt to all the different stages of fault diagnosis, and to provide as much support as possible to the problem solving requirements, giving the problem solver an opportunity to concentrate solely on the diagnostic reasoning process.

(b) The design of a general fault model for complex dynamic systems. In the development of the fault model, the main design goal was to have a fault model that is sufficiently general and unconstrained to be applicable to any complex dynamic system. Therefore, the model had to be able to deal with single faults, multiple faults, sensor faults and state changes in the system during the fault diagnosis process, all of which can occur in a complex dynamic system.

(c) The design of a general testing strategy for complex dynamic systems. Again the design goal was to build a testing algorithm that is applicable to any dynamic system, and that adapts to changes in the important parameters of the system, rather than relying on the static characteristics of one particular system to decide the optimal tests.

(d) The integration of the proposed fault model and test generation strategy with the high level architecture to provide a fault diagnostic shell for complex dynamic systems. The aim here was to show the compatibility of the different design elements and provide a unified tool.

(e) The preliminary validation of the proposed diagnostic shell, including an implementation, performance evaluation and preliminary testing in a simulated environment.

The specific domain chosen for the preliminary validation is that of computer networks and the symptom data relates to connection failures in a computer network. The aim here was to build a research prototype.

1.3. Motivation for the research

Traditionally, computational tools' progress has followed a two stage cycle of development. In the first stage a system specific solution to a particular problem is worked out. In the second stage, a general solution, sometimes based on system specific solutions, applicable to a whole class of problems is developed. This pattern already appeared in the earliest computational tools; Pascal and Leibnitz developed a mechanical calculator to deal with the four basic mathematical operations before Babbage suggested the creation of the first general-purpose computer [Zwas81]. More recently, the pattern was followed in several computer fields. In the development of computer languages, programming in binary code was replaced by by more general and higher level languages such as FORTRAN, COBOL etc. In operating systems, early system specific operating systems such as Burroughs' Master Control Program for its B5000 computer [Deit83] were developed before portable, system independent operating systems such as UNIX. In the development of data storage, system specific filing and data access methodologies have been replaced by much more general databases [Mart77]. The reasons for this development cycle are clear; it is easier to work out a problem specific solution and then generalise it to a whole class of problems than to create a general solution for a class of problems that has never been attacked before. Furthermore, general tools normally require much more computer resources, which tend to become cheaper as time goes by.

The one computing discipline where the reverse approach was attempted is artificial intelligence. The first goal of artificial intelligence in the 1960s was to develop general problem solving mechanisms that could solve a whole range of problems, such as the General Problem Solver [Erns69]. In the early 1970s, researchers in artificial intelligence concentrated on general methods for improving problem representation and the search for the solution [Wate85]. Neither of these two periods produced any major breakthroughs; they were as it were "putting the cart before the horse" by immediately attacking the most general problem without attempting to first gain experience in a limited subfield.

It is not until the late 1970s that artificial intelligence came back to the natural order of the development cycle, by producing highly specialised programs attempting to imitate

human behaviour in a very narrow field. These programs were called expert systems and this period corresponds to the expert systems' vintage period. It produced systems like MYCIN [Shor76], PIP [Pauk76], CASNET [Weis74], INTERNIST-I [Popl75] all assisting medical diagnosis, PROSPECTOR [Duda79] assisting geological exploration and DENDRAL [Lind80] analysing chemical molecular structures.

The development of the expert system field since then has been much more conventional and has broadly followed the normal trend towards generalisation. First, the problem solving methodologies of the initial systems, implicit within each system, were generalised to problem solving environments, called shells, which can be applied to a whole class of problems. For example, MYCIN was generalised to EMYCIN [VanM79], CASNET lead to EXPERT [Weis79] and PROSPECTOR lead to KAS [Rebo81] Second, new expert system tools were developed, such as OWL [Szol77], KEE [Kunz84], KBS [Redd82], HEARSAY-III [Balz80] and OPS5 [Forg81] all of which offer particular knowledge representation techniques and specific problem solving techniques. Finally even more general aids have been built, such as ADVISE [Mich83] and AGE [Nii79] which offer a choice of knowledge representation techniques and a choice of problem solving techniques.

As far as fault diagnostic expert systems are concerned, the movement towards generalisation can be analysed from the point of view of the three phases of the fault diagnostic cycle; the translation of symptoms into suspects, the combination of the suspect elements and the requests for new tests.

The translation of symptoms into suspects has been the major research area of diagnostic expert systems. All the early diagnostic expert systems (which were concentrating on medical diagnosis) such as the already mentioned MYCIN, PIP, CASNET and INTERNIST-I used shallow knowledge to perform this translation. *Shallow knowledge* is surface knowledge in the sense that rules tend to be heuristic rules of thumb [Cohn85] of the type: "if behaviour A happens, then in 90 % of the cases, such and such components will have failed" [Reit87]. This knowledge is strictly specific to the application in which it is being used and cannot normally be transferred to another system.

Subsequent expert diagnostic systems, mainly in the domain of troubleshooting hardware systems and electronic circuits, such as DART [Gene84], CRITTER [Kell82], IDT [Shub82] GDS [Klee87], Davis' diagnostic system [Davi84] and Hamscher's diagnostic system [Hams83] used deep knowledge about the system under observation to translate

the symptoms. *Deep knowledge* about a system generally means that the system is described through its structure and its behaviour; in such a case the translation of symptoms into suspect components can be performed by running the deep knowledge description of the system which will yield the system's expected behaviour and comparing it to the system's actual behaviour. Compared to shallow knowledge, deep knowledge represents a generalisation because it embodies a *methodology* for translating symptoms into suspects, applicable to all systems having similar structure and behaviour. Thus all the deep knowledge systems mentioned in the last paragraph incorporate a language for describing the structure and behaviour of digital electronic circuits. However deep knowledge methodologies and languages are currently confined to systems with similar structure and behaviour. For example, a deep knowledge approach suitable for electronic circuits is not suitable for medical problems.

The combination of the suspect elements in the early expert systems in medical diagnosis and electronic troubleshooting such as MYCIN, PIP, DART and CRITTER was made under the simplifying assumptions that there is only one fault in the system, that there are no intermittent faults or state changes and that the sensors cannot go wrong. More recently, expert systems like NDS [Will83] and Williamson's diagnoser [Will87] have attempted to develop system dependent methodologies for dealing with multiple faults. NDS's approach depends on the ability to repair a component as soon as it is found to be faulty, and Williamson's depends on shallow knowledge about the nature of the faults. [Klee87] , [Reit87] and [Peng86] have developed a system independent methodology for dealing with multiple faults in static systems, which takes into account only the results of failed tests. Sensor faults are considered in only a handful of expert systems [Fox83] , [Chan87]. State changes in the system under observation during diagnosis and intermittent faults are apparently allowed in NDS and Williamson's diagnoser but the literature does not explain how they are dealt with. We have found no existing system whose fault model includes reasoning about intermittent faults and state changes in the system during diagnosis.

As far as test selection optimisation is concerned, very few general, completely system independent techniques have been developed, none of them suitable for complex dynamic systems. Some systems such as CRIB [Addi80] (computer faults), ACE [Veso83] (telephone network faults) and FALCON [Ches84] (process control in chemical plants) assume that all the fault reports are already available and limit themselves to analysing those reports. Some diagnostic expert systems in computing and engineering, such

as IDT and DART couple their testing strategies very tightly with a deep knowledge description of the system under observation, thus making the testing algorithm dependent on their particular methodology for translating symptoms into suspects. Furthermore, they do not attempt to optimise the test choice; they just try to generate a reasonable test. Other diagnostic expert systems in computing and engineering such as PDS [Fox83] use shallow knowledge testing methods provided by an expert. SOPHIE, [Brow82] which attempts to split the list of suspects into two sublists of equal probability is geared towards systems with single faults, IN-ATE [Cant83] uses a multi-step lookahead procedure (based on the gamma miniaverage methodology [Slag71]) which is computationally expensive and GDS [Klee87] adapts the entropy measure to test optimisation in a way which is mainly suited to static systems such as electronic circuits.

Of all the systems mentioned in the review of diagnostic expert systems, GDS stands out as the only system which combines a (deep knowledge) translation of symptoms into suspects with a reasonably general fault model incorporating single and multiple faults and a system independent testing strategy.

The first observation that can be made is that both shallow and deep knowledge diagnostic systems concentrate mainly on the first part of the diagnostic cycle, namely the translation of symptoms into suspects. In fact the name deep knowledge (and shallow knowledge) diagnostic systems is misleading; the deep knowledge normally only applies to one of the phases of the diagnostic cycle. The fault models are almost incidental, and are just used to prove (in the case of deep knowledge models) that the suspect generation phase is correct. The simple fault models would certainly be insufficient for dynamic systems; they are also probably insufficient for the domains in question since the single fault assumption is not necessarily true [Hams83]. The non-intermittency assumption and the no-sensor-fault assumption also mean that the whole diagnostic session is monotonic [Klee87], once data are received there is no need to check, revise or reject them.

The second observation that can be made is that there are currently no general diagnostic systems applicable to complex dynamic systems, which is the most general category of systems. Most of the recent research effort towards generalisation has gone into the development of deep knowledge representation techniques for the translation of symptoms into suspects in static systems. This is ironic, since that is probably the most system dependent part of the diagnostic cycle.

This thesis fills the gap in fault diagnostic environments for complex dynamic systems. The approach adopted here differs from most existing approaches. First an attempt is made to redress the balance between the different phases of the diagnostic process by putting much more emphasis on the fault model; the right way to build a diagnostic system should be to decide first on a fault model and then decide on the suspect generation, not the other way round. Second, much more emphasis is put on the test selection algorithm by trying to optimise it. Third, because the research covers all the phases of the diagnostic cycle, much more emphasis is put on the global architecture of the diagnostic system. Fourth, there is a complete separation between the different phases of the diagnostic cycle. Of all the systems mentioned, only GDS uses a comparable approach. However GDS is geared towards approximately static problems such as electronic circuits troubleshooting; the fault model, architecture and testing algorithms developed here apply to the more general category of complex dynamic systems. Finally the proposed diagnostic environment can accept both deep and shallow knowledge based suspect generation.

1.4. Thesis outline

The thesis is divided into seven chapters. The present one has explained the aims and motivations of the research.

In Chapter 2, a new high level architecture suitable for fault isolation in dynamic systems is proposed. The architecture is based on the coupling of a blackboard with an Assumption-based Truth Maintenance System (ATMS). The Chapter first describes the motivations for using a blackboard and an ATMS. Then a blackboard configuration suitable for fault diagnosis is developed. This is followed by an explanation of the interface between the blackboard and the ATMS. Finally, some existing architectures are examined and the the advantages of the proposed architecture are indicated.

In Chapter 3, a new methodology for diagnosing faults in complex dynamic systems is proposed. The Chapter first defines the requirements of fault models for dynamic systems, namely the ability to deal with multiple faults, sensor faults and state changes. Then the existing multiple and sensor fault strategies are evaluated and a new unifying mechanism performing probabilistic diagnosis simultaneously on multiple and sensor faults is proposed. The second part of the Chapter introduces a new methodology for dealing with state changes in the system during diagnosis. The methodology defines criteria for suspecting state changes, mechanisms for confirming state changes and optimal

algorithms for recovering from state changes which only reject suspect data. The Chapter is concluded with a summary of existing temporal logics and a discussion of the reasons why they are not appropriate for dealing with state changes in dynamic systems.

In Chapter 4, a new methodology for optimising the selection of the next test to perform, based on the entropy function is proposed. The chapter begins with an examination of existing test selection techniques and points out their shortcomings. Next, the entropy concept is briefly covered. This is followed by the development of the algorithms which show how tests can be evaluated and compared by calculating their expected entropies. Finally examples are given to show that the entropy methodology is not only mathematically optimal but also chooses tests which are intuitively appealing.

Chapter 5 shows how the fault model and the testing methodology can be integrated with the blackboard/ATMS architecture to provide a fault diagnostic shell. The Chapter begins by showing how a diagnostic problem is mapped onto the ATMS in an optimal way. Then the diagnostic problem is discussed from the blackboard point of view, in terms of the knowledge sources that form part of the system independent diagnostic shell and in terms of the control mechanisms. Subsequently, the cooperation of the blackboard and the ATMS during the diagnostic process is analysed and the proposed shell extended to deal with continuous diagnosis.

Chapter 6 performs the preliminary validation of the proposed system. and suggests directions for further validation and future work. It begins with a brief discussion of the implementation of the proposed diagnostic shell, followed by its performance evaluation. The next section briefly introduces the field of computer networks, which was the domain chosen as the preliminary testing ground. This is followed by the description of a knowledge source which was used to simulate the translation of symptoms into suspects; the knowledge source translates failed connections between two points of a computer network into a list of network components that may have caused such a failure. Then sample runs from the simulation are given. Then the limits of this preliminary validation are discussed together with the additional steps required for further validation. Finally possible directions for future research and work are given.

Chapter 7 presents the dissertation's conclusions.

Finally, Appendix 1 contains two proofs for the entropy calculation in Chapter 4.

CHAPTER II

A HIGH-LEVEL ARCHITECTURE FOR FAULT ISOLATION IN COMPLEX DYNAMIC SYSTEMS

2.1. Introduction

The need for Knowledge-Based Systems (KBS) in tasks performed in complex dynamic systems is well documented [Sevi83] , [Wate85] , [Weis83]. The complexity of such tasks, the shortage of human expertise, the lack of general theory, the variability of the data structures make them ideal targets for the KBS approach. This chapter proposes a high-level architecture particularly suitable for building a knowledge-based system for fault isolation in complex dynamic systems. The first part of the chapter analyses the characteristics of fault isolation in complex dynamic systems and concludes that the blackboard model is the most suitable high-level architecture for solving this type of problem. The second part of the chapter proposes a new method for turning the theoretical blackboard model into a practical computational entity, based on the coupling of a blackboard with an Assumption-based Truth Maintenance System (ATMS). Subsequently, a new architecture for fault diagnosis in complex dynamic systems is developed. This architecture is based on the blackboard/ATMS coupling and the splitting of the blackboard into layers corresponding to the different stages of fault diagnosis. The interface between the ATMS and the blackboard and their respective roles within this architecture are also analysed.

2.2. Major characteristics of fault isolation in complex dynamic systems

The choice of a problem solving model for a particular task is closely related to the characteristics of the task to be performed. The major characteristics of fault isolation in complex dynamic systems and their implications for the ideal problem solving architecture are as follows.

Diversity of knowledge: The task of diagnosing faults is a complex task which needs to integrate a large number of independent sources of knowledge such as knowledge about particular types of faults, knowledge about test generation, knowledge about the combination of the different findings, knowledge about the communication with the sensors etc. These knowledge sources may take many forms such as algorithmic or heuristic, deep knowledge based or shallow knowledge based, procedural or declarative. This suggests the need for an architecture that facilitates cooperation between many diverse knowledge sources.

The need to cooperate with other tasks: The fault management task is usually not the only task that is performed on a complex system; other tasks may include managing the topology of the system, analysing the performance of the system etc. In systems comprising more than one task, the various tasks need to cooperate; for example the topology manager should provide information about the expected topology of the system to the fault manager, and the fault manager should provide fault data to the performance manager. Therefore, the architecture should facilitate data exchange with other tasks.

Irregularity of data arrival: In a dynamic environment, the data about faults and the state of the system do not arrive in any predetermined way. Faults are reported as they are discovered and the reports may take a variable time to reach the module responsible for fault management. This suggests an architecture that can easily switch control from one task to another (such as from analysing one report to accepting another report) and preferably one that can perform actions in parallel.

Lack of firm theory of diagnosis: There is no algorithm or theory that specifies "the best way to perform diagnosis" in complex dynamic systems. Some aspects, such as test generation, may be associated with a deep knowledge algorithm and deep knowledge mechanisms have been developed for less complex systems such as digital circuits [Davi84] , [Gene84] , [Klee87]. However the number of possible states and interactions means that diagnosis of complex dynamic systems always has an element of trial and error in it. This again suggests a flexible type of control that can be easily adapted or tuned to attempt a different approach.

Inconsistency of data: Because the system is dynamic, there is no guarantee that the state that was true at the beginning of a diagnosis is true later. If the state of the system does change, sensor results that were obtained initially may be in contradiction with subsequent sensor findings. This variability of data and states over time suggests a system that can spot inconsistent data and deal with it, i.e a system that has access to all the relevant data at the same time, and that can easily retract its findings and change its focus of attention.

Errors in data: Because the sensors are normally themselves part of the complex system under observation, they may go wrong [Fox83]. Therefore the data sent by the sensors may be false. This sensor data property reinforces the desirable system characteristics mentioned in the previous paragraph.

Changes in the system: A dynamic system changes over time; for example new sensors may be added. The architecture should therefore allow the incorporation of new knowledge sources without any major changes. This suggests an architecture that facilitates the incorporation of new knowledge and has a flexible control mechanism.

Explanation of reasoning: A system that helps a human being or even a fully automated diagnoser needs to be able to explain its conclusions. This suggests a system that facilitates the retracing and the justification of the steps taken during diagnosis.

The need to find all the faults: In many problem spaces, the point of the search is to try and locate one solution to the problem. For example R1 [Breu76] tries to find one good configuration for a proposed computer system. In fault isolation, the point of the search is to find all the solutions, that is all the faulty points. If there are several concurrent faults, isolating just one of them is an insufficient partial solution. We therefore need an architecture that facilitates the search for all the faults in a system.

The ideal architecture therefore:

1) facilitates cooperation between the different knowledge sources belonging to the task at hand;
2) facilitates data exchange with other tasks;
3) moves easily the focus of attention between different search space points;
4) allows parallel processing of the data;
5) allows easy adaptation to a different diagnostic strategy;
6) can deal with contradictory findings by for example easily retracting its findings;
7) facilitates the incorporation of new knowledge;
8) has no predetermined flow of control;
9) is able to explain its line of reasoning;
10) facilitates the discovery of all the solutions;
11) allows easy access to all the partial solutions; this is necessary to facilitate the evaluation of these partial solutions and to facilitate the switching of control between them (context evaluation and context switching).

2.3. Choosing a high-level architecture for fault isolation in complex dynamic systems

The previous section defined the criteria for choosing a problem-solving architecture for fault isolation in complex dynamic systems. Here, the major architectures currently used in knowledge-based systems are described and evaluated according to those criteria.

2.3.1. Rule- based systems (RBS)

Rule-based systems are currently the dominant technique for representing knowledge and making control decisions. They have been used in various fields, including equipment maintenance [Shub82] , fault isolation [Veso83] , process control [Nels82] and geology [Smit83].

Overall architecture

Rule- based systems have three major components: an inference engine (the control), a working memory which contains the case specific facts and a static, long term memory which contains the rules and facts pertinent to the system (the knowledge base). The inference engine builds the current state of the problem solving cycle in the working memory using the knowledge base and dynamic inputs as depicted in Figure 2.1.

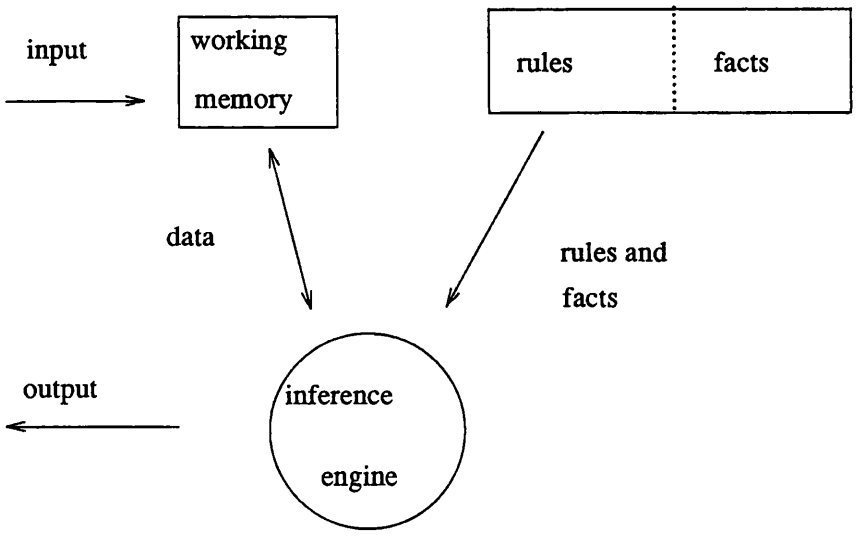


FIGURE 2.1 Architecture of rule-based systems

Knowledge representation

RBSs normally incorporate practical human knowledge in conditional if-then rules [Haye85]. A typical rule from the REACTOR system is shown in Figure 2.2.

IF: The heat transfer from the primary coolant system to the secondary coolant system is inadequate
and the feedwater flow is low

THEN: The accident is loss of feedwater

FIGURE 2.2 If .. then .. rule example

The pool containing the set of these rules and the set of long term facts pertinent to the system is stored in the knowledge base. The case specific facts, relevant to the case that is in progress are stored in the working memory.

Control mechanism

The rules in a RBS are normally used in either or both of the following ways [Weis83] , [Haye85]:

- in forward chaining or event driven reasoning the system starts with a subset of evidence (i.e a small set of facts). These facts satisfy the antecedent ("if") parts of some of the rules which can then be fired (executed), adding the "then" part of the rule to the pool of knowledge. This invocation of the rules proceeds until a satisfactory conclusion has been reached or until no further rules can be invoked;
- in backward chaining the system starts with an initial set of goals. The rules with matching consequents are then examined, and the unmet conditions of the antecedents are defined as the new goals. The system then recursively tries to satisfy these subgoals, until the initial goal has been reduced to a set of subgoals which can be directly satisfied from given information.

This problem solving behaviour is embedded in the inference engine.

Evaluation

While RBSs provide fairly powerful means of data representation and inference, they have several limitations [Bigh87]:

1 They have a fairly rigid form of control; both backward and forward chaining systems are at their best when the links are followed sequentially (either when creating an inference chain or backtracking to change some decision previously made). They do not normally provide control mechanisms to jump opportunistically to some other point in the search space. This lack of control flexibility means that:

   (a) It is more difficult to volunteer information out of sequence or accept new sensor readings while the current ones are being processed; both these actions correspond to opportunistic jumps in the search space. RBSs will normally process a piece of data completely before they are ready for the next one.

   (b) It is difficult to vary the diagnostic strategy. RBSs inference engines are normally centred around an embedded problem space search strategy and varying it can be quite a major task.

   (c) It is difficult to adapt the system to parallel processing because the model does not suggest any natural partitioning of the search space. This is because the search space is seen as uniform and monolithic. Of course, it would be possible to search in parallel all the branches of an RBS's search tree, but this is not 'intelligent' parallel processing.

   (d) It is difficult to apply one search strategy to a portion of the search space and a different one to a different portion. The model is geared towards one search strategy for the whole problem space and most changes in the inference engine will have global implications.

   Points (c) and (d) are influenced by the fact that RBS's normally view the whole search space as a flat graph and perceive their task as optimised graph search. This corresponds to the founding paradigm of Artificial Intelligence that problem solving is search [Enge88], which is a fairly narrow view of Artificial Intelligence.

2 They find it difficult to deal with inconsistent data; the RBSs normally provide the means to search the problem space of consistent data, but an inconsistency implies the need to switch between two portions of the problem space, or a parallel search of separate portions of the problem space both of which correspond to a more opportunistic control structure.

3 They do not provide obvious access to the partial solutions by knowledge sources or tasks that are outside the RBS. While it is true that the partial solutions are held in the working memory, RBSs are not normally geared to sharing this information with foreign sources.

As a conclusion, RBS's do not use sufficiently the "divide and conquer" policy which is beneficial in system design. The only hierarchy in the knowledge about the system is its splitting into rules and facts. This lack of structure in the knowledge space is also reflected in the design of inference engines; they tend to treat the whole search space in a uniform fashion.

2.3.2. Frame based systems

Knowledge representation

A frame is a network of nodes and relations organised in a hierarchy where the top nodes represent the general concepts and the lower nodes more specific instances of those concepts. At each node, the concept is defined as a collection of attributes (e.g name, size etc) and values for those attributes (e.g Smith, small etc) Each attribute is known as a slot. Slots can also have procedures attached to them, which may be executed every time a slot is accessed. Typically two types of relationships exist between nodes; member links where an object is linked to a class it belongs to (eg Truck1 is a truck) and subclass links where a subdivision of a particular class is linked to that class (eg a truck is a vehicle) [Fike85] , [Wate85]. A typical node and frame are depicted in Figures 2.3 and 2.4.

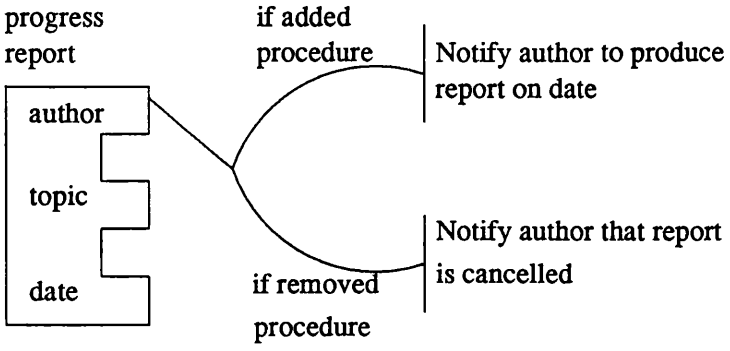


FIGURE 2.3 A frame node

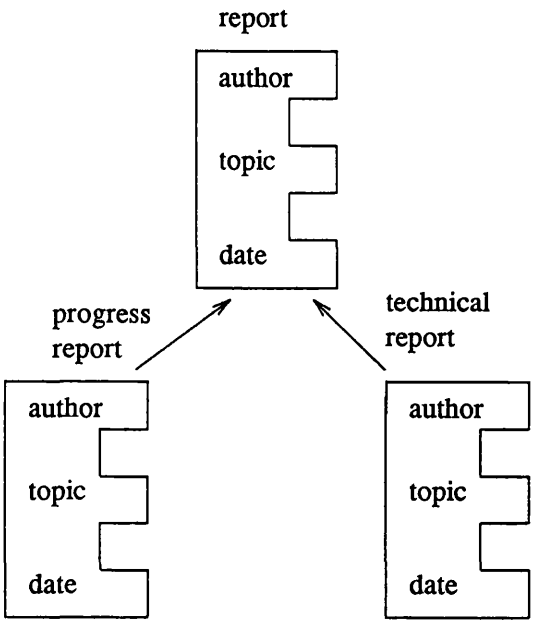


FIGURE 2.4 Frame representation

Control decisions

Most of the work on frames has focused on their use as a method of representation of data (e.g [Bobr77] , [Brac85] , [Stef79]) rather than on their use for the control of reasoning. In such systems, the frame facility becomes simply a database of the knowledge system; the control decisions are made by other parts of the system. However it is also possible to use the frame structure to make control decisions:

Inheritance: The hierarchical structure of frames makes it easy for nodes located lower down in the description hierarchy to automatically inherit the properties of nodes located further up. For example, if all objects in the class "vehicles" have 4 wheels, element "Truck1" which is a vehicle automatically obtains the attribute "has 4 wheels".

Procedural slots: As seen above, it is possible to attach procedures to a slot or a node, which can be executed every time that slot or node is accessed. These procedures can be used to guide the behaviour and thus perform the control of the system. By extension, it is also possible to have full control frames which incorporate procedures only.

Representing rules as frames: "If .. then" rules can be represented as frames by (for example) having the conditional part of the rule in a condition slot and the action part of

the rule in an action slot. This can then be used to group rules into classes and into a hierarchical taxonomy, thus allowing different control strategies for different classes of rules. This is an improvement over pure rule-based systems which do not allow such differentiation.

Overall architecture

Generally in a frame based KBS, control information (i.e the inference engine) and the description of the objects of the system (the knowledge base) are interspersed within the frame hierarchy. The frames model the generic entities and the reasoning process instantiates the frames for an actual case. In other words, the frame slots are empty at the beginning of the reasoning process whose function is to fill the slots with current values, using procedures, inheritance rule application or any other form of reasoning.

Evaluation

The frame organisation is quite a powerful methodology particularly useful in domains where expectations about the form and content of the data play an important role in problem solving, such as classification systems which attempt to classify data according to their type [Wate85]. In such applications, the control path will tend to follow closely the hierarchical frame structure. However, from the point of view of the evaluation criteria developed previously, the frame organisation has several drawbacks:

- there is no clear separation between the control information and the knowledge base; both of them are interspersed in the frame hierarchy. This makes it much more difficult to change the control strategy since it is to a certain extent dependent on the taxonomy of the system under observation;
- the frame model does not suggest any obvious way of performing parallel processing of the data;
- the data is normally stored across the whole frame hierarchy; this makes simultaneous access to data stored in different portions of the frame hierarchy more difficult. It also hinders the exchange of data with outside tasks.

On the whole, frames can be quite a rigid organisation because of their requirement for a hierarchy in the solution space.

2.3.3. The blackboard framework

Overall architecture

The blackboard architecture normally consists of of three major components [Nii86]:

- A general repository of data called *the blackboard* which contains all the problem-solving state data. This blackboard is often divided into several *levels* (or layers), each of which contains data relevant to one particular facet of the solution.

- A set of knowledge sources, representing the expertise relevant to the problem, whose role is to manipulate the state data on the blackboard and which cooperate via the blackboard in the search for the solution.

- A control entity, whose role is to implement a problem-solving strategy.

The overall blackboard architecture is thus as described in Figure 2.5.

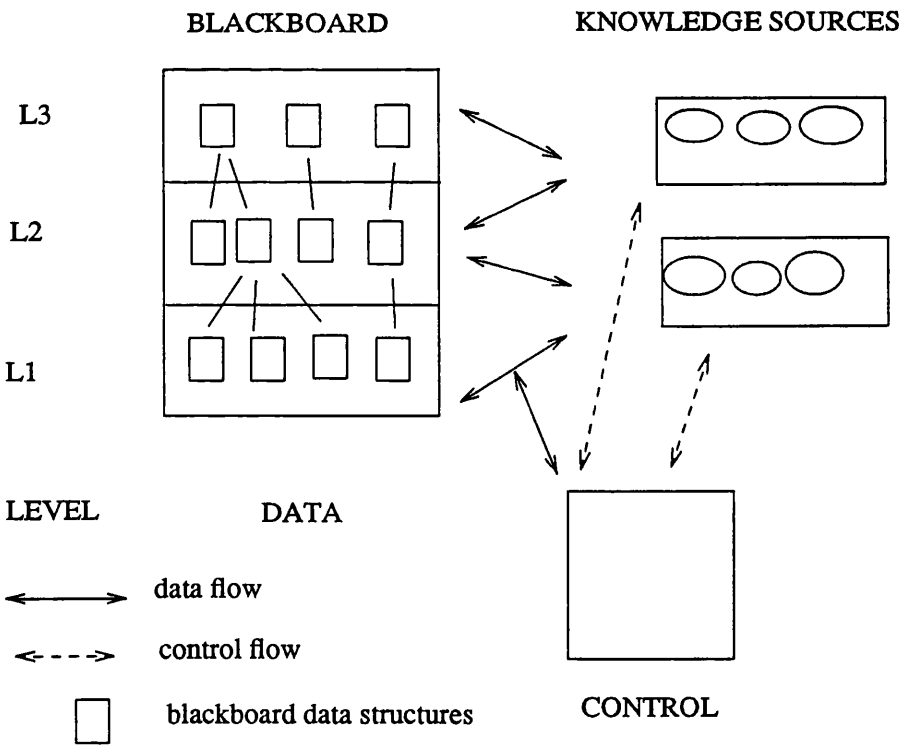


FIGURE 2.5 Blackboard architecture

Knowledge representation

The knowledge representation in a blackboard has two aspects; the representation of static, long term knowledge about the problem and the representation of dynamic knowledge corresponding to the current knowledge about the actual case in progress. The dynamic repository of data is the blackboard, containing the partial solutions achieved so far (i.e recording the state of the search so far). In the blackboard model, the long-term domain knowledge is encoded within a set of independent knowledge sources. Each of these knowledge sources is a specialist for some specific aspect of the problem. Within each knowledge source, the knowledge representation can take any form, such as algorithmic, heuristic etc. Each knowledge source has access to the blackboard. It is triggered by (i.e can contribute to) a particular type of data on the blackboard and if requested transforms it to give new data. The knowledge sources always take their data from the blackboard and always return the enhanced data to the blackboard. If the blackboard is divided into levels, each knowledge source is linked to one or more specific levels.

Control decisions

The role of the control is to coordinate the activities of the knowledge sources and guide the search for a solution. The control entity does this by determining the focus of attention that indicates the next thing to be processed. The focus of attention can be the knowledge sources (which knowledge source to activate next) or the blackboard (which data objects on the blackboard to pursue next). The control module can use any of a number of problem-solving strategies: backward chaining, forward chaining, request driving (that is trying to provide information requested by the knowledge sources) etc.

Evaluation

The blackboard model potentially offers all of the following:

- the possibility of easy change of focus and easy change of diagnosing strategy; this is because the model does not specify either a specific focus of attention or a preset control path;
- easy cooperation between the knowledge sources, with outside tasks, and easy incorporation of new knowledge sources via the blackboard structure where the data can easily be made available;
- easy adaptability to parallel processing since it naturally partions itself into into levels with separate control modules;

- easy retracting of findings since all the data is on the blackboard;
- the possibility of explaining the reasoning since all the data obtained during the reasoning process is available on the blackboard;
- the possibility of finding all the solutions to the problem; this is because the blackboard model is general enough not to put any constraints on the method of searching the problem space.

The blackboard model can therefore satisfy all the criteria for a fault isolation system.

2.3.4. Summary

Of the three major high-level architectures, only the blackboard model satisfies all the evaluation criteria and is thus the one we have chosen for our system. The achievement of the behavioral goals by the alternative architectures is summarised in Table 2.1.

Behavioural goals	Rules	Frames	Blackboard
easy KS cooperation	x	x	x
easy data exchange with other tasks	x		x
easy change of focus			x
potential parallel processing of data			x
easy change of diagnostic strategy			x
easy access to all the data	x		x
good treatment of contradictory data			x
easy introduction of new knowledge		x	x
no predetermined flow of control	x	x	x
recording of the reasoning process	x	x	x
easy path to the whole solution space			x

TABLE 2.1 Relative merits of rule, frame and blackboard systems

2.4. Moving from the blackboard model to a computable framework

The blackboard architecture as presented so far only outlines the organizational principles and is therefore only a model in the sense given by [Nii86]. It does not specify how to translate the model into a computational entity. Several design decisions have to be taken in order to obtain a computable framework.

The knowledge sources

Decisions to be made about the knowledge sources include:

a) the representation of the knowledge sources; in principle any representation (procedural, rules, logic-based etc) can be used;

b) the trigger of each knowledge source; that is the state in which the blackboard has to be for the particular knowledge source to be able to contribute to the general pool of knowledge. Knowledge sources look for specific data on the blackboard;

c) the action of each knowledge source; the action normally results in new data being added to the general pool of knowledge;

d) the interface between the knowledge sources and the blackboard; this interface allows the knowledge sources to write on and read from the blackboard.

The blackboard data

Decisions to be made regarding the blackboard data include:

e) the division (if any) of the blackboard into levels corresponding to the stages of the problem solving process;

f) the objects i.e data to be stored on the blackboard;

g) the relationships between the objects stored on the blackboard, i.e the record of the inference chains followed so far.

The blackboard control

Decisions to be made regarding the blackboard control include:

h) the selection of control modules;

- the organisation of control modules, i.e how they interact, and how the flow of control is formed and maintained;

j) the selection of a focus of attention (i.e the next thing to be processed [Nii86]). The focus of attention can be the knowledge sources, the partial solution data on the blackboard or both. This will determine when and how the knowledge sources are triggered and scheduled and also implicitly how the focus of attention will switch between various points of the search space.

The following section shows how several of those requirements are provided by Truth Maintenance Systems (TMSs), which solve them in a uniform, structured and consistent way. Additionally, TMSs also provide some of the other desirable characteristics introduced in section 2.2.

2.5. Truth Maintenance Systems (TMSs)

2.5.1. The concept of TMS

The methodology of TMSs was first introduced by Doyle [Doyl79] , as a way of improving the efficiency of the search of problem spaces, and especially providing a structured way of searching a problem space. The idea is to divide the problem solver into two separate entities; one only concerned with the rules of the domain (The Problem Solver proper) and the other only concerned with recording the current state of the search (i.e the TMS), as depicted in Figure 2.6.

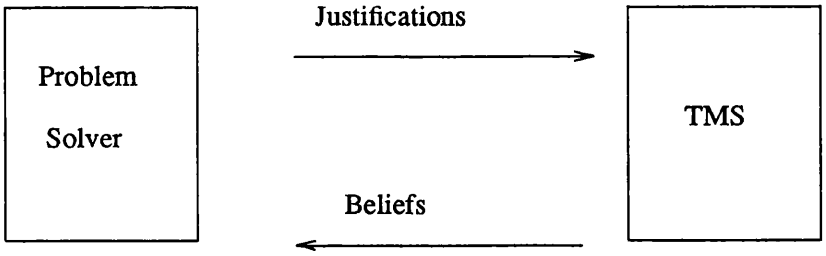


FIGURE 2.6 TMS approach to problem solving

The process of problem solving is seen as being based on making assumptions, making inferences based on existing assumptions and existing inferences, and revising the assumptions and inferences as the problem solving progresses. A point in the search

space is ultimately characterised by the set of assumptions under which it is valid (the context).

2.5.2. Benefits of TMSs

By definition [Klee86], all TMSs provide the following services:

- an interface to the data and inferences contained in the TMS.
- a permanent cache of all the inferences and assumptions ever made, with the inference chains that were followed;
- a method of switching the focus of attention between the search space points;
- a record of the consistency of the data, i.e the TMSs record all the contradictions that have been discovered during the problem solving;
- a method of dealing with inconsistent data (be it by eliminating inconsistencies or tolerating them);

The interface to the data and inferences already made can be used to offer direct support to the interface between knowledge sources and the blackboard (requirement d in Section 2.4). The record of all the inferences, inference chains and assumptions directly supports the need to record the relationships between the blackboard objects (requirement g in Section 2.4). The method for switching the focus of attention directly supports the blackboard control requirement for selecting and switching the focus of attention (requirement j in Section 2.4). The last two services contribute to a good treatment of contradictory data which was one of the ideal architecture characteristics listed in Section 2.2.

Therefore coupling a TMS to the blackboard already provides some of the requested architecture features. The TMS services are summarised in Table 2.2.

- a way of recording the inferences made; - a way of explaining how the inferences where achieved; - a way of retrieving the data; - a method of switching between different points of the search space; - a way of dealing with inconsistent data.

TABLE 2.2 TMS features

2.5.3. Choosing a TMS

Several truth maintenance systems have been developed (TMS [Doyl79] , RUP [McAl80] , [McDe83] , MBR [Mart83] , ATMS [Klee86a] , [Klee86b] , [Klee86]). A detailed appraisal of the different TMSs can be found in [Klee86]. For the purpose of fault diagnosis, the ATMS seems to have the best features, because it provides:

- an explicit representation of the context in which the data is valid: this is done by associating with each piece of data the assumptions under which the data is valid. In other TMSs this context is only implicit. This facilitates the checking of the validity and the consistency of the data and further contributes to the good treatment of contradictory data;
- a way to switch contexts without backtracking: because the contexts under which the data are valid are kept explicitly with the data, switching contexts simply means starting work on another piece of data. In other TMSs, the system has to backtrack via the inference links to return to a previously reached point in the search space and find the context under which it was valid. This avoidance of backtracking facilitates the change of the focus of attention;
- a way to follow mutually contradictory search paths in parallel; because the contexts are explicit there is no reason to insist that all the data in the database are consistent. Other TMSs insist that all the data be consistent, hence all the backtracking to reach a consistent state. The benefits of non consistent data are clear for our purposes; for example if a test t_i finds that element A is faulty, then either t_i is faulty or A is faulty; the ATMS allows both these (mutually exclusive) facts to be present in the database, and to continue the search in both directions. Other TMSs would require the elimination of one of the two from the set of currently believed data. This facilitates the easy access to all the data;
- a way of finding all the solutions at the same time. This is again achieved by allowing searching in multiple contexts; other TMSs can only find one solution at a time.
- the possibility of parallel processing; the ATMS is an inherently parallel searching tool, and potentially allows parallel searching of the different contexts.

All of these characteristics form part of the desirable behavioural goals for the high-level architecture listed in Table 2.1.

2.5.4. Assumption-based Truth Maintenance System (ATMS)

The ATMS is described in detail in [Klee86a] , [Klee86b] , [Klee86]. Here only an overview is given.

Architecture

As all TMSs, the ATMS forms part of a two component reasoning architecture: a problem solver and an ATMS, as depicted in Figure 2.6. The role of the ATMS is to determine belief and disbelief given the justifications supplied so far, and not with respect to the logic of the problem solver. For example, if the problem solver deduced fact A, then made the inference that A implies B and later determined that B is false (all of which were communicated to and recorded by the ATMS), the ATMS would deduce that A is false from the justification that A implies B without understanding the meaning of A and B or the reason why A implied B.

Definitions

The ATMS stores problem solver data in *nodes.* An ATMS node has several fields, described later, one of which is reserved for the problem solving datum. A new ATMS node is created every time a new problem solver datum is communicated to the ATMS.

An ATMS *assumption* is a particular type of node, designating the decision to assume. A new ATMS assumption is created every time the problem solver decides to make an assumption in the world in which it is reasoning.

An *assumed datum* is a problem solver datum that is assumed to hold. It is linked to an ATMS assumption via a justification as will be shown later.

The ATMS thus distinguishes between making a decision to assume (called assumption) and the datum that is assumed (called assumed datum). The main implication of this distinction is that every time the problem solver assumes something, the ATMS will create two nodes; one representing the assumption and one containing the assumed datum. The main reason for this indirection is that it makes it possible to remove (and reuse) unnecessary assumptions by contradicting the assumption, while at the same time keeping a record of the inference chains starting at the assumed datum. It is beneficial to reuse unnecessary assumptions (i.e perform garbage collection) because of the way the assumptions are implemented, as will be seen later.

An ATMS *justification* describes how the node is derivable from other nodes. The justification of a node N is made of three parts. The first part holds the *antecedents* of N, that is the nodes used to justify N. The second part holds the *consequents* of N, that is the nodes in which N was used as an antecedent. The third part holds the Problem Solver's description of the justification, called the *informant* which is used for interfacing with the ATMS as will be explained in the interfacing section.

An ATMS *environment* is a set of assumptions. An environment is *consistent* if its assumptions are mutually compatible, i.e if there is no contradiction when all its assumptions hold.

An ATMS *context* is formed by the assumptions of a consistent environment and all the nodes derivable from those assumptions.

The ATMS is thus provided with assumptions and justifications and its role is to efficiently determine the contexts. To this end, the ATMS associates with every datum a succinct description of all the contexts in which the datum is valid, called the label.

Labels

The ATMS associates a label with every node. A *label* is a set of environments. Every environment E of node N's label is consistent and N must be propositionally derivable from E's assumptions and the current set of justifications given to the ATMS. In contrast to the justification which contains immediately preceding antecedents, the label describes how the datum ultimately depends on assumptions. The label is computed by the ATMS every time a new inference is made. The ATMS guarantees that the label is:

- consistent; a label is consistent if all its environments are consistent, i.e if no environment of the label is a superset of an inconsistent environment;
- sound; a label is sound if the datum is derivable from each environment of the label;
- complete; a label is complete if every consistent environment E in which the node holds is a superset of some environment of the label;
- minimal; the label is minimal if no environment of the label is a superset of any other.

As a consequence, a node is derivable from an environment E iff E is a superset of an

environment of the label. The basic operations on the label will be discussed a bit later.

Basic data structures

The basic data structure of the ATMS is the node, containing a node identifier, the problem solver datum, the label and the justifications:

NodeId:< Datum, Label, Justifications >

There are four types of nodes (distinguished only by their label and justification patterns).

Premises, which hold universally (having the empty environment as justification).

Assumptions, which are self justified; they correspond to the decision to assume, without any commitment as to what is assumed, as described above.

Assumed nodes, which are justified by an assumption; the assumed node contains a Problem Solver assumed datum.

Other nodes, which are derived.

One distinguished node holds all the inconsistent environments, which are called *nogoods.* All the supersets of 'nogood' environments are also inconsistent. The set of 'nogood' environments is known as the 'nogood' database.

For example, let us assume that the problem solver has thus far made three assumptions;

x+y=1 ;

x=0;

x=1.

(The problem solver can make contradictory assumptions, about the value of x in this case, but they cannot co-exist within one ATMS node). As described above, for each one of the Problem Solver's assumptions, two ATMS nodes are created; one corresponding to an ATMS assumption, and one corresponding to an ATMS assumed node (containing the assumed data). The first two assumptions will make the problem solver deduce that y=1 and the first and third assumptions will make the problem solver deduce that y=0. After these deductions, the state of the ATMS is as follows:

1:<A,{{A}},{{1}},{4}>

2:<B,{{B}},{{2}},{5}>

3:<C,{{C}},{{3}},{6}>

4:<x+y=1,{{A}},{{1}},{7,8}>

5:<x=1,{{B}},{{2}},{7}>

6:<x=0,{{C}},{{3}},{8}>

7:<y=0,{{A,B}},{{4,5}},{}>

8:<y=1,{{A,C}},{{4,6}},{}>

Nodes 1,2 and 3 are ATMS assumptions. Nodes 4,5 and 6 are assumed nodes. Nodes 7 and 8 are deduced nodes. The environment {B,C} is a 'nogood', since it corresponds to the assumptions x=1 and x=0, which cannot be simultaneously true. The environment {A,B} corresponds to making the assumptions that x+y=1 and x=1. The environment {A,C} corresponds to making the assumptions that x+y=1 and x=0. Nodes 1,4 and 7 are all in the same context {A}. Similarly nodes 2,5 and 7 are all in the same context {B}.

The set of nodes can logically be separated into four non-overlapping sets:

- The *true* set containing premises;

- The *in* set containing nodes that hold in at least one currently consistent environment. A consistent environment may later be discovered to be inconsistent;

- The *out* set containing nodes whose label is empty; they do not currently hold in any known consistent environment but may do so at a later stage;

- The *false* set containing nodes whose datum is known to be false.

Basic operations and algorithms

The basic operations are the creation of an ATMS node (which is trivial), the creation of an assumption (which is trivial) and the addition of a justification to a node which can cause significant processing, because the ATMS has to construct the new label. The new label is constructed as follows:

1 - A sound and complete label is created by doing a union operation on all the combinations of picking one environment from each antecedent node label.

2 - This sound and complete label is then made consistent and minimal by eliminating inconsistent and subsumed environments.

The new label (if different from the old one) is then recursively propagated to the consequent nodes. If however the problem solver indicated that the new node is inconsistent, all the environments of the label are added to the 'nogood' database, and all the supersets of those environments are eliminated.

For example, let the ATMS currently contain the following data:

1:<x+y=1,{{A,B},{B,C,D}},{..}>

2:<x=1,{{A,C},{D,E}},{..}>

nogood{A,B,E}

From x+y=1 and x=1, the Problem Solver concludes that y=0 and tells the ATMS to create a new node, whose datum is y=0 and whose justification are nodes 1 and 2. To calculate the label of node 3, the ATMS:

1) Creates a sound and complete label using the algorithm above, giving: {{A,B,C},{A,B,C,D},{A,B,D,E},{B,C,D,E}}.
2) This sound and complete label is then made consistent by eliminating {A,B,D,E} which contains the inconsistent {A,B,E} and {A,B,C,D} which is subsumed by {A,B,C}.
3) Node 3 therefore becomes:

3:<y=0,{{A,B,C},{B,C,D,E}},{(1,2)}>

The notion of classes

The ATMS incorporates the notion of a node set or *class*. Each node can be made a member of one or more classes. The notion of class is completely independent of the label and the environments; it is simply an indexing scheme to facilitate the Problem Solver-ATMS interactions.

Interfacing

The ATMS is accessed via consumers. A *consumer* is a rule which does some problem solving on the data. A consumer is attached to one or more nodes and run when appropriate. It is never rerun since the data it is run on can never change. The two basic operations on a consumer are creating a consumer and running a consumer.

A consumer is created at the request of the Problem Solver, when the Problem Solver finds that it can make an inference. It informs the ATMS of the set S of nodes that form the basis of the inference and gives a description D of how this inference is to be performed. The ATMS then creates a new consumer node N whose antecedents are the nodes in set S and whose informant is D, calculates N's label in the usual way and attaches a consumer to node N. A consumer node is different from the ordinary nodes in two ways; first it has a consumer attached to it and second its datum field is empty.

A consumer is run by consulting the informant and performing the prescribed operation on the list of antecedent nodes.

The reason for having an informant field in the node now becomes clear. It facilitates the separation between the creation of a consumer and the execution of a consumer, i.e the separation between the realisation that an action can be performed and the decision to perform the action. This forms the core of the scheduling operation whose role is to determine which of the possible actions to undertake.

In the consumer creation algorithm, the label of the node to which the consumer is attached is calculated at consumer creation time. Theoretically, it could be equally well performed when the consumer is actually run. The reason for performing it at consumer creation time is to ensure that a consumer is run only if the subsequent datum will hold in at least one consistent environment (i.e when the label is not empty). This is beneficial because it prevents the execution of a possibly very complex inferencing algorithm in a (currently) non existent environment. It also allows the scheduler to use the environments to schedule the consumers. Scheduling a particular environment is equivalent to choosing a particular area of the problem search space.

Access paths to the data offered by the ATMS

The ATMS offers:

- A list of all the nodes together with the associated environments in which they hold;
- A list of all the classes with associated nodes;
- A list of all the consumers with the environments in which they can be run.

Conclusion

The basic ATMS therefore offers to a Problem Solver a method of recording the inferred data, a method of accessing the inferred data, a full record of the way the inferencing was performed, hooks for the triggering, scheduling and running of the different phases of the problem-solving process and a complete methodology for working with multiple contexts and dealing with inconsistent data. As such, it is a powerful tool that solves many design problems of the blackboard.

2.6. A blackboard architecture for fault diagnosis

2.6.1. Overall blackboard structure

The problem of fault diagnosis in complex systems naturally divides itself into several qualitatively different phases:

1 a communication stage where the fault reports are received;

2 a translation stage where the fault reports are translated into the implicated objects;

3 a filtering stage where all the expected findings are justified;

4 a combination stage where new findings are combined with existing findings;

5 a testing diagnostic stage where new tests to discriminate between the suspects are suggested or where the final diagnosis is made.

For example, if the system under observation is a computer network, the information received at the communication stage could be that a connection between two points A and B of the network failed. At the translation stage, this report would be translated into a list L of network components that can cause such a fault, in this case all the components on the route between A and B. At the filtering stage, the list L of suspect components would be checked to see if any of the members of L were expected to be down

(for example for scheduled maintenance), which would immediately explain the fault report. At the combination stage, the components of L would be combined with all the other suspect components obtained from the fault reports received during the current diagnostic session. This would lead to a new list of possible explanations for the system symptoms. At the testing/diagnostic stage a final diagnosis would be made if there was enough information, otherwise a new test would be suggested in order to discriminate further between the possible explanations for the system symptoms.

The problem space is thus divided into five well defined phases. Each phase performs a qualitatively different function within the diagnostic algorithm, which suggests the creation of a separate (level) control module for each phase. These different phases have to be coordinated in order to obtain the required diagnostic behaviour, which suggests an (overall) coordinating control module.

The data manipulated during the diagnostic process also have qualitatively different characteristics. The first type of data comprises untreated fault reports, the second translated fault reports and the third test requests or diagnostic reports. This suggests that the data should be divided into three separate modules.

The overall control and data characteristics suggest the architecture in Figure 2.7, partly based on [Nii86] , [Haye85a] , [Will87]. This architecture allows different control strategies to be applied to individual control levels. For example at the combination level, it may be desirable to trigger all the knowledge sources in order to find all the possible explanations (by performing all the combinations), but at the translation level it may be desirable to trigger only one knowledge source in order to process one fault report at a time. Furthermore the overall controller allows different overall strategies to be applied; for example one overall strategy can be to process each sensor datum all the way to the top level; an alternative strategy is to stop work on the current sensor data as soon as new sensor data appear and filter it up to the combination level [Bigh87].

In the proposed architecture a knowledge source frequently deposits the data on the same blackboard data level from which it took it (unlike [Will87]). This is because in the following processing cycle, the same knowledge source may need access to the data it produced in the previous processing cycle. For example assuming that in cycle 2 the sensor data combinator produced a list of suspects S, in cycle 3 it will want to combine new sensor data with S. A consequence of this is that in the proposed architecture, several control levels may correspond to one blackboard data level. It would be possible to join them under one large controller in order to establish a one-to-one correspondence between

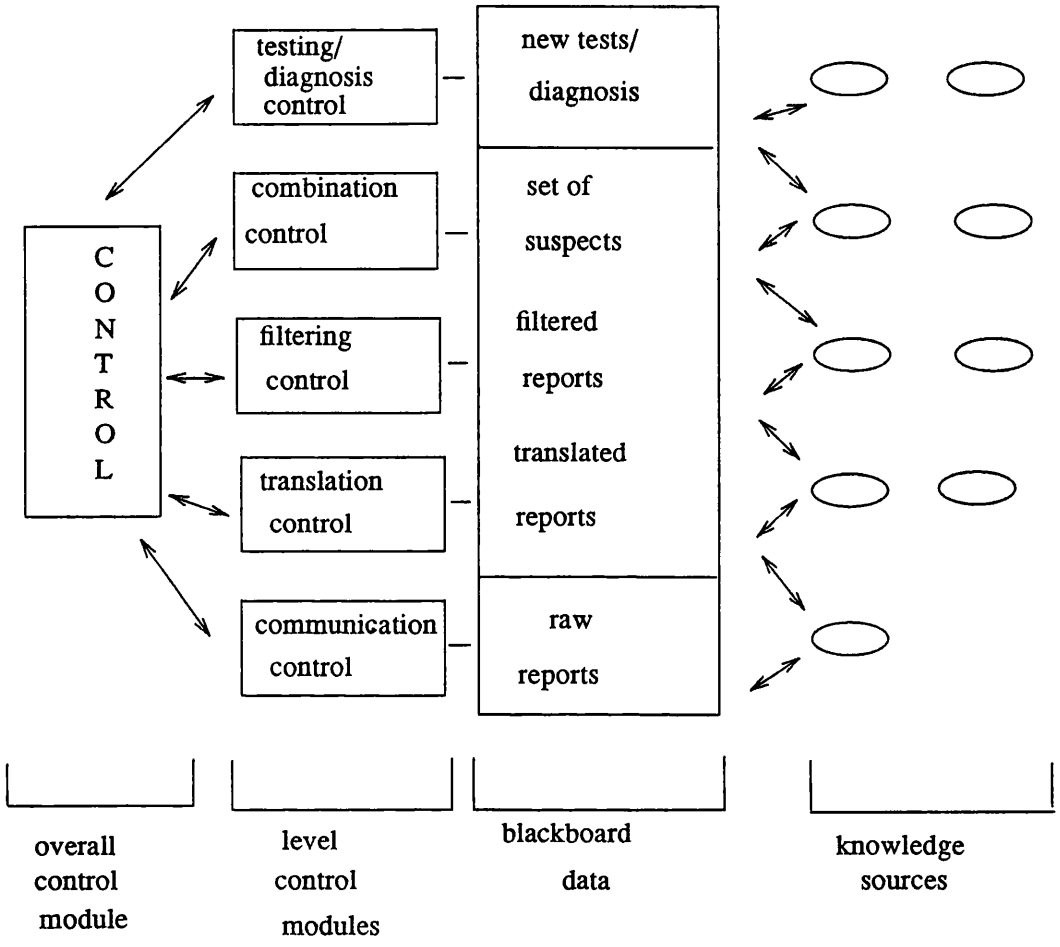


FIGURE 2.7 The proposed blackboard architecture

control levels and blackboard data levels, but since each control level corresponds to a logically independent activity, the proposed architecture offers greater flexibility. Therefore each knowledge source is allocated to a particular control level and among its characteristics, it has a blackboard input and a blackboard output data level.

2.6.2. Level control modules

The format of every level control module is the same. It is based on the model suggested by Hayes-Roth [Haye85a] and ensures total flexibility in the choice of knowledge sources at every control level. The basic loop of every level control module consists of 3 control actions; an action that checks which of the knowledge sources belonging to the control level are triggered by newly arrived data, an action which decides which of the

triggered knowledge sources to run and an action that controls the running of the knowledge sources. Thus, three basic control knowledge sources form the layer control (as depicted in Figure 2.8):

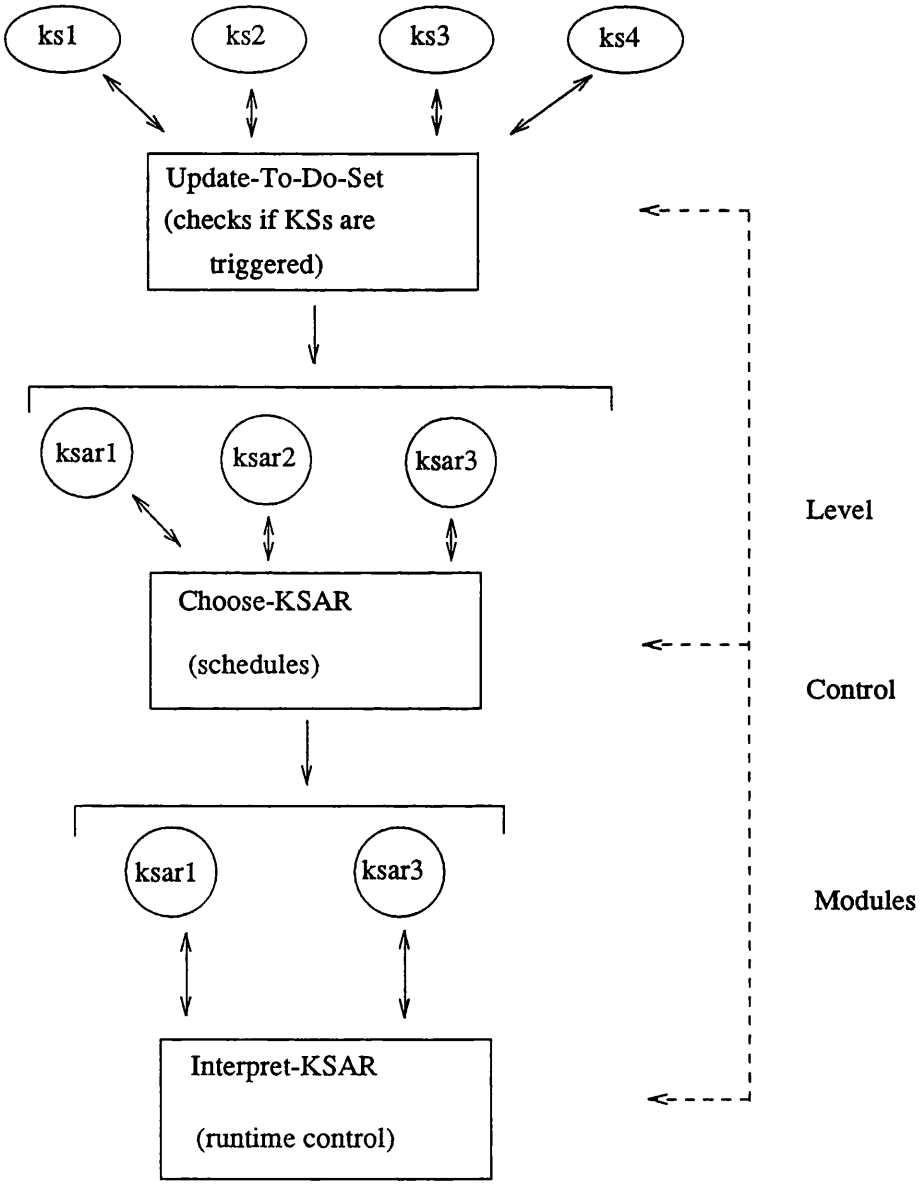


FIGURE 2.8 Level control procedure

An Update-To-Do-Set knowledge source, normally triggerable when new data arrive on the relevant data level of the blackboard. Its role is to loop through all the knowledge sources attached to that particular control level, check which ones have been triggered as

a result of the latest event, and create a Knowledge Source Activation Record (KSAR) for every knowledge source that is triggered. A KSAR records the fact that a knowledge source has been triggered and can contribute to the knowledge on the blackboard;

A Choose-KSAR knowledge source, triggered by the creation of new KSAR's on a level. It is basically a scheduling knowledge source, choosing one, some or all of the KSAR's according to the current scheduling rule;

An Interpret-KSAR knowledge source, triggered by the creation of a chosen set of KSAR's. Its role is to interpret and execute the action of the chosen KSAR.

2.6.3. Overall control module

The main role of the overall control procedure is to schedule the different level controllers. This enables the system to use alternative problem solving behaviours as described earlier in section 2.6.1.

2.6.4. Knowledge sources format

The attributes of a knowledge source are given in Table 2.3.

Attribute	Description
Name	Identifier
Description	Behaviour description
Control level	Triggering level
Trigger	Triggering conditions
Assumptions	Additional assumptions made by the knowledge source
Action	Program of blackboard changes
Input level	Corresponding blackboard data level for inputs
Output level	Corresponding blackboard data level for outputs

TABLE 2.3 Knowledge source format

The *name* is an identifying label.
The *description* is a description of its characteristic behaviour.
The *control level* is the control level at which it is triggered.
The *trigger* is a description of the events that trigger it.
The *assumptions* are the additional assumptions that the knowledge source makes, i.e preconditions to be fulfilled before it can be scheduled. This field gives more flexibility to the triggering and scheduling of knowledge sources. For example, in a diagnostic system, a knowledge source may be triggered by the arrival of new sensor data, and make the additional assumption that there is only a single fault in the system. It would then only be schedulable if the scheduler decided to use the scheduling rule that makes the single fault assumption.
Action is the program of blackboard changes instigated by the knowledge source.
Input level is the blackboard data level on which the knowledge source looks for its events.
Output level is the blackboard data level on which the knowledge source deposits its results. By specifying the input and output levels explicitly, we enable the knowledge sources on the same control level to send their output to different data levels (if necessary) and to accept events from different data levels.

2.7. Interfacing the blackboard and the ATMS

The functional division between the blackboard and the ATMS is well defined; the blackboard plays the role of Problem Solver as defined in the generic TMS in section 2.5.1. The blackboard possesses the knowledge (in the knowledge sources) and the control (in the level control and overall control modules) which enables it to perform all the inferencing. The ATMS offers its services (described earlier) to the blackboard, in connection with the recording of the current knowledge about the actual case in progress.

2.7.1. Interfacing the consumer construct to level control

The interface between the blackboard control modules and the knowledge sources on one side and the current case blackboard data occurs at each level control; new and existing data are examined to see which knowledge sources are triggered, creating KSAR's, and KSAR's are run, accessing the existing data and depositing the new data.

The ATMS/Problem Solver inteface is done through the consumer construct as seen in section 2.5.4. Consumer nodes are created, together with the environments in which their data will be valid, and consumers are run by calculating the data.

There is thus a close correspondence on one side between the operations of triggering KSs and creating a consumer, and on the other side between running a KSAR and running a consumer, as shown in Figure 2.9 and thus the two are interfaced at this point.

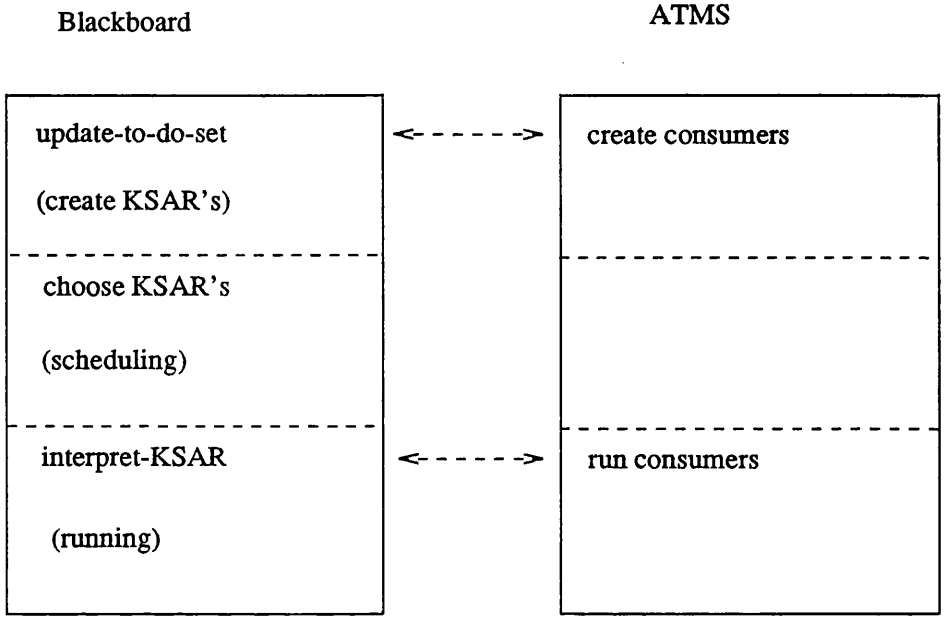


FIGURE 2.9 ATMS/Blackboard correspondence

The Update-To-Do-set control knowledge source which controls the triggering of knowledge sources at the layer control level is thus defined by the following algorithm:

Given the latest addition to the pool of data on the corresponding data level;

1. find all the knowledge sources belonging to the control level;
2. for each one of them;
   - 2.1 check if it is triggered by the new data and the existing pool of information;
   - 2.2 for each triggering;

     create a new ATMS consumer node whose antecedents are the knowledge source, the new data and the existing data that was necessary for the triggering, whose label is calculated in the normal ATMS way and whose informant is the action part of the knowledge source (this is applying the consumer creation algorithm described in section 2.5.4). Therefore the triggering algorithm will produce one or more consumers attached to consumer nodes (which are equivalent to KSAR's).

The scheduling algorithm will schedule one or more consumers.

The Interpret-KSAR control knowledge source which controls the running of scheduled KSAR's (i.e consumers) on a layer is defined by:

For all scheduled consumers:

1. Consult the informant part of the node to which the consumer is attached (equivalent to the action part of the knowledge source).
2. Execute the operation prescribed in the informant on the antecedent nodes, giving a result (the new datum).
3. Delete the consumer and insert the new datum into the node's datum area (corresponding to the blackboard data).

2.7.2. Adapting the ATMS to the proposed control structure

The basic ATMS does not have a concept of levels; all the nodes are treated as one block. Therefore all the information about the environments is kept in one block as depicted in Figure 2.10.

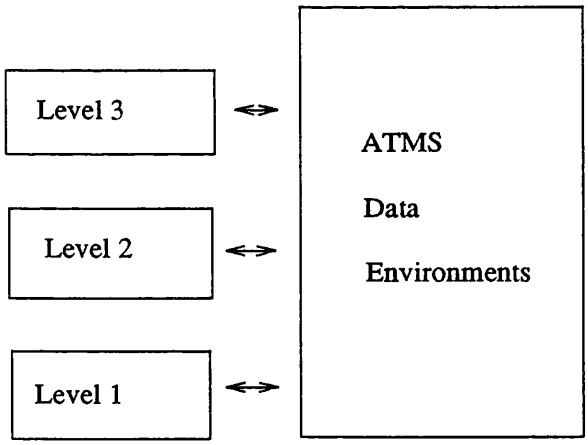


FIGURE 2.10 Basic ATMS

However, since the search space has been split into several levels (the blackboard control levels), since the triggering and scheduling decisions are mainly done inside these control

levels, and since the list of environments is used to make and help scheduling and triggering decisions, it is advantageous to split this monolithic environment list. Instead, there is one ATMS environment list corresponding to every blackboard data level, as depicted in Figure 2.11.

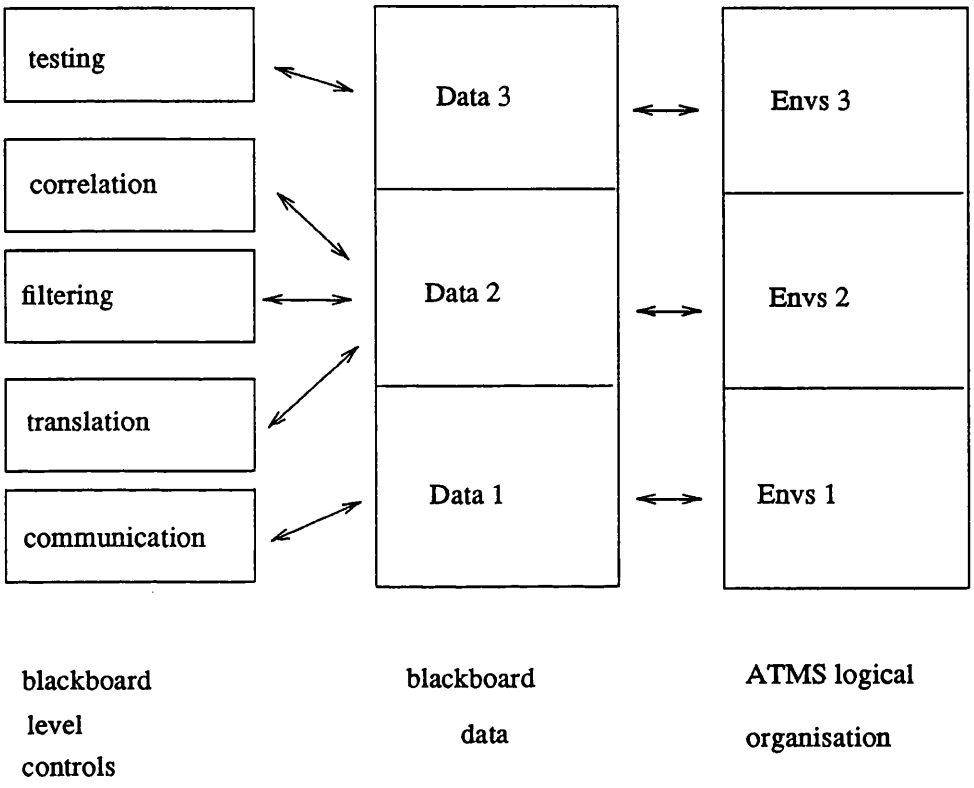


FIGURE 2.11 Split environment ATMS

This means that the scheduling and triggering decisions made on one blackboard control level need examine only a portion of all the environments, with a corresponding increase in speed. This is done by adding a slot to the ATMS node corresponding to the blackboard data level and allowing the ATMS to use that slot to determine which environment list is to be accessed or updated.

The same idea cannot be applied to 'nogood' environments because they span all the levels; this is because the inference chains cut across the level boundaries. A 'nogood' environment on one ATMS level is 'nogood' on any ATMS level. For the same reason, we cannot introduce the idea of an assumption belonging to an ATMS level. The inference chains and the label update algorithms automatically carry assumptions across

ATMS level boundaries. An assumption is only "made" on a particular ATMS level.

An interesting idea would be to separate the environments found in consumer nodes, which are mainly used for scheduling and the environments in other nodes which are mainly used for triggering decisions. This would mean that to make scheduling decisions, we need only examine the "schedulable" environments. At the same time, when looking for nodes that trigger a knowledge source, we would scan only the nodes which can lead to triggering (we do not allow a knowledge source to be triggered by a consumer). This would lead to the architecture depicted in Figure 2.12.

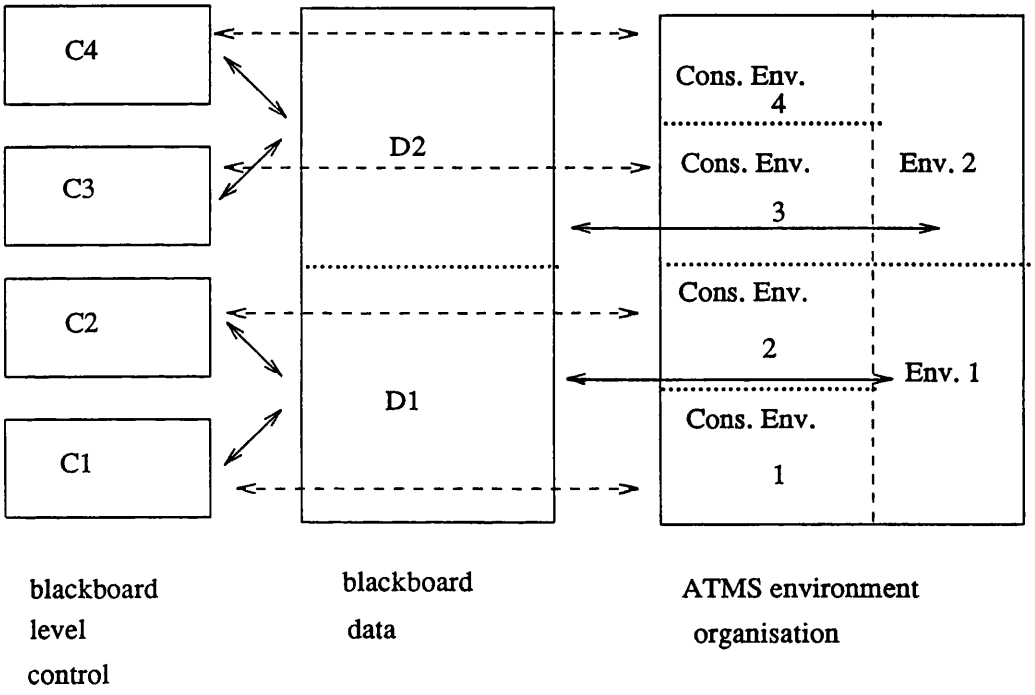


FIGURE 2.12 ATMS with separate consumer environments

The gain in processing time from this organisation has to be weighed against the fact that every time we run a consumer, we have to transfer the environment from the consumer list to the other list.

2.7.3. Using the ATMS to help triggering

The latest data to be put on the blackboard is easily available; it corresponds to the node number(s) of the last consumer(s) to be run. However, the triggering of a knowledge source does not depend only on the latest additions to the pool of data; it also depends on

the data that had been deduced previously. The ATMS can help the knowledge sources to find the data which triggers them in at least two ways; through the node class construct and through the indexing of the environments.

The ATMS can be instructed to put all the nodes that are potentially interesting to a knowledge source into a particular class. For example if a knowledge source knows that z = x + y + 5, a class X can be formed to include all the nodes whose datum gives a value to x. The knowledge source would then be triggered every time a value for y is put on the blackboard if there is at least one node of class X.

The knowledge source may be triggerable by all the nodes that are valid in a certain environment. Again the ATMS provides the node-environment indexing that makes it very easy to provide the knowledge sources with this type of information. This facility generalises to several or all the environments.

The logical splitting of the ATMS into levels also helps the triggering by immediately structuring and reducing the search space for already deduced data. While a sequential search of all the nodes in the ATMS is normally too expensive to consider, a sequential search on a level with a relatively small number of nodes is often a good solution.

Of course the problem solver can itself provide an indexing or retrieval scheme for the already deduced data instead of or in conjunction with the ATMS facilities.

2.7.4. Using the ATMS to help scheduling

The basic ATMS itself is neutral with respect to scheduling; the scheduling is the responsibility of the problem solver. The ATMS is inherently at its best when all the solutions are wanted because it automatically looks at all the contexts, which means that there is no backtracking and expensive context switching. However the ATMS can be controlled to obtain different scheduling strategies.

The ATMS offers to the problem solver the list of environments that the consumers are valid in. By using the list of environments to help the scheduling and in effect scheduling environments rather than consumers, different problem solving behaviours can be achieved. This highlights the advantage of calculating the label at the time the consumer is created rather than when the consumer is run; the calculated label can be used to make sure that only consumers valid in at least one consistent context are run and also as an aid

for scheduling consumers.

Depth first search

Normally as a search progresses along a particular line of thought, new assumptions are added, (or at least the old ones are not normally lost). Therefore by always scheduling the environment with the largest number of assumptions, we obtain a very good approximation of the depth first search algorithm. This can be done by the ATMS without any understanding of the semantics of the data.

Following one line of thought

The ATMS automatically looks at the whole solution space. However, the problem solver may want to explore one particular line of thought before trying any others (because the solution(s) are more likely to be in that subspace for instance). This can be done by asking the ATMS to find and schedule only consumers whose label contains at least one superset of the wanted environment. If that line of thought leads nowhere (normally when the environment proves to be inconsistent), attention can be switched to some other part of the search space by scheduling supersets of a different environment. For example, initially, a diagnostic procedure may want to make the assumption that there is a single fault, and follow that line of thought. If that assumption proves inconsistent, it could then switch attention to the part of the search space that contains multiple faults. This approach can furthermore be generalized to follow several lines of thought at the same time. In contrast with the previous case, the ATMS has to be given the "best" environment in this approach.

Best first search

Because the ATMS does not have any idea of the semantics of the data, it cannot assist in finding what the best partial path to the solution is; that is entirely in the domain of the problem solver. The problem solver must keep an independent record of the rankings of the partial solutions reached. This ranking must be applicable to the environments that are schedulable. The ATMS can then be instructed to schedule whichever is the 'best' environment and this procedure can be repeated at every step.

Scheduling by class

Since consumers are linked to nodes, and nodes can belong to classes, consumers can be considered to be directly linked to a class. Therefore control over scheduling can be exercised by instructing the ATMS to schedule only consumers which are linked to one or

some node classes.

2.8. Comparison to other blackboard architectures

The coupling of a blackboard with an ATMS has in one move yielded an architecture which:

- is exceptionally well suited to looking for all the solutions;
- solves the problem of triggering and running knowledge sources, helps scheduling and keeps track of the way the solution was reached through the same mechanism;
- automatically offers a method of giving consistency information, checking the consistency of the information and eliminating the inconsistencies;
- allows instant change of focus.

All these functions are needed for diagnostic problems. The alternative of providing each one of them separately can easily lead to a duplication of some of the functions. Furthermore, by providing all of these functions together, and clearly separating them from the process of reasoning, more attention can be devoted to the problem solving aspect. The mechanism chosen provides a very clean interface both for data and for control. This section looks at how some other blackboard organisations used in comparable tasks have solved these problems.

HEARSAY

Description: The HEARSAY [Erma80] system's task was to try and understand speech utterances. This involved making several alternative hypotheses about the words, and exploring them. The main characteristics of the HEARSAY architecture included:

- The division of the blackboard into several levels;
- The usage of an AND/OR tree to link the hypotheses on the different levels;
- A method to maintain alternative hypotheses based on AND/OR trees;
- Triggering performed by the knowledge sources searching the blackboard for interesting data;
- The splitting of the control into monitor and scheduler. The role of the monitor is to decide which KS's may be triggerable, and the role of the scheduler to decide which triggered KS to execute.

Discussion: The weak points of the system are:

- The blackboard structure did not offer any help in the triggering activities; therefore the KSs had to search vast areas of the blackboard to find relevant data. To alleviate this problem, the condition part of the KSs declared a priori the kind of changes it is interested in.
- The complexity of the maintenance of alternative hypotheses. In the blackboard/ATMS architecture, the maintenance and the switching between alternative hypotheses is trivial and comes as part of the architecture. In HEARSAY, these procedures were not provided and supported by the system. They were part of the problem solver and based on AND/OR trees where the OR branches represented alternative solutions. This led to an expensive and complex way of maintaining alternatives [Nii86]. This was made worse by the task of the system (understanding speech utterances), which is typically characterised by a large number of alternative hypotheses (due to the difficulty of understanding speech).
- The architecture did not offer any help in the evaluation of the consistency of the data. This was again left to the knowledge sources.
- The use of one scheduler to select the focus of attention across all the layers is quite a rigid control mechanism for a layered architecture. Splitting control between layers and creating a control hierarchy makes it easier to pursue local control decisions and make local changes to the control application.

HEARSAY is notable because it was the first system to use the blackboard architecture. It therefore defined most of the concepts and rules of blackboard architectures.

HASP/SIAP

Description: The Hasp/Siap system [Nii82] was built to record the position of ships and submarines in a region under surveillance, with the data coming from several sources (radar, intelligence reports etc). The main characteristics of the system were;

- the blackboard was divided into levels;

  a solution consisted of partial results across the different levels;
- each hypothesis element on the blackboard could have alternative values for its attributes but no alternative links to other hypothesis elements; the hierarchy of

hypothesis elements was organised as an AND tree;

- the entire reasoning process was kept in a history file;
- the control was divided into strategic control to choose a global strategy and tactical control to control specific strategies.

Discussion: The major weaknesses of the system are:

- Its treatment of the search of alternative solutions. The choice of an AND tree (with no alternative links) to support the hypothesis hierarchy meant that it was difficult for the system to change its mind and change the focus of attention. To change the focus of attention, HASP in fact forgot previously deduced data and restarted from the point of departure. This can be a sensible method if depth-first search is performed i.e if the focus of attention is changed only when the current one becomes inconsistent. However, if the focus of attention is changed between partial currently consistent solutions, the method is extremely inefficient. It is in fact even more inefficient than using an ordinary TMS, (as opposed to an ATMS). An ordinary TMS remembers previously deduced data, but has to backtrack in order to find it. HASP has to rededuce all the data.
- The architecture does not provide any help in dealing with inconsistent data, and that for a task where the amount of noise is very high and the reliability of the data is very low.

HASP is notable because it was the first system to use a hierarchical control organisation, which enabled it to reason about control strategies at the higher levels of hierarchy, thus making the control mechanisms more explicit.

KRITIC

Description: The Kritic project [Will87] , [Bigh87] was formed to study fault problems in large industrial systems such as advanced telecommunications switching systems. Therefore the characteristics of the problem are very similar to the ones discussed here. The architecture proposed by the project is in part very similar to the architecture proposed in this Chapter; a layered blackboard with one control module for each layer and an overall control module. Furthermore one of the layers is linked to the RUP truth maintenance system developed by McAllester [McAl80].

Discussion: The weak points of the system are:

- The usage of the TMS for only one of its functions; all TMSs provide at least the consistency checking function and the tracing of the reasoning path. Most of them can also be used to help scheduling and triggering. Here the TMS is used only for its consistency checking function. Given the inevitable overheads of a TMS, it makes sense to use all of its facilities.

- The usage of a TMS on one layer only; this is linked to the previous point of using the TMS for one function only. Since the consistency checking is mainly needed at one level, the TMS was linked to the hypothesis level only. It gives a good example of how easy it is to unwillingly duplicate the functions needed for a blackboard. Since the TMS's provide a way of tracing the reasoning path to the solution, by putting the TMS on one level only, either this facility of the TMS is ignored, or an alternative (i.e duplicated) mechanism has to be provided at the other layers.

- The choice of McAllister's TMS which insists on one consistent state (i.e gets one solution at a time) for a problem where all the solutions are wanted. This leads to unnecessary backtracking.

This system is notable because it seems to be the first system to couple a blackboard with a TMS, thus providing a uniform methodology for switching contexts and dealing with inconsistent data.

2.9. Summary

This chapter has analysed the characteristics of fault diagnosis in dynamic systems and proposed a new architecture particularly suitable for the problem, based on the coupling of a blackboard with an Assumption-based Truth Maintenance System. The major motivation for this coupling is the inherent complementarity of the two systems. The blackboard provides a flexible means of control and scheduling of the tasks, while the ATMS provides the means to store and retrieve the data, a record of the inferences, the means to pursue alternative hypotheses, consistency checking and support for the blackboard control functions. The Chapter developed the interface between the blackboard and the ATMS and showed how the blackboard uses the ATMS to help it perform its control functions. Finally some existing architectures which deal with similar problems were analysed and the advantages of the proposed architecture were shown. The proposed architecture is particularly suitable for complex tasks with many solutions where all the solutions are required, but can be adapted to other tasks as well.

CHAPTER III

DIAGNOSING SINGLE,MULTIPLE AND SENSOR FAULTS AND REACTING TO STATE CHANGES

3.1. Introduction

Diagnosis is the process of explaining a sequence of behavioural observations which includes discrepancies between expected and actual behaviour. It normally requires repetitive application of three phases: one in which symptoms are translated into sets of suspect elements, one in which the new suspect elements are combined with existing suspect elements deduced from previous tests, and one in which new evidence-gathering tests are proposed in order to refine the set of suspects. Most research work on the topic has made the following simplifying assumptions: single fault assumption, single state assumption and no sensor fault assumption [Gene84] , [Hams83] , [Shub82] , [Laff84]. By making those assumptions, the search space for faults is immediately constrained, and no backtracking or data consistency checking is ever necessary. This chapter will develop a general deep knowledge fault isolation approach more suitable for complex dynamic systems, that can be used in conjunction with the architecture developed in the previous chapter. The approach consists of two parts. The first part is a novel methodology for unifying single, multiple and sensor faults. The second part is a methodology for dealing with state changes, based on generating suspicions about state changes, confirmation of state changes and belief revision following a state change. This chapter assumes that there is already a function that translates the symptoms into a list of suspects.

3.2. Notation and definitions

$t_i(a,b,c)$ describes a test t_i which can be performed on the set of elements a,b,c. Such a test cannot distinguish between the individual elements of the set; it observes the behaviour of the whole set.

t_i<a,b,c> describes a *failed test* also called an *unsuccessful test.* A test has failed if the observed behaviour of the element set it is testing is not equal to the working behaviour of that set. <a,b,c> then becomes a *symptom:* it is symptomatic of a fault in the set (a,b,c).

t_i/a,b,c/ describes a *successful test.* A test is successful if the observed behaviour of the element set it is testing is equal to the working behaviour of that set.

$[\overline{f_1..f_i}:w_1..w_j]$ is a *candidate tuple* or simply a *candidate.* A candidate represents a particular hypothesis which explains all the test results obtained so far. A candidate consists of two parts separated by the symbol ":". The first part consists of a set of elements $\overline{f_1..f_i}$ which are not working under the current hypothesis (also called *suspects)* and the second a set of elements $w_1..w_j$ which must be working under the current hypothesis.

A *test fault* is a test result which does not represent correctly the state of the set of elements it is testing, i.e it corresponds to a false diagnosis by a test. It is not to be confused with a failed test where the test records the correct state, which is that one of the tested elements is not working.

The notation introduced here is different from existing notations in general, and from that used by DeKleer [Klee87] in particular. The main reason for this difference is that DeKleer bases his reasoning on failed tests only, thus eliminating the need to find a notation for working elements. Consequently DeKleer's candidates have only one part (the elements whose failure would explain the symptoms) whereas this notation introduces candidates consisting of two parts: failed and working elements (since our reasoning will be based both on successful and failed tests). The other major difference is that this notation puts a bar over the non- working elements in the current hypothesis which DeKleer does not do. This makes it possible to have a simple differentiation between the non working and working elements of the hypothesis. It is also more logical to use the bar to denote non-working elements than working ones.

The words sensor and test do not have the same meaning. A sensor is an element whose role is to observe the system, and it can have the capacity to perform many tests. However, in the text, the terms sensor fault and test fault and sensor data and test data have been used interchangeably to signify in a loose sense the objects that perform the observation of the system and the data from those objects. In the few places where the

distinction was important, the reason for the distinction is made abundantly clear.

3.3. Characteristic points of complex dynamic systems

The single fault, single state and no sensor fault assumptions are normally valid in the circuit diagnosis environments that they have been used in. In complex dynamic systems they are too restrictive for the following reasons:

Possibility of multiple faults: There is no basis for making the simplifying single fault assumption; since the system is dynamic, any number of faults can be present in the system at any time and in particular the system may never be fault-free.

Possibility of sensor faults: Since the sensors are normally part of the system under observation and may use it to transmit their data, there is no basis for assuming that the sensors are always working or producing correct data. The data can therefore be wrong or contradictory.

Possibility of system changes during diagnosis: Since the system is dynamic, there is no guarantee that it will stay in the same fault state throughout the diagnosis period. New faults may appear at any time, or old ones may disappear (intermittent faults).

Confusion between sensor and temporal faults: State changes in the system during diagnosis and sensor faults both manifest themselves in the same way; through an inconsistency in the data set. Additional mechanisms are therefore needed to separate them.

No proof of correct diagnosis: One of the consequences of allowing state changes in the system during diagnosis and sensor faults is that it is impossible to guarantee the correctness of the diagnosis. Any current diagnosis can already be false due to a state change, or because of sensor faults. The methodology should thus be able to make probabilistic conclusions.

Data synchronisation problem: Because diagnosing complex dynamic systems is normally performed on-line, and the sensors are part of the system being diagnosed, it is almost impossible to synchronise the sensor data.

3.4. Existing multiple fault strategies

3.4.1. Explicit enumeration of possible faults

This strategy entails keeping a list of all the possible states in the system, where a system state is a combination of the states of each individual element of the system. The diagnostic process then applies the symptoms to this explicit list, eliminating the states that are incompatible with the symptoms, hence keeping the ones that could explain the symptoms. Diagnosis is finished when only one suspect state remains.

Assuming that the state of a component is either "working" or "not working", the number of possible states the system can be in is 2^n , where n is the total number of components. Hence explicit enumeration of all the possible states will work if the system is small. For any other system, the explicit enumeration of faults becomes prohibitively expensive.

3.4.2. Network Diagnostic System (NDS)

NDS [Will83] is an expert system to perform fault isolation in a communication network (COMNET). The system claims to deal both with multiple and intermittent faults. The diagnosis is a type of top-to-bottom strategy. One of the hypotheses that can explain the current symptoms is chosen and successively refined until a replaceable unit is diagnosed as broken. This replaceable unit is then repaired, all symptoms that referred to it are eliminated and the system checked to see whether there are any other symptoms. If some symptoms are left the strategy is reapplied to the most general hypothesis not affected by any symptom data revoked after the repairs.

The literature does not explain how intermittent faults are dealt with. Furthermore, the whole strategy is based on the strong assumption that it is always possible to repair a component before continuing the diagnosis. If that is not possible, the whole strategy fails, because there is no way of revoking any symptom data.

3.4.3. The suspect islands/ clustering approach

This approach as reported by Williamson et al [Will87] specifies a three level strategy. On the first level, raw fault reports are received from the system. On the second level, these fault reports are translated into implicated objects. (These two levels correspond to the communication and translation levels proposed in the previous Chapter). On the third levels, these implicated objects (i.e the suspect elements) are sorted into separate clusters, each cluster containing the symptoms related to the same cause. Subsequently each

cluster is in effect treated separately. This approach is depicted in Figure 3.1.

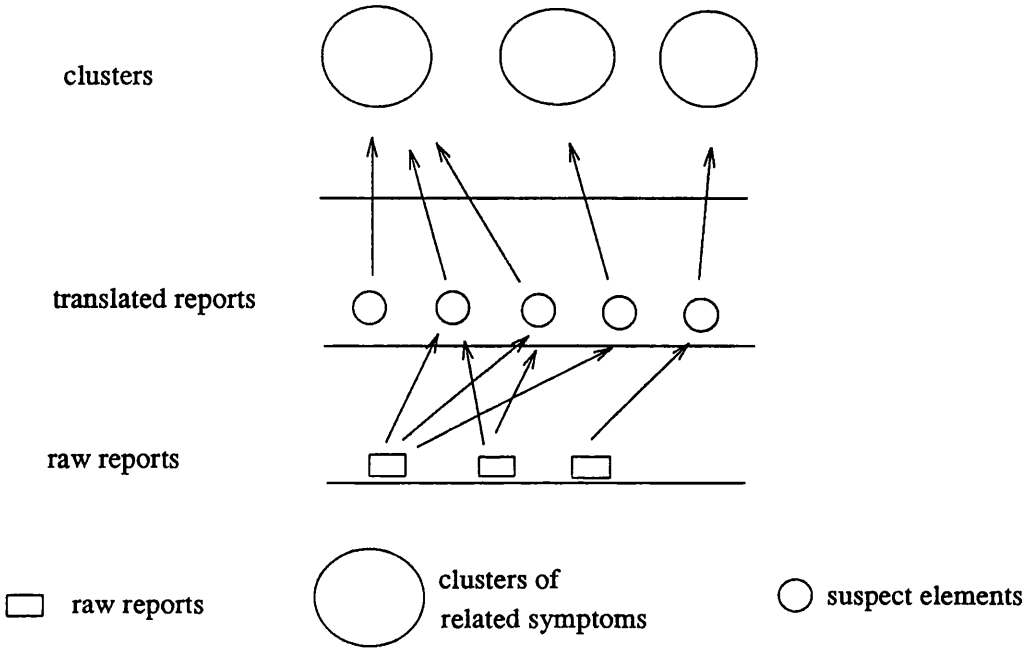


FIGURE 3.1 Cluster approach to multiple faults

The pre-requisites for the clustering approach are:

- knowledge about choosing the clusters;
- knowledge about allocating the symptoms to the clusters.

For example if the result of test t_1 is <a,b,c> and of test t_2 <a,d,e>, it becomes necessary at this stage to decide whether these two reports belong to the same cluster (which would mean that a is faulty) or to two separate clusters (which would mean that either b or c are faulty in cluster 1 and that either d or e are faulty in cluster 2). The search for the faults would then continue separately on each cluster. If subsequently the result of test t_3 is <b,d>, it is necessary to decide whether this result should be combined with cluster 1 or cluster 2.

Sometimes this clustering knowledge is obvious. For example, if there is no intersection between the objects implicated by two tests, they will form separate clusters. However, if there is an intersection between the objects, there is no clear system independent

methodology for choosing the clusters and allocating the symptoms to the clusters. There may be a system dependent cluster allocating methodology but:

- the clustering approach fails if such a methodology does not exist;
- the approach becomes complicated if mistakes are made when allocating the symptoms to the clusters. A mechanism is then needed for undoing all the intermediate conclusions made under the wrong allocation, for transferring symptoms from one cluster to another and creating different clusters for the same data.

This methodology does not solve the problem of multiple faults; it bypasses it by assuming that they can be treated as separate, individual single faults.

3.4.4. Geffner & Pearl

Geffner and Pearl [Geff87] propose a distributed scheme for diagnosing multiple faults. In their model, the system under observation is represented using a Bayesian belief network. Bayesian belief networks are directed acyclical graphs where the nodes represent propositional variables and the links represent causal influences between the nodes. This methodology is depicted in Figure 3.2 which shows how a simple multiplier with two inputs I1 and I2 and one output X is translated into a Bayesian network.

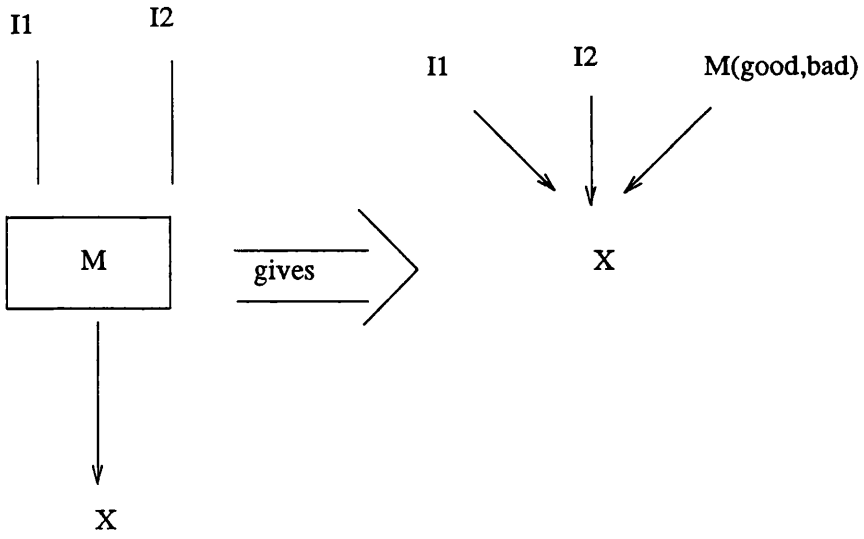


FIGURE 3.2 Mapping to a Bayesian network

Each new piece of evidence (i.e fault symptom) is viewed as a perturbation that

propagates through the network via local communication between neighbouring nodes. At equilibrium, the total of all the values contained in the nodes forms the best possible global interpretation of the evidence.

The problem with this method is that incoming information is assumed to determine unequivocally the values of some variables. There is no allowance for sensor faults or for changes in the systems. Once a value has been propagated, there is no way of retracting it or returning the diagnosis to a point prior to the occurrence of false data. Fundamentally, the problem of dealing with inconsistencies is a global problem and should be dealt with globally. This methodology is more geared towards problems that have definite, consistent data.

3.4.5. Davis

Davis' approach [Davi84] is to enumerate and layer the categories of failures according to their likelihood. The diagnosis then proceeds within the first category until the fault is pinpointed or until some of the assumptions of the category are violated. The sample categories chosen by Davis as an example reflect his application field (troubleshooting hardware), but a typical corresponding ordering would be:

- single faults
- multiple faults
- state change
- sensor faults

This approach is interesting as a general approach to troubleshooting because it offers an orderly and systematic way of approaching the diagnostic problem and limits the number of faults considered (the search space) at any time. The choice of categories of failure is obviously of extreme importance. One important implication of the model however is that when changing the fault category, the previously deduced data normally become invalid, since they were deduced under assumptions that were too restrictive. Recovery to a consistent set of data requires a substantial amount of backtracking which is computationally expensive. The proposed model avoids this backtracking by dealing simultaneously with single, multiple and sensor faults.

3.4.6. Parsimonious covering theories

Parsimonious covering theories [Peng86] are methods for describing the whole space of suspects using the simplest possible explanations. They are based on Occam's razor: simple explanations are preferred to more complicated ones, and more plausible explanations are preferred to less plausible ones. They are the main methodology for treating multiple faults. Any parsimonious theory has implicitly to define four elements :

- a set of legal faults;
- a representation of the current set of suspects;
- a criterion for ranking the different suspects;
- a mechanism for integrating new diagnostic data into the current set of suspects.

The parsimonious theories differ by the restrictions they impose on the set of acceptable explanations for the current symptoms. Four major types of restriction exist.

3.4.6.1. Single fault restriction

This approach restricts the set of possible faults to single faults, thereby using intersection to integrate new diagnostic data. This method is insufficient for fault diagnosis in complex dynamic systems because it does not represent well the space of possible system states.

3.4.6.2. Minimal cardinality approach

In the minimal cardinality approach, the whole space of suspects is theoretically examined. However, given a set of suspects, only the ones containing the smallest number of faulty components are kept. For example if a fault can be explained by either element a being faulty ([$\overline{a}$:]), or b being faulty ([$\overline{b}$:]) or c and d being faulty ([$\overline{cd}$:]), only ([$\overline{a}$:]) and ([$\overline{b}$:]) are taken into account. This methodology is again inappropriate because it eliminates perfectly plausible candidates.

3.4.6.3. Relevancy

In the relevancy approach, all the candidates that can explain the current test findings are included in the list of suspects. For example, in Figure 3.3, given the symptoms t_1<a,b> and t_2<b,c>, the relevant candidates are [$\overline{b}$:], [$\overline{ab}$:], [$\overline{ac}$:], [$\overline{ad}$:], [$\overline{bc}$:], [$\overline{bd}$:], [$\overline{cd}$:], [$\overline{abc}$:], [$\overline{abd}$:], [$\overline{bcd}$:], [$\overline{abcd}$:] (and the minimal cardinality and single fault candidate is [$\overline{b}$:]). The relevant candidate line on the diagram separates the relevant and irrelevant candidates.

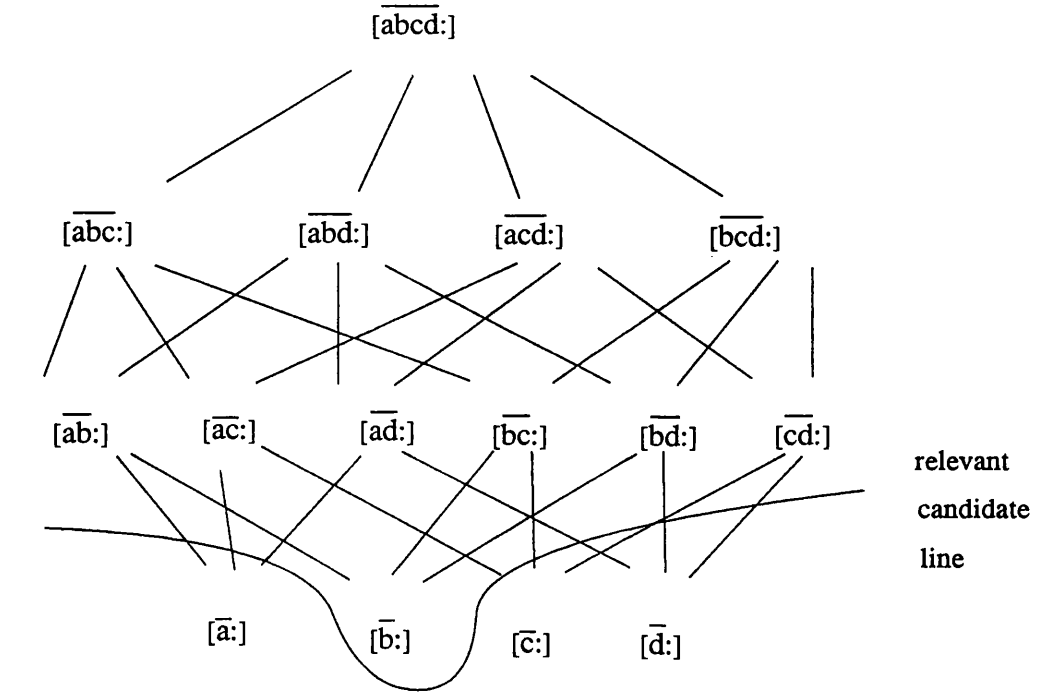


FIGURE 3.3 Relevant candidates after measurements

The problem with this definition is that it is too loose and still too expensive in computational terms. Most of the candidates from the initial set of candidates remain. This approach also does not sufficiently link the list of suspects and the diagnostic findings. For example several of the candidates in Figure 3.3 contain element d. None of the symptoms at that point suggest that d is faulty. There is therefore no reason to consider d as suspect, until some test suggests that d may in fact be faulty.

3.4.6.4. Minimal candidates

General strategy

The concept of minimal candidates [Klee87] also called irredundancy [Peng86] and minimal hitting set [Reit87] is used to represent and manipulate the set of possible candidates in a concise way. It uses the fact that any superset of a candidate is also a candidate (e.g if [$\overline{a}$:] is a candidate, then [$\overline{ab}$:] is a candidate). Therefore, the set of candidates is

represented by the set of minimal candidates, where a candidate is a minimal candidate if it has no subset which itself is a candidate. For example, in Figure 3.4, given symptoms <a,b> and <b,c>, the minimal candidates are [$\bar{b}$:] and [$\overline{ac}$:].

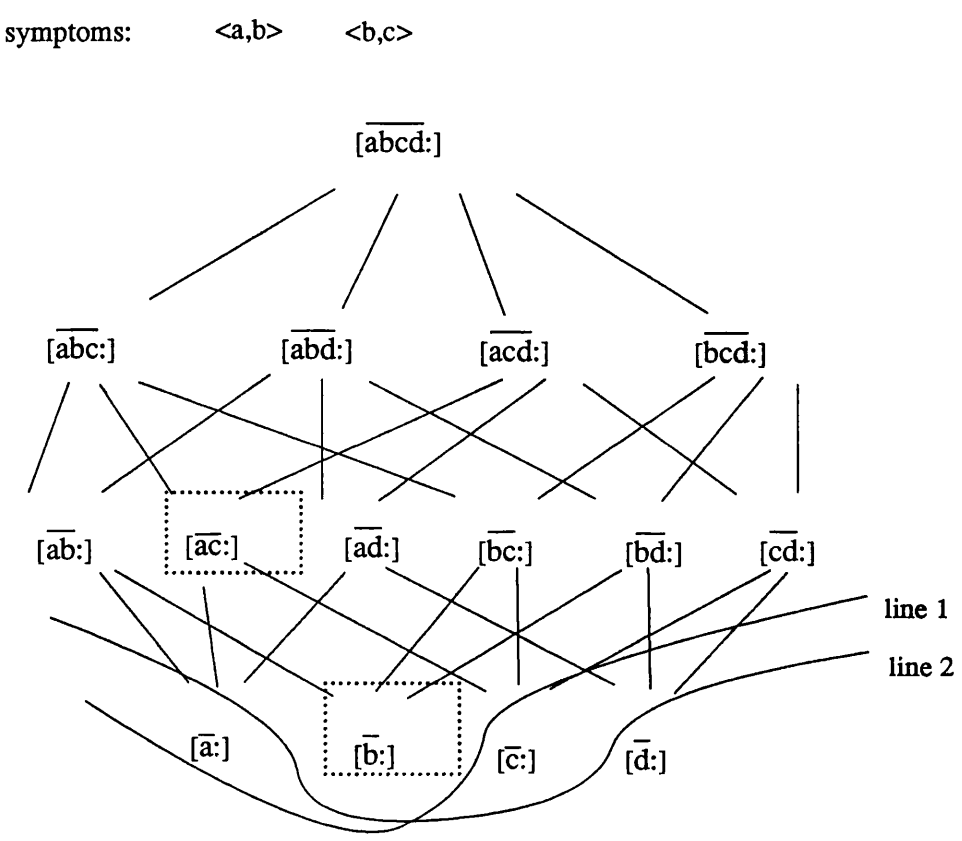


FIGURE 3.4 Minimal candidates after measurements

The advantage of this method is that:

- it takes explicitly into account only the elements implicated by the diagnostic findings (unlike the relevancy concept);
- it still implicitly covers all the legal candidates (unlike the single fault and minimal cardinality concepts);
- it is concise even for a large number of elements because the number of suspects can be made to depend completely on the diagnostic findings and not on the size of the system;
- it does not make any strong assumptions about the system (such as the ability to repair

faults immediately);
- it keeps the suspects together (unlike the Geffner/Pearl algorithm).

Minimum candidate generation

Diagnosis is done incrementally; new measurements are used to refine the current beliefs. New minimal candidates are generated incrementally, using the current minimal candidates and the new test findings.
The minimal candidate generation algorithm, proposed by DeKleer et al [Klee87] is as follows. When a new symptom, giving a new list of suspects is discovered:

1) all previous minimal candidates that can explain the new symptom are added to the set of new minimal candidates.

2) all previous minimal candidates that do not explain the new symptom are replaced by new superset candidates. These candidates are obtained by replacing the old minimal candidate with a set of new candidates, each of which contains the old candidate plus one element from the new list of suspects. The elements of this set that are subsumed or duplicated by some other element of the set or by any element kept in 1) are eliminated, the others are added to the new list of minimal candidates.

For example, given a system of 4 elements a,b,c,d, all the possible combinations of suspects were depicted in Figure 3.4. Before any symptoms appear the minimal candidate [:] (not shown on the figure but meaning that everything is working) covers all the observations. Once the symptom <a,b> appears, using part two of the algorithm, we obtain minimal candidates [$\overline{a}$:] and [$\overline{b}$:], which implicitly cover all the candidates above line 1. Once symptom <a,c> appears, [$\overline{a}$:] remains a minimal candidate. Using the second part of the candidate generation algorithm, we create candidates [$\overline{ab}$:], [$\overline{ac}$:] and [$\overline{bc}$:] and then eliminate [$\overline{ab}$:] and [$\overline{ac}$:] because they are subsumed by [$\overline{a}$:]. This leaves us with new minimal candidates [$\overline{a}$:] and [$\overline{bc}$:], implicitly covering all the candidates above line 2 in Figure 3.4.

The algorithm described here only applies to tests that result in a symptom (i.e failed tests). This is because DeKleer bases all his reasoning on failed tests (i.e malfunctions) and ignores successful tests.

Candidate ranking

Assuming that at any stage:

- Only the minimal candidates are considered as suspect.
- Faults are assumed to be independent.
- The a priori probabilities of the components failing (using manufacturer's data or prior experience) are available.

it is easy to calculate the probability of each candidate being responsible for the symptoms. The probability of each candidate is simply the product of the individual probabilities of the components that constitute the candidate. The sum of the probabilities of all the minimal candidates can then be normalised to 1. This gives a ranking based on the probabilities of the candidates. Strictly speaking it is not even necessary to normalise the sum to 1 since the normalisation does not change the relative values of the probabilities, only the absolute ones. The pre-normalised probabilities thus already embody the ranking.

For example, given components a,b,c,d each with probability 0.01 of being down, after the symptom <a,b> there are two candidates [$\overline{a}$:] and [$\overline{b}$:] with normalised probabilities equal to 0.5. After the second symptom <a,c> there are two candidates, [$\overline{a}$:] and [$\overline{bc}$:]. Their pre-normalised probabilities are respectively 0.01 and 0.0001. Their normalised probabilities are respectively 0.99 and 0.009 and the pre-normalised values already contained the fact that candidate [$\overline{a}$:] was ten times more likely than candidate [$\overline{bc}$:].

3.4.7. Discussion

The minimal candidate strategy is the most general strategy; it implicitly looks at all the possible candidates but represents them in a compact way which means that it can be applied to systems with a very large number of components. However, as it stands, it has two major defects with respect to complex dynamic systems; it does not take into account successful tests and it does not allow the possibility of sensor faults.

3.5. Existing sensor fault strategies

While most diagnostic work in AI assumes that the sensor observations are reliable [Chan87], some existing complex systems (for example in process control) do attempt to validate the data. These methods rely on a reasoning mechanism about the sensors which

is totally separate from the reasoning mechanisms for the other faults. Some of these methods are described below.

Redundant sensors of the same kind: A very common strategy for anticipating and resolving sensor faults is to have redundant identical sensors. Then the loss of a sensor does not preclude data gathering and disagreements between identical sensors can be solved using a majority voting strategy. This redundancy is outside the control of the fault diagnoser. It is a system-specific characteristic and taken as given by the fault diagnoser.

Overlapping sensors: In most diagnostic systems, different sensors will overlap, allowing cross checking of sensor findings, confirmation of sensor findings and alternative diagnostic routes to the solution. This overlapping of sensors thus allows the combination of sensor data making it possible to deal with sensor faults if an additional mechanism is employed to resolve conflicts. Again the extent of overlapping is system-specific and taken as given by the fault diagnoser.

Sensor belief strategies: Another method used to deal with sensor faults is to assign measures of belief and disbelief to each item of sensor data [Fox83]. These measures are then combined using fuzzy logic (in a MYCIN [Shor76] fashion) to try and guide the search along the most probable path.

Sensor availability: Once the test and sensor faults are accepted as part of the fault model, it is necessary to reason about the availability of the tests and sensors [Bara86].

Discussion: The knowledge about the availability of the sensors is mainly useful when choosing a test to perform; this problem will be ignored for the time being and assume that all the sensors are available, or that there is a process keeping track of the availability of the sensors.

All the other data validation techniques are in fact probability based, in the same way that the minimum candidate method for fault finding is probability based.

In the majority voting system, the fault diagnoser implicitly chooses to accept the verdict of the majority of the tests because they are presumably more likely to make a correct observation. If there are three equal tests t_1, t_2 and t_3 with a low chance of being faulty

say p1, where t_1 and t_2 think that set A is working and t_3 thinks that A is faulty, the fault diagnoser chooses to believe t_1 and t_2 because the chance of both T1 and T2 not working ($p1^2$) is smaller than the chance of t_3 not working (p1). If the findings of t_3 are discarded, the fault diagnoser is also ignoring the small but positive possibility that t_3 made the correct observation i.e the fault diagnoser is discarding the small probability candidate(s).

Similarly in cross combining results of different tests, the fault diagnoser will implicitly choose the most probable outcome. If for example test t_1 thinks that one of A,B is faulty, t_2 and t_3 think that A is working and t_4 and t_5 think that B is working, the fault diagnoser would assume that A and B are in fact working and that t_1 made a wrong diagnosis because it is more probable that one test will fail than two.

Using measures of belief in sensors is obviously a probability based technique. The reason probabilities of failure were used in preference to fuzzy logic is that the probabilities can be coupled with the entropy measure in order to determine the best test to be performed, as described in the following Chapter. Fuzzy coefficients cannot be used in the calculation of a set's entropy, which would have made test generation much more difficult.

3.6. A new integrated sensor fault/minimal candidate approach

As discussed previously, none of the multiple fault strategies examined so far make any allowance for sensor faults; the sensor data is taken at its face value and combination of the data is made under the assumption that this data is correct. In complex dynamic systems where the sensors are part of the system, this assumption is wrong. Furthermore, the best and most general approach to multiple faults (the minimal candidate approach) ignores successful tests and bases its reasoning only on failed tests. Similarly the sensor fault methodologies only deal with sensor faults. This section, presents a new integrated mechanism for dealing simultaneously with sensor faults and other faults which takes into account successful tests.

3.6.1. Assumptions

The first assumption made here is that a test result is either "successful" or "unsuccessful". A successful test indicates that during the test, the involved elements reproduced their working behaviour and an unsuccessful test indicates that the involved elements did not behave as expected during the test. Measurements which do not operate in such a "digital" way, that is those where the outcome is not necessarily a clear-cut failure or

success, can be transformed into this digital pattern by deciding on a reasonable boundary between normal behaviour and abnormal behaviour. This boundary is system dependent and would normally reflect the knowledge of an expert in the field. For example, if the error rate of a transmission line was being measured, an error rate of over 10% could be treated as abnormal.

The second assumption is that there is no direct online way of checking whether the sensors are working. In other words there is no test that tests the sensors. This is a way of limiting the search space. If it were possible to test the sensors, there would be a recursive problem of reasoning about the accuracy of the sensor testers. An implication of this assumption is that it is never possible to prove with absolute certainty that an element is not working (or working); the deduction of an element's behaviour is based on a sensors' observations which cannot be checked (online). What is possible however is to conclude that an element is (not) working with a high probability.

The third assumption is that a test can be given an a priori probability of giving a correct result (i.e of not being faulty). This is a reasonable assumption since a sensor is ultimately an element of the system like any other (albeit with a different function). This a priori probability can be obtained by analysing the performance of the sensor over a large number of diagnostic sessions, from manufacturer's data about the reliability of the sensor, or a combination of both.

The final assumption is that faults occur independently. This is a common assumption in fault diagnosis [Hams83] , [Klee87] , [Reit87]. The proposed model could be extended to deal with dependent faults but that would require the acquisition and processing of voluminous data about the inter-fault relationships.

3.6.2. The meaning of test data

Once the possibility of test faults is accepted, the meaning of a test result has to be redefined. A failed test on set A means that the test *indicates* that there is a fault and not that the test *shows* that there is a fault. Similarly a successful test on set A only *indicates* that A is working.

3.6.3. Minimal candidate generation

As before the set of candidates is updated incrementally by applying the test results sequentially. The procedure has three steps:

1) The test finding is translated into all the minimal candidates that would justify the test finding.

2) Then each of these new minimal candidates is combined with each of the current minimal candidates, eliminating the incompatible ones and producing a new set of candidates.

3) Then all the non - minimal candidates from the new set of candidates (i.e those that are subsumed or duplicated) are eliminated leaving a new set of minimal candidates.

Test translation into candidates

A failed test t_i on elements $(e_1 \ldots e_e)$, i.e $t_i<e_1 \ldots e_e>$ is translated into the following set of minimal candidates: $\{[\overline{t_i}:], [\overline{e_1}:t_i] \ldots [\overline{e_e}:t_i]\}$

The meaning of this is clear: when a test indicates that a set of components is not working, either the test itself is faulty and nothing is known about the set of the tested components which corresponds to the first minimal candidate, or the test is working and one of the tested components is faulty which corresponds to the other minimal candidates.

A successful test t_i on elements $\{e_1 \ldots e_e\}$ is translated into the following candidates: $\{[\overline{t_i}:], [\ :t_i, e_1 \ldots e_e]\}$

The meaning of this is again clear: when a test indicates that a set of components is working, either the test itself is faulty and nothing is known about the set of the tested components which corresponds to the first minimal candidate, or the test is working and all of the tested components are working which corresponds to the second minimal candidates.

Combination of new candidates with existing candidates

As seen above, every hypothesis explaining the current symptoms is encoded in a candidate tuple [elements not working : elements working]. Similarly, the latest test to be received is translated into a set of candidate tuples. Every newly created candidate tuple is combined with every existing candidate tuple using the following algorithm:

1 - If any of the working elements of one of the tuples corresponds to any of the non working elements of the other tuple no combined element is created. This is because the combination would lead to an element on both sides of the tuple, with the illogical interpretation that the element is both working and not working.

2 - Otherwise, a new candidate tuple is created. The working set part of it is obtained by performing a union operation on the two working sets. The non working part is obtained by adding the single element of the non working part of the minimal candidate explaining the latest test to the non working part of the other minimal candidate. For both operations, if an element is duplicated, the copy is obviously eliminated from the new tuple.

The fault diagnoser is mainly interested in the left hand side of the minimal candidate tuple, i.e the non working elements that would explain all the current test findings. This is because the ultimate goal of a diagnostic procedure is to find the faulty components. The right hand side of the minimal candidate tuples (the working elements) is only used in the combinationphase, to eliminate inconsistent tuples.

A non sensor element e_i of the system under observation may be tested by any number of sensors. During a diagnostic session, some of these sensors may find that some sets containing e_i are working while others may find that some sets containing e_i are not working. Therefore the element e_i may appear on both sides of minimal candidate tuples and may be used to eliminate inconsistent tuples during combination, that is tuples that would contain the element e_i on both the working element side and the non-working element side.

A test element t_i of the system under observation is on the other hand not tested by any other elements (this is an explicitly made assumption). Therefore, the only time that test t_i will appear in a new minimal candidate tuple is when t_i is performed. It may be present in several of the existing minimal candidate tuples, but these are not combined with each other.

As a consequence, there is no point in including the test element in the working element part of the tuple. In the combination phase, it would always be compatible with the failure of any other component be it a sensor or any other component. since the assumption that a test works is only incompatible with the assumption that the same test is not working. On the other hand since the diagnostic process is interested in test failures, the test will obviously appear on the non working part of minimal candidate tuples.

The only time when a combination between the assumption that a test is working and the assumption that the same test is not working is made is if a test is repeated. However, as will be seen later, this repetition is dealt with by either overwriting the previous finding (if the repetition yields the same result), or invalidating the previous finding and replacing it with the new finding (if the repetition gives a contradictory result). Therefore, in the minimal candidate tuples, there is no need to represent the sensors on the working elements side.

Elimination of the non-minimal candidates

Once all the tuples have been created, all those that are subsumed or duplicated are eliminated. The rest form the new set of minimal candidates.

Candidate ranking

The existing candidates all have a probability attached. The probabilities of the new candidates created by test t_i are easy to calculate; the tuple $[\overline{t_i}:]$ has an a priori probability of failing $p(t_i)$; the probabilities of the other new tuples are then normalised to $1 - p(t_i)$. Assuming fault independence, the probabilities of the final minimal candidates are obtained by multiplying the probabilities of their constituent candidates and normalising all the final probabilities to 1.

Examples

Example 1

The system consists of tests t_1, t_2, t_3, t_4, t_5 performed in that chronological order, and two elements a and b. The test results are:

t_1<a,b> ; t_2/a/ ; t_3/a/ ; t_4 /b/ ; t_5 /b/

After t_1, the tuples are $[\overline{t_1}:]$, $[\overline{a}:]$ and $[\overline{b}:]$.

The result of t_2 is translated into $[\overline{t_2}:]$, and [:a]. Combining $[\overline{t_2}:]$ with the existing tuples (from t_1), gives: $[\overline{t_1 t_2}:]$, $[\overline{a t_2}:]$ and $[\overline{b t_2}:]$. Combining [:a] with the existing tuples gives: $[\overline{t_1}:a]$ and $[\overline{b}:a]$ (the combination of $[\overline{a}:]$ and [:a] leads to a contradiction). Finally $[\overline{b t_2}:]$ is eliminated because it is subsumed by $[\overline{b}:a]$ and $[\overline{t_1 t_2}:]$ is eliminated because it is subsumed by $[\overline{t_1}:a]$. This leaves three minimal candidates: $[\overline{a t_2}:]$, $[\overline{t_1}:a]$ and $[\overline{b}:a]$.

The result of t_3 is translated into $[\overline{t_3}:]$, and [:a]. Combining $[\overline{t_3}:]$ with the existing tuples gives: $[\overline{a t_2 t_3}:]$, $[\overline{t_1 t_3}:a]$ and $[\overline{b t_3}:a]$. Combining [:a] with the existing tuples gives: $[\overline{t_1}:a]$ and $[\overline{b}:a]$. This set of 5 tuples is minimised to: $[\overline{a t_2 t_3}:]$, $[\overline{t_1}:a]$ and $[\overline{b}:a]$.

The result of t_4 is translated into $[\overline{t_4}:]$ and [:b]. The combination of $[\overline{t_4}:]$ with existing tuples yields: $[\overline{a t_2 t_3 t_4}:]$, $[\overline{t_1 t_4}:a]$ and $[\overline{b t_4}:a]$. The combination of [:b] with existing tuples yields: $[\overline{a t_2 t_3}:b]$ and $[\overline{t_1}:ab]$. This set of 5 tuples is minimised to: $[\overline{b t_4}:a]$ $[\overline{a t_2 t_3}:b]$ and $[\overline{t_1}:ab]$.

Finally the result of t_5 is translated into $[\overline{t_5}:]$ and $[:b]$. The combination of $[\overline{t_5}:]$ with existing tuples yields: $[\overline{bt_4t_5}:a]$, $[\overline{at_2t_3t_5}:b]$ and $[\overline{t_1t_5}:ab]$. The combination of $[:b]$ with existing tuples yields: $[\overline{at_2t_3}:b]$ and $[\overline{t_1}:ab]$. This set of 5 tuples is minimised to: $[\overline{bt_4t_5}:a]$, $[\overline{at_2t_3}:b]$ and $[\overline{t_1}:ab]$.

Without even giving specific a priori probabilities of failure, and just assuming that the sensors have a smaller probability of failure than other elements, it is clear that after t_1 the most probable faulty elements were a or b, after t_2 and particularly after t_3, b became the most probable fault, and after t_4 and t_5, t_1 became the most likely fault. This example is summarised in Table 3.1.

Tests t1 <a,b> t2 /a/ t3 /a/ t4 /b/ t5 /b/		
Test	Current tuples	New test tuples
t_1	$[:]$	$[\overline{t_1}:],[\overline{a}:],[\overline{b}:]$
t_2	$[\overline{t_1}:],[\overline{a}:],[\overline{b}:]$	$[\overline{t_2}:],[:a]$
t_3	$[\overline{at_2}:],[\overline{t_1}:a],[\overline{b}:]$	$[\overline{t_3}:],[:a]$
t_4	$[\overline{at_2t_3}:],[\overline{t_1}:a],[\overline{b}:a]$	$[\overline{t_4}:],[:b]$
t_5	$[\overline{at_2t_3}:],[\overline{t_1}:a],[\overline{bt_4}:a]$	$[\overline{t_5}:],[:b]$
result	$[\overline{at_2t_3}:],[\overline{t_1}:a], [\overline{bt_4t_5}:a]$	--------

TABLE 3.1; Combination example 1

Example 2

The system has three identical tests t_1, t_2 and t_3, all testing elements a,b and c and the results of the tests are (in that chronological order):

t_1<a,b,c> t_2/a,b,c/ t_3/a,b,c/

After t_1, the tuples are $[\overline{t_1}:\ ]$, $[\overline{a}:\ ]$, $[\overline{b}:\ ]$ and $[\overline{c}:\ ]$.

t_2 is translated into $[\overline{t_2}:\ ]$ and [:abc]. Combining $[t_2:\ ]$ with the existing tuples yields: $[\overline{t_1 t_2}:\ ]$, $[\overline{at_2}:\ ]$, $[\overline{bt_2}:\ ]$ and $[\overline{ct_2}:\ ]$. Combining [:abc] with the existing tuples yields: $[\overline{t_1}$:abc]. Minimising these 5 tuples gives: $[\overline{at_2}:\ ]$, $[\overline{bt_2}:\ ]$, $[\overline{ct_2}:\ ]$ and $[\overline{t_1}$:abc].

t_3 is translated into $[\overline{t_3}:\ ]$ and [:abc]. Combining $[t_3:\ ]$ with the existing tuples yields: $[\overline{at_2 t_3}:\ ]$, $[\overline{bt_2 t_3}:\ ]$, $[\overline{ct_2 t_3}:\ ]$ and $[\overline{t_1 t_3}$:abc]. Combining [:abc] with the existing tuples yields: $[\overline{t_1}$:abc]. Minimising these 5 tuples yields: $[\overline{at_2 t_3}:\ ]$, $[\overline{bt_2 t_3}:\ ]$, $[\overline{ct_2 t_3}:\ ]$ and $[\overline{t_1}$:abc].

Again without even specifying any of the a priori probabilities of failure, and just assuming that the three sensors have an identical probability of failure, it is clear that the most probable fault is t_1. This example is summarised in Table 3.2.

Tests t1 <a,b,c> t2 /a,b,c/ t3 /a,b,c/		
Test	Current tuples	New test tuples
t_1	[:]	$[\overline{t_1}:\ ]$,$[\overline{a}:\ ]$,$[\overline{b}:\ ]$,$[\overline{c}:\ ]$
t_2	$[\overline{t_1}:\ ]$,$[\overline{a}:\ ]$,$[\overline{b}:\ ]$, $[\overline{c}:\ ]$	$[\overline{t_2}:\ ]$,[:abc]
t_3	$[\overline{at_2}:\ ]$,$[\overline{bt_2}:\ ]$, $[ct_2:\ ]$, $[\overline{t_1}$:abc]	$[\overline{t_3}:\ ]$,[:abc]
result	$[\overline{at_2 t_3}:\ ]$,$[\overline{bt_2 t_3}:\ ]$, $[\overline{ct_2 t_3}:\ ]$,$[\overline{t_1}$:abc]	------

TABLE 3.2; Combination example 2

3.6.4. Discussion

Example 1 in fact corresponds to the combination of data from overlapping sensors which is a major method for dealing with sensor faults as mentioned in Section 3.5. However, there it was just an intuitive approach; here the whole combination is performed using an explicit probabilistic methodology which makes it much easier to decide whether a sensor fault has occurred.

Example 2 is in fact an example of the voting method for sensor validation described in Section 3.5. The only difference is that the earlier voting method automatically eliminates the less probable candidates while the proposed methodology keeps the less probable candidates as well. This does not preclude the elimination of such candidates, but allows full control over the elimination decision.

Dealing with a priori probabilities of failure is akin to having a measure of belief in a sensor as mentioned in section 3.5. The fuzzy logic method described there is used to guide the search in what currently seems to be the best search path. The proposed methodology on the other hand makes it possible to look at all the suspects in parallel. Furthermore, since a sensor can be used to perform different tests, two possible measures of belief should be taken into account; an overall measure of belief in a sensor which affects all the tests performed by a sensor and a measure of belief in a sensor when it performs a particular test. The two measures are in fact closely related; diminishing the belief in a particular sensor is akin to diminishing the belief in all the tests performed by that sensor in the same proportion. The proposed methodology already gives a probability of failure to individual tests. It is easily extended to dealing with the degree of confidence in a sensor by attaching an additional parameter to all the tests that are performed by a single sensor, and using that parameter in the calculation of the a priori probability of failure of the test.

Finally this approach is an extension of the DeKleer approach which is based on failed tests only. Using the sensor faults, positive tests could be accepted with the alternative that the test reading is faulty. The additional fault dimension (test fault).

The proposed methodology covers and extends all the major techniques for dealing with sensor faults. All these techniques are unified in one homogenous and consistent approach, which deals well with conflicting data. Furthermore the whole technique is integrated with the multiple fault approach which means that both multiple and sensor

faults can be dealt with without any backtracking.

Treating test and other faults in an identical way may sound controversial because of their differing functions. However, during the fault isolation phase, the aim is to try and apportion the blame for the current symptoms; the blame is apportioned in the same way (probability of failing) and from the same data; therefore intuitively there is no reason for treating them separately. It is only during diagnosis and repair that the treatments may differ, but that is another issue altogether.

3.7. Dealing with state changes

The fault model so far assumes that the system is in the same state from the beginning of the diagnosis until its end. All discrepancies and contradictions between the test data are thus explained by test faults. However, since the system under observation is dynamic, state changes may occur at any time between the moment the first data about a malfunction is received (beginning of diagnosis) and the end of the diagnostic process. Discrepancies and contradictions between test data can then also be explained by the fact that the data refer to different states. This section develops a model to deal with state changes and particularly to distinguish between test faults and state changes.

At first sight the model presented so far could be extended to deal with state changes, by giving another interpretation to test data. For example, if test t_1 found that the set (a,b,c) is faulty (t_1<a,b,c>), that could mean that either t_1 is faulty, a is faulty, b is faulty, c is faulty or there has been a state change, i.e state v_1 is not valid any more (thus invalidating the accumulated test data). The new candidates would thus be: $[\overline{t_1}:\ ]$, $[\overline{a}:\ ]$, $[\overline{b}:\ ]$, $[\overline{c}:\ ]$ or $[\overline{v_1}:\ ]$. If subsequently a test t_2 found that (a,b) is faulty (t_2<a,b>), by applying the minimal candidate generation algorithm, we would obtain the new minimal candidates: $[\overline{a}:\ ]$, $[\overline{b}:\ ]$, $[\overline{t_2 c}:\ ]$, $[\overline{t_1 t_2}:\ ]$, $[\overline{v_1 t_2}:\ ]$, $[\overline{v_2 t_1}:\ ]$, $[\overline{v_2 c}:\ ]$, $[\overline{v_1 v_2}:]$. This approach was rejected for several reasons:

- It would be very difficult to get the probability of a state change (associated with suspect v_1 for example) for each test performed. Such probabilities are quite difficult to calculate for individual elements and more importantly, this probability depends on the length of time the current diagnosis will take. The longer it takes, the higher the probability that the state will change. At the beginning there is no way to know how long the diagnosis will take. That depends on the particular fault that has occurred and on the testing sequence, and that is exactly the data that we are trying to establish.

- The suspect $[\overline{v_1 v_2}:]$ does not contain the fact that v_2 is a state subsequent to v_1, if test t_2 was performed after test t_1. Therefore some potentially useful information has been lost and there is no way of expressing it inside the model. This highlights the need for a model that has at least some knowledge of time.

- when t_1 suggested that (a,b,c) was faulty, it made an observation on the state of subset (a,b,c) of the system. Similarly t_2 made an observation on the state of subset (a,b). The only time state v_1 becomes invalid is if there is a state change within the set (a,b,c) and similarly for v_2 and (a,b). The model does not capture this locality of the states; the candidate $[\overline{s_1 s_2}:]$ does not contain any information about the subsets to which the state refers.

Therefore an alternative and supplementary mechanism for dealing with state changes and temporal information is needed.

The proposed model makes the assumption that bursts of short term fluctuations in test findings are already eliminated by the tests themselves. In other words, a test will attempt to establish the failure of an element for a significant period of time before declaring that there is a failure. This significant period of time is system and test dependent. It should help the test to distinguish between a working but relatively slow component and a failed component. For example a test that attempts to make a connection to an element of the system should make several attempts before signaling that the connection failed. This assumption improves the performance of the proposed system because the knowledge revision phase caused by a state change is computationally expensive. Current sensor design normally incorporates this type of filtering [Chan87].

3.7.1. Time information during fault isolation

In a typical dynamic system the fault management configuration consists of:

a) A central fault manager which receives all the fault reports, analyses them and combines them to find and repair the faults.
b) Sensors which are part of the system, whose duty is to perform tests and send the results to the fault manager. The fault manager makes test requests to the sensors. The sensors receive requests via the system, (usually) perform the tests via the system and use the system to send the test data results to the fault manager. The time taken by the data between a sensor and the fault manager is unpredictable.

All the other elements which perform a variety of functions, such as transmitting data, providing particular services etc.

This is depicted in Figure 3.5.

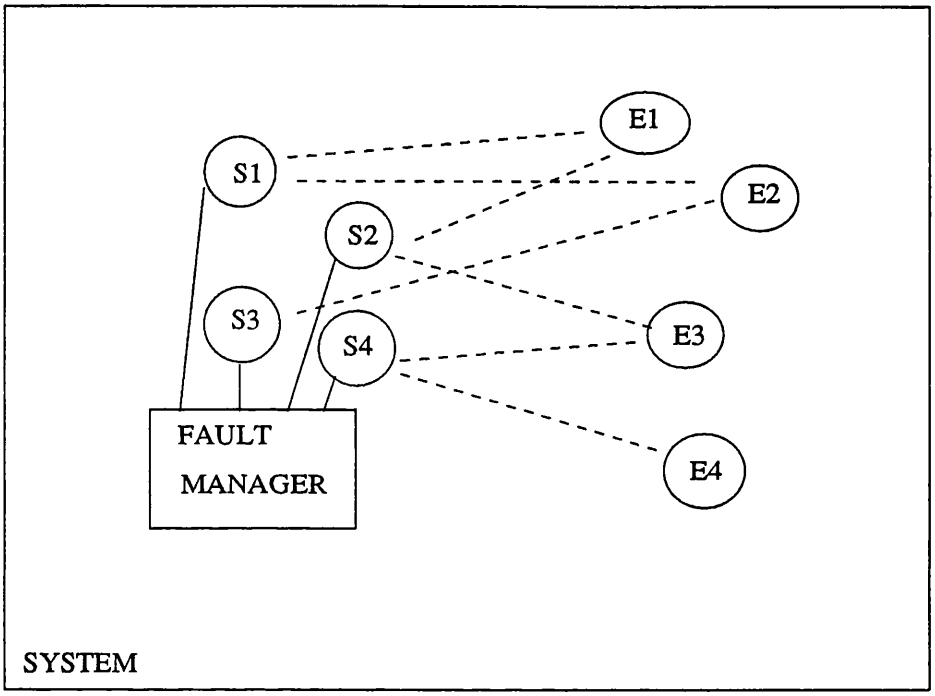


FIGURE 3.5 Typical dynamic system

The amount of time information in such a system is very limited. This point can be illustrated with a very simple example of a system consisting of one fault manager, two tests t_1 and t_2 and a set of elements a,b,c which is tested by both tests.

The testing cycle is as follows. First the fault manager requests a test at time T1. This request reaches the sensor at time T2. The sensor then performs the test which takes an amount of time T3 and receives the result at time T4. The result is sent back to the fault manager which receives it at time T5. This sequence is depicted in Figure 3.6 and contains two simplifying assumptions: that there is no delay between the time the sensor receives a test request and the time it performs the test and that there is no delay between the time the sensor receives the test result and the time it sends the result to the fault manager.

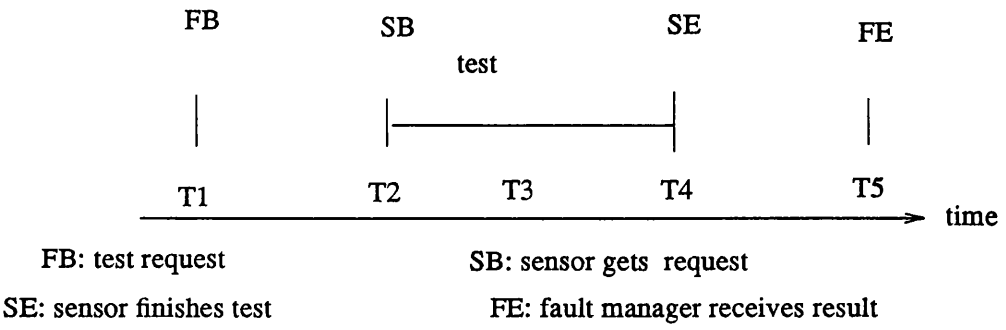


FIGURE 3.6 Testing cycle

Let the fault manager make a simultaneous request to t_1 and t_2 to perform a test on the set (a,b,c) If the sensors do not provide the exact time at which they performed the test, the only temporal information the fault manager has is the request time T1 and the arrival times T5(1) from sensor t_1 and T5(2) from sensor t_2. The fact that T5(1) precedes T5(2) (for example) does not mean that sensor t_1 performed its test before sensor t_2, because of the variable delays in the data transmission. Any of the time sequences shown in Figure 3.7 could explain the event sequence from the point of view of the fault manager.

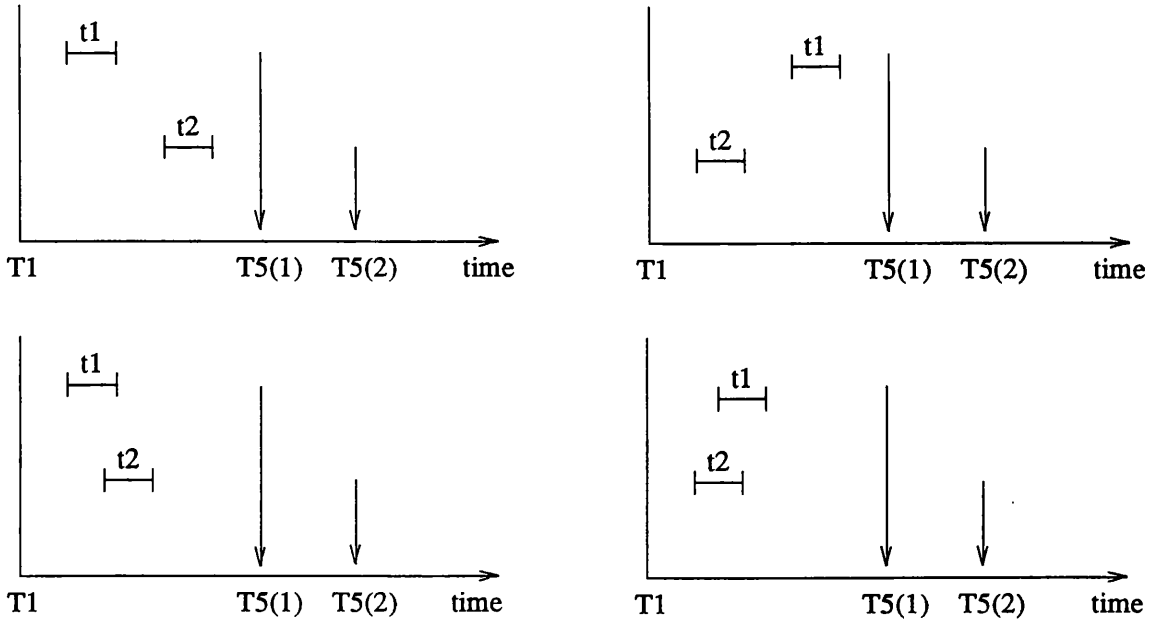


FIGURE 3.7 Possible testing sequences

Therefore if the sensors disagree on the status of the set (a,b,c) the fault manager does not have a way to determine which of the two results refers to the current state of the set (assuming that both sensors are working). If the sensors send temporal information of the type "this test was performed at time T", the problem becomes whether the sensors have exactly the same idea of time. To be able to determine the time of the test, a sensor must normally have an internal clock. It is notoriously difficult to synchronise the clocks of several independent objects. If the clocks between the sensors (and the fault manager) are not synchronised, the temporal information that the fault manager receives is insufficient; the time values referred to by the sensors have no global significance and cannot be compared. In this case the problem is exactly the same as if we didn't have any temporal information at all. Even if the sensors have internal clocks and the clocks are synchronised, it may not be possible to determine the sequence of events. This is because a test is normally performed on a set of elements and the sensor may know the time it began the test and the time it received the result but not necessarily the sequence of events between the two. This can be illustrated with the example of Figure 3.8, representing a very simple computer network.

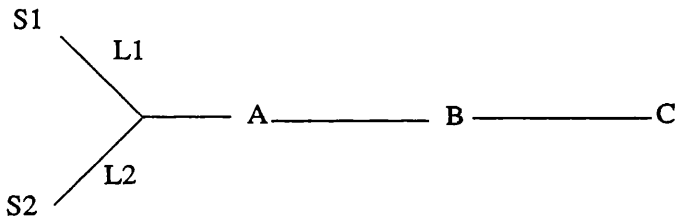


FIGURE 3.8 A simple system

The sensors s1 and s2 both test element C (and therefore elements A and B as well since they are en route to C). At 12.00.00, s1 performs the test on A,B,C and finds that they are working at 12.00.04. At 12.00.01 s2 performs the test on A,B,C and finds that they are not working at 12.00.06. The obvious explanation is that the state recorded by s1 precedes that recorded by s2, and that one of A,B,C went down between the two tests. However another explanation could be that the test performed by s1 got delayed by traffic eg at L1 while s2's test did not and that therefore the two states are reversed.

The problem with the last case was that the time intervals of the two tests overlapped. If a test is requested and performed after the result of another one is received, then its observation must refer to either the same system state or a subsequent system state. The

fault manager can perform this control and obtain the temporal information by sequencing the tests, thus eliminating the need for clock synchronisation.

Unrequested data (spontaneously sent by sensors) pose the same problem because it may be difficult for the fault manager to place it in the time spectrum.

The proposed model will thus make the assumption that the only temporal information available to the fault manager is the sequencing information; a test requested and performed after the result of another test is received must refer to the same or a subsequent system state.

3.7.2. State change manifestation

In the first instance, let the fault model be based on the assumption that there are no test faults, i.e that all the data sent by the sensors are correct.

Then a state change in the system manifests itself when it becomes impossible to explain the current set of test data by any fault set. This incompatibility of the test data is equivalent to the statement: " For some unsuccessful test on a set A, all the elements of set A have been established as working by other tests". For example, if test t_1 finds the symptom <a,b,c>, the minimal suspects become $[\overline{a}:]$, $[\overline{b}:]$, $[\overline{c}:]$. If t_2 then finds that a and b are working, the only minimal candidate left is $[\overline{c}:]$. Finally if t_3 finds that c is working there are no candidates left. No fault set can explain the test findings and a state change must have taken place (since we assumed that the sensors are working). Similarly at the beginning of a diagnosis, all the elements are believed to be working and the first failed test leads to this type of incompatibility, indicating that a state change has occurred.

3.7.3. State change /test fault duality

Now the assumption that there are no test faults is relaxed. As far as the state changes are concerned, they still manifest themselves in the same way; if a state change occurs, it will ultimately lead to an incompatibility between the test data. However any such incompatibility can now also be explained in terms of a test fault, i.e that the data provided by the test is itself wrong. To take again the example from the previous paragraph (and by applying the algorithms developed in section 3.6.3):

After t_1 finds the symptom <a,b,c>, the minimal suspects are : $[\overline{a}:]$, $[\overline{b}:]$, $[\overline{c}:]$, $[\overline{t_1}:]$.

After t_2 finds that /a,b/ is working, the new minimal suspects become $[\overline{c}:ab]$, $[\overline{t_1}:ab]$, $[\overline{at_2}:]$ and $[\overline{bt_2}:\ ]$.

After t_3 finds that c is working, the new minimal suspects become: $[\overline{ct_3}:ab]$, $[\overline{t_1}:abc]$, $[\overline{at_2}:c]$ and $[\overline{bt_2}:c]$.

In a way, the situation is different from the previous paragraph. There is now a plausible explanation for the set of data (in fact four plausible explanations). There is no longer a fundamental contradiction of the type: "there are symptoms with no possible explanations". However, the crucial point is that each and every candidate postulates a test fault. This is equivalent to an incompatibility between the test data. This incompatibility may indicate a state change. Alternatively, it can mean that one of the tests sent false data. This duality between sensor faults and state changes makes the diagnostic task much more difficult, particularly since there is no way of checking whether a test is working correctly (except indirectly via its data).

3.7.4. A model for dealing with state changes

A model to deal with state changes in the system under observation has to provide three functions:

- a method for triggering suspicions that a state change has occurred. This is because systems do not normally indicate directly that a state change has occurred; this information has to be deduced from sensor data;

- a method for confirming or denying suspicions that a state change has occurred. This function is necessary because, as seen above, the combined test data does not indicate unequivocally that a state change has occurred;

- a method for recovering from confirmed state changes by updating the beliefs about the state of the system in a manner that reflects the perceived state change.

3.7.4.1. Suspecting a state change

As seen above, a state change usually manifests itself indirectly via the data. Therefore it is necessary to formulate a state change suspicion policy, based on the data. This suspicion policy will be used to trigger a reevaluation of the currently believed data.

Permanent suspicion approach to state changes

Ultimately there is no way to be sure that any of the test data received earlier is still valid. The only theoretical way to try to determine the current state is to keep repeating

the tests already performed. This is equivalent to a permanent suspicion that a state change has occurred. In practice such an approach cannot be applied because it is infinite. Thus suspicions about state changes have to be guided by heuristics.

Data incompatibility suspicion approach to state changes

As seen in Section 3.7.2, the state change in the system manifested itself when every single minimal candidate postulated a test fault. This fact can be used as a trigger for state change suspicion. This would imply an analysis of all the minimal candidates at the end of every testing cycle to see whether each one of them indeed postulates such a test fault.

Information content suspicion approach to state changes

Every diagnostic procedure starts with a relatively large number of possible explanations for the symptoms, with little to distinguish between them. As the diagnostic process progresses, it may be expected to differentiate better and better between the candidates, since it is learning about the system state through the test data. A good heuristic approach is thus to start suspecting a state change when some measure of the information content of the test data shows that in the last testing cycle, some loss of information about the system has taken place. This is intuitively appealing; when a state change occurs, some of the information previously obtained and used becomes wrong. Therefore the knowledge about the system, embodied in the current list of minimal suspects and the information content of this list can be expected to diminish. This is only a heuristic because the converse is not necessarily true; if the information content does indeed diminish, it could be because a less likely fault or combination of faults has occurred, rather than because of a state change.

One possible way of measuring the evolution of the information content in the set of suspects is to look at the consecutive probabilities of the most likely candidate. If the probability of the most likely candidate after the current testing cycle is smaller than the probability of the most likely candidate after the previous testing cycle, we could suspect a state change. This can be illustrated with the following example.

The testing sequence is:

t1 <a,b>, t2 <a,c> t3 /a/ with the probabilities:

pdown(a)=0.1, pdown(b)=0.1, pdown(c)=0.1, pdown(t1)=0.01, pdown(t2)=0.01, pdown(t3)=0.01.

After t1, the suspects are [$\overline{a}$:], [$\overline{b}$:] and [$\overline{t1}$:] with approximate respective probabilities

0.48 0.48 and 0.04.

After t2, the minimal candidates become [$\overline{a}$:], [$\overline{bc}$:] [$\overline{t2b}$:], [$\overline{t1c}$:] and [$\overline{t1t2}$:] with approximate respective probabilities 0.9 0.09 0.009 0.009 and 0.0009. The most probable candidate has gone from 0.48 to 0.9, so according to the proposed measure there has been a gain of information and no state change is suspected.

After t3, the minimal candidates become : [$\overline{bc}$:], [$\overline{t2b}$:a], [$\overline{t1c}$:a] and [$\overline{t3a}$:] with approximate probabilities 0.75 0.075 0.075 0.075. Now the most probable candidate has gone down from 0.9 to 0.75. This is an information loss according to the chosen criterion and leads to the suspicion of a state change. For example an intermittent fault in element a which cleared itself after test t2 would account for the symptoms. However, any of the existing minimal candidates can also account for the fault. Therefore the indication of information loss does not imply a state change; it should however trigger the suspicion of a state change.

A better measure of the information contents of the set of minimal candidates can be obtained by calculating its entropy. The concept of entropy is discussed in Chapter 4 where it is used in order to decide which test to perform next. Here only one of its important properties is mentioned, namely that the better the differentiation between the candidates the smaller the entropy. Therefore, if the new entropy is larger than the previous one, some information has been lost and a state change can be suspected. The entropy approach in a way subsumes the previous approach. When the most probable candidate increases in probability the entropy of the set will decrease and vice versa.

Finally, both the entropy approach and the highest probability candidate approach subsume the data incompatibility approach. Assuming that the sensors are less likely to fail than the other elements, in the transition from the previous list of minimal candidates to the current list of minimal candidates which all postulate a test fault, some minimal candidate which did not postulate a test fault must have been eliminated. This candidate would have had a higher probability of failure than the current candidates. There would have therefore been an increase in the entropy of the list of the minimal candidates and a fall in the probability of the most likely candidate.

Any of the three approaches discussed (test data inconsistency, change in the probability of the most likely candidate, entropy difference) can be used as a basis for state change suspicion. In the implementation (discussed in Chapter 6) the test data inconsistency approach was used; this approach only suspects state changes when there is irrefutable

evidence that either a test fault or a state change has occurred. As such, it is more suitable for systems where the state changes during diagnosis occur infrequently. The other two approaches are more suspicious and will trigger the test repetition phase more frequently.

3.7.4.2. Testing/confirming a state change

Once a state change is suspected, whatever the methodology used, it is necessary to check whether a state change has indeed occurred. Assuming for the moment that a faulty test will give a permanently false result (i.e will not vary its result randomly when repeated), it is certain that a state change has occurred if when repeated a test obtains a different result. Conversely, it is certain that no significant state change has occurred if, when repeated, all the tests reproduce the same results. A *significant state change,* is defined as a state change among one or more of the elements which have been part of our diagnosis since the system began. Such a state change is significant because it invalidates some of the believed data and therefore affects the whole reasoning process. Any other state change may affect the diagnostic process from that moment onwards (by changing the test results performed later), but does not affect the current reasoning or beliefs. For example, in a system consisting of components a,b,c,d,e,f, let t1 find the symptom <a,b,c>. At this point, a state change among d,e,f is not significant because it does not affect the belief about the current state (which is that there is a fault among a,b,c). A state change among a,b,c is significant because it invalidates the basis of the diagnosis (the results of t1).

The operation of checking whether a significant state change has occurred is performed by repeating the tests which supplied the data that is currently believed in the same chronological order in which they were performed initially. This chronological order is preserved for two reasons; it enables a revision of the believed test data using the chronological information and permits the reutilisation of previously deduced data as described in the following section. The test repetition is performed until a test that has changed its result is found or until all the previous findings have been reconfirmed. The reason the test repetition is performed only until a state change has been confirmed is as follows; once a state change has been established by one test, the tests that followed it are not necessarily the best tests any more. It will be up to the testing module to determine what the next best test is, once the suspect data have been eliminated.

3.7.4.3. Recovering from a state change

At the moment the state change suspicion phase was triggered, tests t_1, t_2, ..,t_n had already been performed.

If all these tests give the same result when they are repeated, no significant state change has occurred and the diagnosis can continue normally using the test data that triggered the suspicion of a state change (as seen in the previous section).

Let us assume that when the tests t_1 ..t_n are being repeated in the same chronological order, test t_i yields a different result. At this stage the following data is available:

1) The data from the tests t_1 .. t_{i-1} which have been repeated and which have given the same result as before;
2) The data from test t_i which was repeated and which found that a state change has occurred;
3) The data from the tests t_{i+1} .. t_n which have not be repeated;
4) The chronological order of all the tests in 1, 2 and 3;
5) The deductions performed so far on the basis of data received from t_1 .. t_n before the state change was suspected. These deductions are recorded by the ATMS in the datum part of the ATMS node, as described in the previous Chapter.

The data from tests t_1 .. t_{i-1} can still be believed since they have been confirmed by the repeated tests. The deductions based on those data are contained in the ATMS node N_{i-1}, which was created when the data from test t_{i-1} was being combined with the data from tests t_1 .. t_{i-2}.

Continuing to believe in these data may mean believing in data that are not valid any more. If a state change has occurred during the repetition phase, some of the repeated data may now be invalid. However, the likelihood of that happening is very small; furthermore this inconsistency would be eliminated the next time a state change is suspected and the tests repeated. The alternative is to use the algorithms applied to the data from the unrepeated tests t_{i+1} .. t_n (described below) and apply them to the data from the repeated tests as well. If this is done, there is hardly any need for temporal information and logic; all the knowledge revision is then based on the properties of the data sets. Even the repetition of the tests in the chronological order then becomes unnecessary. The option chosen here is however consistent with the general approach that state changes are considered only when there is a reason for suspecting them, not every time they may occur.

The major remaining problem is what to do with the data from tests t_{i+1} .. t_n.

The first alternative is to also repeat the tests t_{i+1} .. t_n. This is correct but not optimal because given the new result of test t_i, it is very unlikely that these tests are again the best tests that can be performed, whatever the test selection method used.

A second alternative is to discard all the data from tests t_{i+1} .. t_n and continue the diagnosis by combining the result of t_i with the result held in ATMS node N_{i-1} and choosing a new best test. This again is correct but not optimal because it eliminates some data from t_{i+1} .. t_n which have no reason to be disbelieved as shown in the next section.

The best method is to discriminate between the data from tests t_{i+1} .. t_n that can be believed and the data that has become suspect. This can be done using the following steps:

1) First, the data referring to any subset of the system that is disjoint from the subset tested by t_i (which noticed a state change) can still be believed. For example if t_i found a state change in subset (a,b,c) and t_{i+1} had observed the state of subset (d,e), the data from t_{i+1} can still be believed. This is because there is no a priori reason for thinking that a state change among (a,b,c) will affect (d,e) (under the independence of faults assumption).

2) Of the rest of the data, the data which are compatible with the particular state change that has occurred can also be kept. t_i may have observed one of two types of state changes; either the set it tests went from a non working state to a working state, or it went from a working state to a non working state. In the first case, all the data referring to working subsets can be kept; they shouldn't have been affected by the state change. Similarly in the second case, all the data referring to non working subsets can be kept. For example, if t_i first found that (a,b) is not working and then when repeated that it is working, a test that initially found (a) to be working should not be affected.

3) The rest of the data from tests t_{i+1} .. t_n, which was not kept under categories 1) and 2) above is then eliminated.

Once this belief revision phase is finished, the data from test t_i (which noticed the state change) and the still believed data from the unrepeated tests t_{i+1} .. t_n (as defined above) can be applied to the deductions based on the data from tests t_1 .. t_{i-1}, contained in ATMS node N_{i-1}.

Finally the chronological order is altered; the repeated tests t_1 ..t_i are now subsequent to the believed but unrepeated tests in t_{i+1} .. t_n.

The diagnosis can then continue normally.

3.7.4.4. Algorithm refinements

Relaxing the persistent fault assumption

In the section on the confirmation of a state change, the assumption was made that if the test is faulty it remains faulty in a persistent way during the whole period of the diagnosis, i.e it repeatedly gives wrong information. The statement that if a test is repeated and gives a different result, there must have been a state change was based on that assumption. However it is also possible, that the sensor is faulty and that it is giving random (unpredictable) results. Such a sensor could repeatedly trigger the state suspicion phase. In order to avoid this situation, it is possible to keep track of the number of times a particular sensor triggers the state suspicion phase and ignore data from sensors that "change their mind" too often during a particular diagnostic session. Such counters could also be part of a learning mechanism concerning the behaviour of sensors over a longer period of time.

Terminating the diagnosis

Because of the possibility of state changes, it makes sense to end the diagnosis with a test that confirms some of the symptoms. Without state changes, this additional precaution would not be necessary. For example in [Gene84], where the system under observation is assumed to be static, if a test found a set of suspects, and all but one of those are found to be working, the last element is automatically assumed to be down. Because the systems under consideration here are dynamic, and it is natural to assume that most components spend most of their time working, it is useful to make an additional test at the very end of the diagnosis in order to confirm the symptom. The simplest way to perform this is to pick up any test that tests at least one of the components of the most likely suspect. An alternative method based on entropy is described in Chapter 4. Briefly, this method relies on the biasing of the entropy measure (used to evaluate the tests) in favour of unsuccessful tests, thereby encouraging the selection of tests that are more likely to fail. Finally, it is of course possible to use shallow knowledge which would link every possible set of suspects with a test (or tests) which can confirm that the set of suspects is indeed faulty. Whichever method is used, the diagnostic procedure should always aim to finish on a

confirmation of some of the symptoms.

3.7.5. Other approaches to temporal reasoning

The zero option

A common rule-of thumb method for dealing with state changes is to restart the diagnosis from the beginning as soon as the test data become incompatible. This approach is unsatisfactory because:

- it does not allow the possibility of test faults. As seen before, both test faults and state changes can manifest themselves in an identical way. They can only be distinguished by repeating the tests with contradictory data, not by ignoring the previous data;
- it does not use the sequential time information available when a sequence of tests is performed.

Representation based on the notion of states

When a system is observed at regular intervals, each observation can be treated as a snapshot of the system state at a particular moment. Therefore the long term behaviour of the system is recorded as a series of states. This type of view is very common in medicine [Long83]; a doctor tends to see a patient's condition as a series of states where each state is characterised by the data collected in one snapshot (temperature, heart beat, breathing ...). The current state can then be interpreted in the light of current symptoms and the previous symptoms.

This method is successful if the data arrive in discrete intervals, i.e if it is obvious to which state the data are referring. If the data do not clearly fit into a state or fall between two states, the approach either discards them (as being too old) or updates the current state (i.e assumes that the data are current). Thus this method does not really address the continuous arrival of data; it basically needs the information about the state to which every datum belongs.

Representation based on dates and before/after chaining

Kahn and Gorry [Kahn77] have described a representation of time based on before/after reasoning chains. This knowledge is embodied in a time specialist which can answer three types of questions; did event X happen at time T, when did X happen, and what happened at T.

This method is probably the closest to the one developed in this Chapter. However its emphasis is on the absolute times at which events happened while the proposed method is mainly concerned with the relative time ordering of the events.

Allen's time reasoning

Allen [Alle83], extends the time specialist by providing a set of time predicates which can unambiguously describe the time relations between any two time intervals. The time functions provided are equal (same time interval), before (A before B), meets (A consecutive to B) , during (A included in B) , overlaps (A starts before B but B starts before A finishes), starts (A and B start at the same time but A finishes first) and finishes (A and B start at the same time but B finishes first). These relationships are maintained in a network of nodes and new relationships are deduced and propagated using the existing ones and a set of rules.

This method becomes too expensive for the amount of time information available in a dynamic system and for the amount of time reasoning that can be performed in the short term fault isolation. Furthermore it does not offer any help with determining the consistency of the data, nor any help with dealing with the test fault/state change duality and the revision of knowledge.

Formal models

McDermott [McDe82] and Fusoaka [Fuso83] have developed formal logics for temporal systems. Using such temporal logics one can for example perform deep knowledge descriptions of the systems and their behaviours.

These formal logics have a longer term horizon; they attempt to fully capture and describe the behaviour of the system over a period of time. For fault management purposes they can be used to record degradations in the equipment over time, analyse longer term trends, etc. They can therefore feed into the fault isolation model by for example tuning the a priori probabilities of failure of the different components. However it seems neither necessary nor possible to perform complicated temporal logic in short term fault isolation.

3.8. Summary

This Chapter developed what we believe to be the first general fault model to deal with single, multiple and sensor faults as well as state changes in dynamic systems. The main characteristics of the proposed system are:

- a uniform approach to single, multiple and sensor faults based on probabilistic information about component failures that extends and improves on current approaches in the area.
- a time reasoning element based on the suspicion about state changes, confirmation of state changes and minimal belief revision of current data.

The methodology used is not based on any shallow knowledge about the system, and does not make the usual restricting assumptions about the nature of the faults (such as single faults only) and the dynamics of the faults (such as no state changes), making it ideal for a variety of systems.

Other approaches to the problems discussed were also analysed and the reasons why they were not considered applicable were given.

CHAPTER IV

TEST SELECTION OPTIMISATION DURING FAULT ISOLATION

4.1. Introduction

During a diagnostic session, a set of symptoms will have been discovered. The possible interpretations of these symptoms are embodied in the set of minimal candidates, as seen in the previous Chapter. Given such a list of minimal candidates, new tests must be performed in order to discriminate further between the different suspect elements. The choice and the quality of the available tests in the system are of course paramount and will be a major factor in determining the performance of the testing algorithm in particular and the fault diagnoser in general. They are thus a very important research area. However, from the test selection optimisation point of view, the choice of the tests in a system at a particular moment is *given*. This Chapter develops a new test selection optimisation method particularly suitable for complex dynamic systems. In such systems, it is normally difficult to predict in advance what the best test will be, given every possible system state. The methodology therefore uses test optimisation algorithm based on the application of the entropy measure to estimate the effect of each test on the list of suspects and choose the test which is expected to find the fastest route to the solution.

It is important to stress again the difference between test *generation* and test *optimisation*. The first one is concerned with determining which tests can be performed in a particular system. The second one is concerned with deciding which particular test to choose (i.e perform) given a list of possible tests and is the aspect of testing this Chapter deals with.

4.2. Algorithmic versus heuristic test optimisation

In heuristic test optimisation methods, the choice of a test is based on a direct correspondence between the current symptoms and the test to be applied. Thus there is no permanent online test optimisation. The testing strategy is precompiled according (usually) to the knowledge and intuition of a human expert. The heuristic optimisation rules are usually of the form: "If symptom A occurs perform test T1". Test optimisation is performed only when the expert decides that test T2 may be better than T1 for symptom A. It then requires explicit deletion of the association between A and T1 and explicit insertion of the association between A and T2.

In algorithmic test optimisation, the choice of the test to be performed is based on an algorithm or method that is explicitly defined. Every time a new symptom appears, the test optimisation algorithm is applied, yielding the best test.

Using algorithmic test optimisation is generally slower than heuristic test optimisation since the optimisation algorithm has to be applied at every stage of the diagnostic procedure. However an algorithmic knowledge test optimisation has several advantages:

- It is a portable optimisation method which can be applied to any system, since it is not making any hidden system dependent assumptions or heuristics. For example, an algorithmic knowledge test optimisation algorithm should be applicable to two completely different computer networks, as long as the symptoms and the tests in both networks are described in a way compatible with the algorithm. On the other hand, a rule of the type "if symptom A appears, perform test T1" which is optimal on one network may be inapplicable to another network (if symptom A does not occur on this other network) and in any case is extremely unlikely to be optimal on the other network.

- It is more adaptable to changes within the system being tested since it is being evaluated at every step. This is particularly relevant in systems where the important parameters such as the probabilities of the components being down tend to vary. In such a system, an algorithmic method which depends on such probabilities will adapt. Heuristic knowledge rules with symptom/test associations will not. For example, let us assume that there is a set of suspect components A,B,C,D,E,F and tests t1(A), t2(A,B,C) and t3(F). A heuristic testing algorithm will simply be: "Given these suspect components, perform test t2". An algorithmic testing method iwhich takes into account the probabilities of failure of the components would request t1 if A had a very high probability of failure, t3 if F had a very high probability of failure and t2 otherwise.

- It is much more suitable when the components of the system being tested may themselves vary. For example new nodes can be added to a computer network. In such a case, data about tests and their coverage (i.e the nodes that they test) will have to be changed. However, an algorithmic test optimisation method will automatically adapt to the new configuration. A heuristic test optimisation method will need to be given explicit instructions as to which tests are optimal when the new nodes become suspect.

- In the long run, it is cheaper to run than heuristic knowledge methodologies since it requires less frequent expert intervention than a heuristic system.

- It can anyway be pre-processed to produce a heuristic knowledge testing sequence if speed is mandatory by explicitly writing down the algorithm's legal input/output pairs in the form of "if .. then" rules.

All these advantages make the usage of an algorithmic approach desirable for test optimisation. However, algorithmic methods may be too time-consuming in complex systems if the time in which they run is dependent on the size of the system.

The optimisation method proposed in this Chapter is based on the entropy measure. This choice was made for several reasons:

- The entropy measure is an algorithmic measure, with all the advantages (described above) that that entails.

- The entropy of a set of elements with known probabilities represents a good measure of the cost of discriminating between these elements [Klee87]. This differentiation between the elements is the ultimate goal of the diagnostic process.

- The entropy measure depends solely on the description of the tests and the current minimal candidates (thus making it independent of the system size). It thus avoids one of the major pitfalls of algorithmic systems.

- It has been used with success in the field of electronic circuit diagnosis [Klee87] , This suggests that it is a suitable diagnostic tool.

4.3. Assumptions

As in the previous Chapter, the following two assumptions are made:

- A test can be either successful, in which case the tested set of components should be working, or unsuccessful in which case at least one of the tested components should be faulty. This is in fact a simplified version of allowing a measurement or test to have many possible outcomes; of all the outcomes one is the correct one, and all the others are incorrect. These other ones are being treated as one group. This simplification is crucial in order to be able to devise a general strategy for test optimisation without a detailed description and knowledge of the failure modes. It is also particularly appropriate in the early stages of the fault management process when the task is simply to locate the fault, (i.e finding exactly where something went wrong) as opposed to diagnose it (i.e finding exactly what went wrong).
- If a test is successful, the components involved in the test can be treated as working components. This is appropriate for two reasons. First in a complex dynamic system, the number of actions performed during one test is likely to be quite substantial and thus the likelihood of normal behaviour being exhibited in the presence of a fault is small. This is in contrast with digital circuits where some inputs may mask a fault. Second, the fault model developed in the previous Chapter allows the possibility of test faults. Thus even if a faulty component is masked by a successful test, it will not be completely removed from the list of suspect components.

Additionally it is assumed that all tests carry equal costs and that they do not affect the system under observation. It is possible to extend the algorithm to take into account tests with differing costs; the cost of the test would then become another parameter.

The test optimisation algorithm developed here does not take into account the possibility of state changes when it is searching for the optimal test to be performed. This is because it is difficult to estimate the probability of a state change occurring, as discussed in section 3.7. It also does not take into account the possibility of sensor faults when estimating the entropy; this simplifies the algorithm and has a negligible effect since it affects the entropy estimate of all the tests and since the probabilities of sensor faults are very small.

Finally, the test optimisation algorithm makes the assumption that faults are independent. The algorithm could be extended to deal with dependent faults but would require a lot of additional data.

4.4. Entropy definition and properties

The concept of entropy was first introduced by Shannon in [Shan49]. The entropy H of a set elements E_1 .. E_n with probabilities p_1 .. p_n where $\sum_{i=1}^{i=n} p_i = 1$ is defined as:

$$H = -\sum_{i=1}^{i=n} p_i \log(p_i)$$

The important properties of the entropy measure as far as test optimisation is concerned are:

- The entropy of a set of probabilities is maximal if all the probabilities are equal. This is equivalent to having a set of minimal candidates all with equal probabilities, which intuitively takes a longer time to diagnose because there is no initial discrimination between the candidates. This also means that there is little information about the system when the entropy is high.

- The contribution of an individual component A to the total sum is p(A)*log(p(A)). When p(A)=1 or p(A)=0, this contribution tends to 0, i.e it does not add anything to the total entropy. Intuitively this is beneficial because p(A)=1 or p(A)=0 means that there is perfect information about A. In fact p(A)=1 gives perfect information about the whole system since the sum of the probabilities of all the other components must then be equal to 0. Hence the smaller the entropy the more information there is about a system.

- Different entropies from different sets can be combined to give an average entropy. This is a useful property because it means that the entropy to which a test will lead if it is succesful can be combined with the entropy to which it will lead if it fails, giving the average entropy for the test.

The calculations in the following section differ from the above-mentioned formal definition of entropy by taking its absolute value. This is computationally equivalent and means that the result is positive and that it has to be maximised rather than minimised.

4.5. Test optimisation using entropy

At the beginning of the test optimisation phase, there is a set of candidate tuples, each of which can explain the symptoms. Each of these minimal candidates has a probability and the methodology for calculating this probability was described in Section 3.4.6.4.

4.5.1. Entropy and test generation

Clearly it is possible to calculate the entropy of the list of minimal candidates since each one of them has a certain probability and the sum of those probabilities is 1. The higher the entropy, the longer it will take on average to find the faulty component and vice versa. Furthermore, since each test will lead to a new list of minimal candidates, the expected entropy to which the test will lead can be calculated by combining the entropy to which the test would lead if it were successful with the entropy to which the test would lead if it failed (entropy property 3 above). The best test will be the one that minimises the expected entropy of the new minimal candidate probabilities resulting from the test (entropy properties 1 and 2 above). The expected entropy of a test t_i is therefore:

$$H_e(t_i) = p(t_i=w)\ H(t_i=w) + p(t_i=f)\ H(t_i=f)$$

where t_i=w means that test t_i finds the tested elements to be working and t_i=f means that test t_i fails.

In other words, the expected entropy to which a test will lead is equal to the probability that the test is successful times the entropy to which this would lead, added to the probability that the test is unsuccessful times the entropy to which this would lead.

4.5.2. Basic notation and equations

Let R_p be the prior (i.e initial) set of minimal candidates. The sum of the probabilities of the elements of R_p is equal to 1.

Let t_i be one of the available tests that tests system components x_{i1} .. x_{ix}.

Then the set R_p can be divided into 2 subsets:

- S_{ip} which is the set of the prior minimal candidates directly tested by the test t_i, i.e candidates necessarily eliminated if t_i succeeds.

- U_{ip} which is the set of prior minimal candidates not directly tested by t_i, i.e candidates remaining if the test succeeds.

The sets R_p,S_{ip} and U_{ip} have the following properties:

$$R_p = S_{ip} \cup U_{ip} \text{ and } S_{ip} \cap U_{ip} = 0$$

Furthermore the components x_{i1} .. x_{ix} of t_i can be divided into 3 disjoint groups:

- A_i are the elements tested by t_i which are also minimal candidates in S_{ip}.

- B_i are the elements tested by t_i which are part of some minimal candidates of S_{ip}.

- C_i are the other components of t_i, i.e the elements which are neither minimal candidates nor part of any minimal candidate in S_{ip}.

For example if $R_p = \{[\overline{bc}:], [\overline{a}:], [\overline{f}:]\}$ and $t_i(a,c,d,e)$, then:

$S_{ip} = \{[\overline{bc}:], [\overline{a}:]\}$
$U_{ip} = \{[\overline{f}:]\}$
$A_i = \{a\}$
$B_i = \{c\}$
$C_i = \{d,e\}$

Another consequence of the definition $R_p = S_{ip} \cup U_{ip}$ is that:

$$H(R_{ip}) = H(S_{ip}) + H(U_{ip})$$

$$= \sum_{i=1}^{i=s} p(x_{is}) \log p(x_{is}) + \sum_{i=1}^{i=u} p(x_{iu}) \log p(x_{iu})$$

$$= H1 + H2$$

where

$$(1)\ H1 = \sum_{i=1}^{i=s} p(x_{is}) \log p(x_{is})$$

$$(2)\ H2 = \sum_{i=1}^{i=u} p(x_{iu}) \log p(x_{iu}).$$

The reason for splitting the set of minimal candidates R_p and the set of components tested by test t_i in this way is that we will be estimating the effects of the failure of t_i and of the success of t_i on the list of minimal candidates R_p. As will become clear in the following sections, each of these subsets behaves differently during the calculation of the effects of t_i on R_p.

Furthermore, if the sets S_{ip}, U_{ip}, A_i, B_i and C_i are defined as follows:

$S_{ip} = \{x_{s1} .. x_{ss}\}$

$U_{ip} = \{x_{u1} .. x_{uu}\}$

$A_i = \{x_{a1} .. x_{aa}\}$

$B_i = \{x_{b1} .. x_{bb}\}$

$C_i = \{x_{c1} .. x_{cc}\}$

then the following equations follow:

(3) $p(S_{ip}=f) = \sum_{i=1}^{i=s} p(x_{si}=f)$

(4) $p(U_{ip}=f) = \sum_{i=1}^{i=u} p(x_{ui}=f)$

(5) $p(S_{ip}=f) + p(U_{ip}=f) = 1$

(6) $p(A_i=f) = \sum_{i=1}^{i=a} p(x_{ai}=f)$

(7) $p(A_i=w) = 1 - p(A_i=f)$

(8) $p(B_i=f) = \sum_{i=1}^{i=b} p(x_{bi}=f)$

(9) $p(B_i=w) = 1 - p(B_i=f)$

(10) $p(C_i=w) = \prod_{i=1}^{i=c} p(x_{ci}=w)$

(11) $p(C_i=f) = 1 - p(C_i=w)$

(12) $p(A_i \text{ or } B_i=f) = p(A_i=f) + p(B_i=f) = p(S_{ip}=f)$

(13) $p(A_i \text{ and } B_i=w) = 1 - p(A_i \text{ or } B_i=f) = p(U_{ip}=f)$

(14) $p(t_i=w) = p(A_i \text{ and } B_i=w)\ p(C_i=w) = p(U_{ip}=w)\ p(C_i=w)$

(15) $p(t_i=f) = 1 - p(U_{ip}=S)\ p(C_i=w)$

Equations (3),(4),(5),(6),(8) and (12) are true because the probabilities of failure of those candidates have been normalised to 1.

Equation (10) is true because the probability of having several elements working is equal to the product of each element's probability to work.

Equation (13) is true because it represents the probability that there is no fault in A_i,B_i which is complementary to the probability that there is a fault in A_i,B_i (equation (10)).

Equation (14) is true because the probability of two sets working is equal to the product of each set's probability to work.

Equations (7),(9),(11) and (15) are true because for events with two outcomes, the sum of the probabilities of the two outcomes must be equal to 1.

These equations will be used in the evaluation of a test's expected entropy, which will be discussed in the following sections.

4.5.3. Candidate probabilities of a failed test

This section looks at the new minimal candidates a test would generate if it fails and at the probabilities of these candidates.

The result of a failed test is a set of minimal candidates R_f that can be divided into three parts S_{if},X_{if} and Y_{if} where:

$$R_f = S_{if} \cup X_{if} \cup Y_{if}$$

These three parts of R_f are obtained as follows:

$$S_{if} = S_{ip}$$
$$X_{if} = \{ [\overline{x_{bi}}, \overline{x_{ui}}:] \}$$
$$Y_{if} = \{ [\overline{x_{ci}}, \overline{x_{ui}}:] \}$$

This is a result of the application of the minimal candidate generation algorithm described in section 3.6.3 and the way the subsets of R_p and t_i were defined in section 4.5.2:

a) The combination of the minimal candidates from A_i, B_i on one side and S_{ip} on the other side recreates S_{ip}.

b) The combination of the candidates from B_i and U_{ip} is simply performed by adding the elements of the candidates together, since there is no intersection between them. This yields X_{if}.

c) The combination of the candidates from C_i and U_{ip} is again simply performed by adding the elements of the candidates together, since there is no intersection between them. This yields X_{if}.

d) The candidates obtained from the combination of C_i and S_{ip} are subsumed by the candidates from a).

e) The candidates obtained from the combination of A_i and U_{ip} are also subsumed by the candidates from a).

To take a previously mentioned example, let: $R_p = \{[\overline{bc}:], [\overline{a}:], [\overline{f}:]\}$ and $t_i(a,c,d,e)$.

Then, as seen:

S_{ip} = {[$\overline{bc}$:], [$\overline{a}$:]}

U_{ip} = {[$\overline{f}$:]}

A_i = {a}

B_i = {c}

C_i = {d,e}

When t_i fails, it yields 4 minimal candidates:[$\overline{a}$:],[$\overline{c}$:],[$\overline{d}$:] and [$\overline{e}$:].

a) Combining [$\overline{a}$:],[$\overline{c}$:] and [$\overline{bc}$:],[$\overline{a}$:] yields: [$\overline{a}$:] and [$\overline{bc}$:] (the other two candidates created are subsumed by those two).

b) Combining [$\overline{c}$:] and [$\overline{f}$:] gives [$\overline{cf}$:].

c) Combining [$\overline{d}$:],[$\overline{e}$:] with [$\overline{f}$:] gives [$\overline{df}$:] and [$\overline{ef}$:].

d) Combining [$\overline{d}$:],[$\overline{e}$:] with [$\overline{bc}$:],[$\overline{a}$:] will give tuples which are subsumed by those from a).

e) Combining [$\overline{a}$:] with [$\overline{f}$:] gives [$\overline{af}$:] which is again subsumed by [$\overline{a}$:] from a).

Thus the final combination result of [$\overline{bc}$:], [$\overline{a}$:], [$\overline{f}$:]} and [$\overline{a}$:], [$\overline{c}$:], [$\overline{d}$:] and [$\overline{e}$:] is:
[$\overline{a}$:], [$\overline{bc}$:], [$\overline{cf}$:], [$\overline{df}$:] and [$\overline{ef}$:].

At each step of the diagnosis, the new candidate probabilities are calculated using the existing candidate probabilities and Bayes rule. Bayes rule states that for a candidate C:

$$p(C/t_i=f) = \frac{p(t_i=f/C)\, p(C)}{p(t_i=f)}$$

In this equation:

- $p(C/t_i=f)$ is the probability of candidate C given that the test has failed, i.e the posterior probability of C.
- $p(C)$ is the prior probability of candidate C.
- $p(t_i=f/C)$ is the (conditional) probability that the test t_i has failed given candidate C.

For all the new candidates, the term $p(t_i=f/C)$ is equal to 1. This is because every

minimal candidate will have to contain some element tested by t_i. This term can therefore be ignored.

Let p'(C) be the new probability of a candidate, i.e $p(C/t_i=f)$ and p(C) be the old one.

Then for any member of S_{if} , the new probability is:

$$p'(x_{is}) = \frac{p(x_{is})}{p(t_i=f)}$$

For any member $[\overline{x_{bi}}, \overline{x_{ui}}:]$ of X_{if} :

$$p'([\overline{x_{bi}}, \overline{x_{ui}}:]) = \frac{p(x_{bi})\, p(x_{ui})}{p(t_i=f)}$$

For any member $[\overline{x_{ci}}, \overline{x_{ui}}:]$ of Y_{if} :

$$p'([\overline{x_{ci}}, \overline{x_{ui}}:]) = \frac{p(x_{ci})\, p(x_{ui})}{p(t_i=f)}$$

This is justified in Appendix 1.

4.5.4. Entropy of a failed test

The entropy of the failed test is the sum of the entropies of S_{if}, X_{if} and Y_{if}.

S_{if} is made of s elements x_{is}. Therefore its entropy is:

$$H(S_{if}) = \sum_{i=1}^{i=s} \frac{p(x_{is})}{p(t_i=f)} \log \frac{p(x_{is})}{p(t_i=f)}$$

$$= \frac{1}{p(t_i=f)} \left[H1 - \log(p(t_i=f)(1-p(U_{ip}=f)) \right]$$

using equations (1),(3),(4) and (5).

X_{if} is made of b groups of candidate sets (b being the size of B_i), and each group is

obtained by joining one element of B_i with all the elements of U_{ip}. Therefore its new entropy is:

$$H(X_{if}) = \sum_{i=1}^{i=u} \frac{p(x_{b1})\,p(x_{ui})}{p(t_i=f)} \log \frac{p(x_{b1})\,p(x_{ui})}{p(t_i=f)} + .. + \sum_{i=1}^{i=u} \frac{p(x_{bb})\,p(x_{ui})}{p(t_i=f)} \log \frac{p(x_{bb})\,p(x_{ui})}{p(t_i=f)}$$

$$= \frac{1}{p(t_i=f)} \left\{ p(x_{b1}) \left[\sum_{i=1}^{i=u} p(x_{ui}) \log p(x_{ui}) + \sum_{i=1}^{i=u} p(x_{ui}) \log \frac{p(x_{b1})}{p(t_i=f)} \right] + .. \right.$$

$$\left. + p(x_{bb}) \left[\sum_{i=1}^{i=u} p(x_{ui}) \log p(x_{ui}) + \sum_{i=1}^{i=u} p(x_{ui}) \log \frac{p(x_{bb})}{p(t_i=f)} \right] \right\}$$

$$= \frac{1}{p(t_i=f)} \left\{ \sum_{i=1}^{i=b} p(x_{bi})\, H2 \right.$$

$$+ \left[\sum_{i=1}^{i=u} p(x_{ui})\,p(x_{b1}) \log p(x_{b1}) + .. + \sum_{i=1}^{i=u} p(x_{ui})\,p(x_{bi}) \log p(x_{bi}) \right]$$

$$\left. - \left[\sum_{i=1}^{i=u} p(x_{ui})\,p(x_{b1}) \log p(t_i=f) + .. + \sum_{i=1}^{i=u} p(x_{ui})\,p(x_{bb}) \log p(t_i=f) \right] \right\}$$

$$= \frac{1}{p(t_i=f)} \left\{ p(B_i=f)\, H2 + p(U_{ip}=f) \sum_{i=1}^{i=b} p(x_{bi}) \log p(x_{bi}) \right.$$

$$\left. - p(U_{ip}=f) \log \left(p(t_i=f)\,p(B_i=f)\right) \right\}$$

using equations (2),(4) and (8).

Similarly for Y_{if} the entropy is:

$$H(Y_{if}) = \frac{1}{p(t_i=f)} \left\{ p(C_i=f)\, H2 + p(U_{ip}=f) \sum_{i=1}^{i=c} p(x_{ci}) \log p(x_{ci}) \right.$$

$$\left. - p(U_{ip}=f) \log \left(p(t_i=f)\,p(C_i=f)\right) \right\}$$

4.5.5. Entropy of a successful test

The result of a successful test is the set U_{if} of minimal candidates that is equal to U_{ip}. This is because A_i and B_i eliminate all the minimal candidates contained in S_{ip}.

The entropy of a successful test can be approximated to:

$$H(t_i{=}w) = p(C_i{=}w)\, p(U_{ip}{=}f) \left[H2 + p(U_{ip}) \log \left(p(C_i{=}w)\, p(U_{ip}{=}f)\right)\right]$$

This is justified in Appendix 1.

4.5.6. Total expected entropy of a test

As seen before, the expected entropy of a test is:

$$H_e(t_i) = p(t_i{=}f)\, H(t_i{=}f) + p(t_i{=}w)\, H(t_i{=}s)$$

$$H_e(t_i) = p(t_i{=}f) \left[H(S_{if}) + H(X_{if}) + H(Y_{if})\right] + p(t_i{=}w)\, H(t_i{=}w)$$

$$= \left[H1 - \log(p(t_i{=}f)\,(1{-}p(U_{ip}{=}f))\right]$$

$$+ \left[p(B_i{=}f)\, H2 + p(U_{ip}{=}f) \sum_{i=1}^{i=b} p(x_{bi}) \log p(x_{bi}) - p(U_{ip}{=}f) \log \left(p(t_i{=}f)\, p(B_i{=}f)\right)\right]$$

$$+ \left[p(C_i{=}f)\, H2 + p(U_{ip}{=}f) \sum_{i=1}^{i=c} p(x_{ci}) \log p(x_{ci}) - p(U_{ip}{=}f) \log \left(p(t_i{=}f)\, p(C_i{=}f)\right)\right]$$

$$+ p(t_i{=}w)\, p(C_i{=}w)\, p(U_{ip}{=}f) \left[H2 + p(U_{ip}) \log \left(p(C_i{=}w)\, p(U_{ip}{=}f)\right)\right]$$

$$= H(R_{ip}) + \left[p(B_i{=}f) + p(C_i{=}f) + \left(p(C_i{=}w)\, p(U_{ip}{=}f)\right)^2 - 1\right] H2$$

$$- \left[1 - p(U_{ip}{=}f) + p(U_{ip}{=}f)\, p(B_i{=}f) + p(U_{ip}{=}f)\, p(C_i{=}f)\right] \log p(t_i{=}f)$$

$$+ \left[\sum_{i=1}^{i=b} p(x_{bi}) \log p(x_{bi}) + \sum_{i=1}^{i=d} p(x_{ci}) \log p(x_{ci}) \right.$$

$$\left. + \left(p(C_i{=}w)\, p(U_{ip}{=}f)\right)^2 \log \left(p(C_i{=}w)\, p(U_{ip}{=}f)\right)\right] p(U_{ip})$$

Since the final expression is:

$$H_e(t_i) = H(R_{ip}) + \Delta H$$

the effect of test t_i on the list of minimal candidates can be calculated without explicitly calculating its entropy; the best test is the one that *maximises* ΔH (since the absolute

value of the entropy is taken).

So the best test is the one that maximises:

$$\Big[\, p(B_i{=}f) + p(C_i{=}f) + \Big(p(C_i{=}w)\, p(U_{ip}{=}f)\Big)^2 - 1\Big] \sum_{i=1}^{i=u} p(x_{iu}) \log p(x_{iu})$$

$$-\Big[\, 1 - p(U_{ip}{=}f) + p(U_{ip}{=}f)\, p(B_i{=}f) + p(U_{ip}{=}f)\, p(C_i{=}f)\Big] \log p(t_i{=}f)$$

$$+\Bigg[\sum_{i=1}^{i=b} p(x_{bi}) \log p(x_{bi}) + \sum_{i=1}^{i=c} p(x_{ci}) \log p(x_{ci})$$

$$+\Big(p(C_i{=}w)\, p(U_{ip}{=}f)\Big)^2 \log(p(C_i{=}w)\, p(U_{ip}{=}f))\Big]\, p(U_{ip})$$

4.6. Terminating a diagnostic session by test biasing

As mentioned in section 3.7.4.4, it is beneficial to end the diagnostic session with a test that tests some of the components included in the most probable minimal candidate (and hopefully finding that the test is unsuccessful). The current formula for calculating the expected entropy of a test will not necessarily yield such a test because it does not have any weighting to differentiate between a successful test and an unsuccessful test (beyond the probability of failure or success of the test). To show this problem in a simple example, let us imagine that we have three minimal candidates [$\overline{a}$:], [$\overline{b}$:] and [$\overline{c}$:] with respective normalised probabilities of failure 0.8, 0.1 and 0.1 and two tests t1(a) and t2(b,c). If t1 fails, which has a probability of 0.8, the list of minimal candidates becomes [$\overline{a}$:] with probability 1 and the entropy is thus 1*log(1)=0. If t1 succeeds, which has a probability of 0.2, the minimal candidates that remain are [$\overline{b}$:] and [$\overline{c}$:] with equal probability 0.5 and its entropy will be 0.5*log(0.5) + 0.5*log(0.5) = log(0.5). Therefore its expected entropy is 0.2 * log (0.5) (We have again taken the absolute value of the entropy). Similarly for t2 the expected entropy is 0.8*0 + 0.2*log(0.5).

t1 and t2 thus have the same expected entropy. However section 3.7.4.4 concluded that t1 should be preferred over t2 since t1 would be confirming the symptom (if it fails).

To get around this problem, the test selection can be biased towards the tests that are more likely to fail by only considering the entropy of the failed test and multiplying it with the probability of the test being unsuccessful. Then the probability of the test being unsuccessful becomes a weight in favour of the tests that are more likely to fail. In the example above, t1 would be the chosen test.

The necessary information for implementing this algorithm is available. In section 4.5.6, the total expected entropy of a test was calculated. By eliminating from the final formula all the elements that were contributed by the expected entropy of the successful test, what is left is the expected entropy of the failed test, which is the required expression.

4.7. Heuristics

There are several ways of reducing the computational cost of this method:

- It is possible to choose the first test which increases the entropy difference measure by more than a certain limit. A limit of 1 - log 0.5 = 0.7 would be appropriate as such a measurement would on average split the candidate space in half [Klee87].
- The generation of new tests can be stopped when a candidate probability reaches some threshold. The choice of this threshold will depend on how dangerous misdiagnosis is.
- The generation of new tests can be stopped when no test can reduce the entropy by more than a certain amount. This will normally mean that there are no tests that can discriminate between the most likely candidates.
- The calculation of the entropy difference measure can be based solely on the most probable candidates. This would in principle simplify the evaluation of each test without affecting the test orderings, since the candidates with small probabilities have very little effect on the overall entropy. The performance effects of this approach are analysed in Chapter 6.

4.8. Examples

In an intuitive way, the best tests are the ones that will on average halve the space of minimal candidates. The expected number of measurements is then logarithmically proportional to the size of the minimal candidate space.

Consider a system made of 6 elements A,B,C,D,E and F and let us suppose that the available tests are:

t0(A,B,C,D), t1(A,B), t2(A,B,E,F), t3(A), t4(A,E,F), t5(A,B,C), t6(A,B,C,E,F)

Example 1

Let us assume that the initial information is that one of the elements A,B,C,D is faulty, and that the initial probabilities of A,B,C,D,E,F being down are all equal to 0.01. This therefore leads to the following minimal candidates and minimal candidate probabilities:

Suspects $[\overline{A}:]$ $[\overline{B}:]$ $[\overline{C}:]$ $[\overline{D}:]$

$p(\overline{A})=0.25$, $p(\overline{B})=0.25$, $p(\overline{C})=0.25$, $p(\overline{D})=0.25$, $p(\overline{E})=0.01$, $p(\overline{F})=0.01$

The expected information gain from each test is summarised in Table 4.1.

Suspects $[\overline{A}:],[\overline{B}:],[\overline{C}:],[\overline{D}:]$ **Probabilities** $p(\overline{A})=0.25$ $p(\overline{B})=0.25$ $p(\overline{C})=0.25$ $p(\overline{D})=0.25$ $p(\overline{E})=0.01$ $p(\overline{F})=0.01$	
Test	Entropy change
t0(A,B,C,D)	0
t1(A,B)	0.78
t2(A,B,E,F)	0.72
t3(A)	0.68
t4(A,E,F)	0.61
t5(A,B,C)	0.52
t6(A,B,C,E,F)	0.49

TABLE 4.1; Entropy example 1

As expected, the best test is t1 which divides the set of minimal candidates into two and does not bring in any new elements. t0 is not bringing in any information and the gain is therefore 0.

Example 2

Let us assume that the initial probability of A being down is 0.02 and the others are still 0.01. Then the minimal candidates and the minimal candidate probabilities become:

Suspects $[\overline{A}:]$ $[\overline{B}:]$ $[\overline{C}:]$ $[\overline{D}:]$

$p(\overline{A})=0.4$, $p(\overline{B})=0.2$, $p(\overline{C})=0.2$, $p(\overline{D})=0.2$, $p(\overline{E})=0.01$, $p(\overline{F})=0.01$

The expected information gain from each test is summarised in Table 4.2.

Suspects $[\overline{A}:],[\overline{B}:],[\overline{C}:],[\overline{D}:]$ **Probabilities** $p(\overline{A})=0.4$ $p(\overline{B})=0.2$ $p(\overline{C})=0.2$ $p(\overline{D})=0.2$ $p(\overline{E})=0.01$ $p(\overline{F})=0.01$	
Test	Entropy change
t0(A,B,C,D)	0
t1(A,B)	0.79
t2(A,B,E,F)	0.74
t3(A)	0.87
t4(A,E,F)	0.81
t5(A,B,C)	0.48
t6(A,B,C,E,F)	0.45

TABLE 4.2; Entropy example 2

The best test now is t3 which is intuitively correct because it splits the minimal candidates into two sets: one containing the higher probability candidate ($[\overline{A}:]$) and the other containing the rest of the candidates.

Example 3

Let us assume that the initial probability of B being down is 0.02 and the others are still 0.01. Then the minimal candidates and the minimal candidate probabilities become:

Suspects $[\overline{A}:]$ $[\overline{B}:]$ $[\overline{C}:]$ $[\overline{D}:]$

$p(\overline{A})=0.2$, $p(\overline{B})=0.4$, $p(\overline{C})=0.2$, $p(\overline{D})=0.2$, $p(\overline{E})=0.01$, $p(\overline{F})=0.01$

The expected information gain from each test is summarised in Table 4.3.

Suspects $[\overline{A}:],[\overline{B}:],[\overline{C}:],[\overline{D}:]$ **Probabilities** $p(\overline{A})=0.2$ $p(\overline{B})=0.4$ $p(\overline{C})=0.2$ $p(\overline{D})=0.2$ $p(\overline{E})=0.01$ $p(\overline{F})=0.01$	
Test	Entropy change
t0(A,B,C,D)	0
t1(A,B)	0.79
t2(A,B,E,F)	0.74
t3(A)	0.57
t4(A,E,F)	0.50
t5(A,B,C)	0.48
t6(A,B,C,E,F)	0.45

TABLE 4.3; Entropy example 3

Test t3 is not as good this time because the high probability candidate is not $[\overline{A}:]$ any more. Test t1 is the best test again. Incidentally the amount of information gained by t1 is the same as in the previous example. This is because t1 does not distinguish between candidates $[\overline{A}:]$ and $[\overline{B}:]$.

Elsewhere, it is interesting to notice that in all the examples, any test which tests a subset S of the list of minimal candidates as well as some other components scores less well

than a test which tests only the subset S (such as t1 and t2, t3 and t4, t5 and t6) which is one of the properties we wanted.

4.9. Discussion

The formula to be calculated looks rather complex. However, any deep knowledge method that tries to optimise the testing algorithm will have to scan the tests, the list of minimal candidates, compare them and at least find their intersection. All the parameters needed to estimate the entropy are easily calculated during this scanning period, by simply adding the (known) probabilities of the elements involved in the tests and in the list of minimal candidates. Therefore the algorithm is at most of the same order as other deep knowledge methods. The major advantages of the method described above are:

1) it depends only on the prior probabilities of the components being down (obtained by observation) and the list of available tests;
2) it optimises the tests for a multiple fault environment; no simplifying single fault assumption is made;
3) the representation of the list of suspect elements by the set of minimal candidates is particularly succinct and time-saving;
4) it is a deep knowledge method, system independent and flexible because it is applied at every testing stage. As such it is particularly suitable in dynamic environments where the probabilities of components being down vary frequently;
5) it does not require any prior knowledge of the failure modes, only a distinction between failed tests and successful tests.

Its main disadvantages are:

1) it is more costly than a shallow knowledge testing mechanism. This can to a certain extent be remedied by accepting the first test that reduces the initial entropy by more than a certain amount;
2) the assumption that the probability of a minimal candidate is equal to the sum of the probabilities of all the candidates implicitly represented by that minimal candidate (used in Appendix 1) is at best an approximation. This can easily be shown using an example.

Let us say there are two elements A and B with $p(\overline{A})=0.1$ and $p(\overline{B})=0.1$ Then initially the system has 4 possible states:

AB with probability 0.81, $\overline{A}B$ with probability 0.09, $A\overline{B}$ with probability 0.09, and $\overline{AB}$ with probability 0.01.

If a test establishes that (A,B) is not working, the new probabilities are approximately (using Bayes' rule) :-

AB with probability 0, $\overline{A}B$ with probability 0.47, $A\overline{B}$ with probability 0.47, and $\overline{AB}$ with probability 0.06.

The probability of all the minimal candidates that contain element A is 0.53 (and similarly for B). The method for calculating the probabilities of the minimal candidates would assign 0.5 to minimal candidate $[\overline{A}:]$ and 0.5 to minimal candidate $[\overline{B}:]$ (i.e the probability of both A and B being faulty is distributed between the two minimal candidates). It is of course possible to calculate the cross-probabilities separately, but that is computationally prohibitive for a system of reasonably large size. However if the probabilities of the components being down are sufficiently small, the error introduced with this estimate becomes negligible. The probability chosen for A and B (0.1) in this example is a very high one. It means that on average the component tested will completely fail one out of ten times. Such a fault rate would not be tolerable in most dynamic systems and thus such a case will not normally occur.

As far as the performance of this testing algorithm is concerned, its main property is that it is not dependent on the number of components in the system under observation; its main parameters are the number of tests available and the number of current minimal candidates. The testing algorithm performance is discussed in more detail in the next Chapter.

4.10. Review of some existing test optimisation mechanisms

4.10.1. Arby's testing methodology

Arby [Will83] is an expert system to isolate faults in a communication network. The authors state that the choice of tests should depend on the tests available, the information content of the tests and their respective costs but do not actually mention how or whether

they have achieved that.

4.10.2. Sophie's ad-hoc half split method

In Sophie [Brow82], the measurement strategy chooses the test that comes closest to splitting the list of suspects into two lists of similar probability. For example, given elements A,B,C,D and tests t1(A), t2(A,B) and t3(A,B,C), it will choose t2 if all 4 elements have similar probability of failure, t1 if A has a higher probability of failure than the others and t3 if D has a higher probability of failure than the others. Sophie's methodology corresponds to a binary search of the suspects in the probabilistic space and is a good method if the set of suspects does not change during the diagnosis. This is true under the single fault assumption, which allows the intersection of new suspects with existing suspects.

On the other hand, if multiple faults can occur, the probabilistic binary search is not optimal. For example, let us assume that there are two suspects [$\overline{A}$:] and [$\overline{B}$:] and two tests t1(A) and t2(A,C,D,E,F) and that there are no sensor faults and state changes

If t1 is chosen, and it is successful, then B is faulty. If it fails, then A is faulty.

If t2 is chosen, and it succeeds, B is faulty. If on the other hand, t2 fails, the new suspects become: [$\overline{A}$:], [$\overline{BC}$:], [$\overline{BD}$:], [$\overline{BE}$:], [$\overline{BF}$:].

Clearly, t1 is to be preferred to t2. However the probabilistic binary search does not distinguish between the two tests and therefore does not necessarily choose the test that converges fastest towards the solution.

4.10.3. Shirley's testing method

Shirley's test optimisation mechanism [Shir83] is based on the selection of tests which combine components known to be working with a few suspect components, so as to discriminate between the suspect components. This assumes that there will always be a test that can combine both components known to be working and suspect components (which is a system dependent property). Furthermore, in dynamic systems, there is never a 100% certainty that a component is working since a state change can always occur. Finally, no real test optimisation methods are suggested, just the above metric for choosing a reasonable test.

4.10.4. DART

DART [Gene84] is a deep knowledge based system which describes the design structure (components) of the system and the design function(behaviour) of the system in terms of propositional logic. When a symptom appears, DART tries to obtain the set of suspect components by logical deduction, that is a logical proposition of the form (NOT A) OR (NOT B) OR .. where A B .. are system components. Similarly for testing purposes, DART attempts to obtain a logical proposition which combines propositions containing one or more suspects with a proposition containing the definition of a test.

This methodology is again aimed at the test generation part of the testing process; it is not a test optimisation methodology. There is no attempt to evaluate and rank the different tests available, just to generate a reasonable test.

4.10.5. General Diagnostic Engine (GDE)

In [Klee87], a general fault solving mechanism is described, incorporating a sequential diagnosis mechanism which proposes measurements based on the application of entropy. The main measurement characteristics of the GDE system are:

- that each measurement (test) is a single point test;
- with each measurement is associated the set of environments in which the measurement is valid;
- a measurement can take any of a number of values;
- a successful measurement does not imply that the component(s) is working. It could exhibit faulty behaviour on another set of inputs. Therefore all reasoning is based on faulty behaviour.

The way in which GDE uses the entropy measure is tailored to the task of troubleshooting digital circuits. In particular, this implies single measurement points, many measurement values and the discarding of successful tests. Our methodology is more suited to complex dynamic systems.

4.11. Summary

In this Chapter, a new mechanism for test selection optimisation in complex dynamic systems was developed. The mechanism uses a deep knowledge algorithm to optimise its test selection and is based on the concept of entropy. The advantage of such a deep knowledge mechanism is its adaptability to any dynamic system and its ability to

perform optimisation at every step. The usage of the entropy measure also provides a methodology whose execution time does not depend directly on the size of the tested system since it requires solely the probabilities of failure of the components and the test characteristics. The Chapter also examined other test optimisation techniques and found them inapplicable to the problem at hand.

CHAPTER V

MAPPING THE FAULT DIAGNOSIS AND TEST GENERATION ALGORITHMS TO THE BLACKBOARD/ATMS ARCHITECTURE

5.1. Introduction

The previous three Chapters presented several elements of a fault diagnostic environment for complex dynamic systems, namely a high level ATMS/blackboard architecture, a general fault diagnostic model and an algorithmic method for optimising the selection of tests. This Chapter describes how these separate elements of the diagnostic environment can be mapped to each other and combined to provide a shell for fault diagnosis in complex dynamic systems. The fault diagnostic process is analysed both from the ATMS's perspective and from the blackboard's perspective and the cooperation between the ATMS and the blackboard is examined.

5.2. Fault diagnosis: the ATMS perspective

5.2.1. ATMS reminder

The general principles of the ATMS were already described in Chapter 2. Here only a brief reminder of the main characteristics of the ATMS is given.

The ATMS stores each datum received from the problem solver in an ATMS node. The format of the node is the following:

node(NodeId,Level,Datum,Label,Antecedents,Consequents,Informant,Status)

- **NodeId** is a unique identifier;
- **Level** specifies the blackboard level to which the node belongs;
- **Datum** contains the problem solver datum;

- **Label** contains the minimal environments in which the datum is valid;

- **Antecedent** lists the nodes from which the current node was deduced;

- **Consequent** lists the nodes in which the current node is an antecedent;

- **Informant** describes the action to be performed if the current node is a consumer node (as described in Section 2.5.4);

- **Status** describes the datum's and hence the node's status; this status can be true (always true), false, in (currently valid in at least one environment) or out (not currently valid in any environment but not necessarily false).

The inputs the ATMS accepts from the problem solver are as follows:

- a new datum that is a premise (a universally true datum);

- a new datum that is an assumption;

- a new datum which is justified by data from some existing nodes;

- the rejustification of an existing datum by a set of nodes;

- the incompatibility (logical contradiction) of a set of one or more nodes (i.e of the data contained in those nodes).

The list of inconsistent environments is kept in the 'nogood' database.

The ATMS is given assumptions and justifications, and its role is to determine the contexts in which the data are valid.

5.2.2. Mapping the diagnostic inference procedure onto the ATMS

Since the ATMS's world consists of premises, assumptions, environments, justifications and contradictions, any problem solver that wants to use the ATMS has to be able to describe its problem in those terms, that is to use a problem-solving procedure which reasons using those concepts.

The inference procedure of the diagnostic problem solver, which has to be mapped to the ATMS has several stages:

- test results are received;

- they are translated into the implicated components;

- they are filtered to eliminate known errors;

- they are combined with all the existing findings generating new minimal suspects;
- new tests are requested to discriminate further between the suspects.

The mapping will have two types of assumptions. The assumptions of the first type will relate to the fact that components are working or not working. Therefore for each component a, there are two possible assumptions of the first type: either that the component a is working (assumption {a}) or that the component a is not working (assumption $\{\bar{a}\}$). The environments containing both these assumptions are declared as 'nogoods' since an element cannot be working and not working at the same time (i.e nogood(a,$\bar{a}$)).

The assumptions of the second type will relate to the temporal value of a test datum (i.e the datum will be valid if the temporal assumption still holds). This second type of assumptions allows the temporal reasoning about state changes, as will be shown later.

Given those assumptions, the way in which the different stages of fault diagnosis can be mapped to the ATMS is described in the following paragraphs.

Test results (symptoms) received from the system can be justified with a temporal assumption T_i. The meaning of this temporal assumption to the problem solver is that the test result is valid if there has been no state change (i.e while assumption T_i holds). If the problem solver suspects a state change and finds that a repeated test has changed its findings, the initial test findings can be invalidated by contradicting this assumption.

Translated results of failed tests can be justified with all the combinations of pairs formed by the failed test result and the assumption that one of the implied components failed. For example, the translation of t_1<a,b> can be stored in 2 nodes: one containing the assumption that t_1 is faulty and the other the candidates $[\bar{a}:]$ and $[\bar{b}:]$. The second one would be justified by the nodes containing the assumptions $\{\bar{a}\}$ and $\{\bar{b}\}$ as depicted below: (the node fields shown are NodeId, Datum, Label, Antecedents, Consequents)

1:$[\bar{t_1}:]$,{{$\bar{t_1}$}},{{1}},{..}

2:$[\bar{a}:]$,{{$\bar{a}$}},{{2}},{6}

3:$[\bar{b}:]$,{{$\bar{b}$}},{{3}},{6}

4:T1,{T1},{{4}},{5}

5:t_1<a,b>,{{T1}},{{4}},{6}

6:$[\bar{a}:]$,$[\bar{b}:]$,{{$\bar{a}$T1},{$\bar{b}$T1}},{{2,4,5},{3,4,5}},{..}

Translated results of successful tests can be justified with the combination of the assumptions that all the involved components work and the premise that the test was successful. For example, the test result t_1/a,b/ would be stored as below:

1:[$\overline{t_1}$:],{{$\overline{t_1}$}},{{1}},{..}

2:[:a],{{a}},{{2}},{5}

3:[:b],{{b}},{{3}},{5}

4:T1,{T1},{{4}},{5}

5:t_1/a,b/,{{T1}},{{4}},{6}

6:[:ab],{{abT1}},{{2,3,4,5}},{..}

The filtering effect is achieved by declaring system components that are known to be down as premises that the components are down. Then if a test finds that a set of components, which includes a component known to be down, is not working, that test finding is justified with the premise that the component is not working, thus stopping any further work on that data.

Final diagnosis or further test requests are justified by all the currently believed data.

The label update/minimal candidate generation duality

The above discussion of the mapping of the diagnostic problem to the ATMS has covered all the stages of the diagnostic problem except the combination of the new and existing minimal candidates. This section will show that given the way the assumptions have been defined and given the way test results have been translated and justified, the label update algorithm in fact performs the minimal candidate generation.

The label update and minimal candidate generation algorithms were given in Sections 2.5.4 and 3.6.3 respectively. They are summarised in Table 5.1.

In the input phase, given the way that successful and unsuccessful test results have been translated, every candidate tuple will have a corresponding environment in the label, whose assumptions are the mirror image of the tuple's characteristics. Thus, in the example from the translation of failed test results, the translated test node was:

6:[$\overline{a}$:],[$\overline{b}$:],{{$\overline{a}$T1},{$\overline{b}$T1}},{{2,4,5},{3,4,5}},{..}

where (ignoring T1 for the moment), the 2 label environments respectively mirror the two minimal candidate tuples. Similarly, in the translation of the successful test, the

Algorithm phase		Label update	Minimal candidate algorithm
1	**Input**	Antecedent nodes	Translated test results
2	**Correlation of elements**	Union of all the combinations of picking one environment from each antecedent node label	Every new candidate tuple correlated with every existing candidate tuple by first doing a union of the non working components and then a union of the working components
3	**Elimination of inconsistent elements**	Inconsistent elements are environments which are supersets of nogood environments	Inconsistent elements are tuples which contain the same system component in both the working and non-working part of the tuple
4	**Elimination of non-minimal candidates**	Eliminate subsumed and duplicated environments	Eliminate subsumed and duplicated tuples

TABLE 5.1; Label update/Minimal candidate algorithm duality

result was:

6:[:ab],{{abT1}},{{2,3,4,5}},{..}

where (ignoring T1 again), the label mirrors the minimal candidate tuple.

In the element combination phase, both algorithms do the combination by performing a union operation on all the possible pairs of elements.

In phase 3, the minimal candidates algorithm eliminates all the tuples which contain the same element in both the working and non-working part of the tuple. The label update algorithm eliminates supersets of 'nogood' environments. However, since all the pairs of

assumptions (a,$\overline{a}$) are declared as 'nogood', all the mirror images of tuples which are eliminated are 'nogood' and thus eliminated from the label.

Finally, the same operation is performed on the tuples and the label during phase 4.

For example, let us assume that the test results are t_1<a,b,c> and t_2/a,b/.

After t_1, there are two nodes:

1: [$\overline{t_1}$:], {{$\overline{t_1}$}}

2: [$\overline{a}$:],[$\overline{b}$:],[$\overline{c}$:], {{$\overline{a}$T1},{$\overline{b}$T1},{$\overline{c}$T1}}

Similarly, t_2 is translated into:

3: [:ab], {{abT2}}

4: [$\overline{t_2}$:], {{$\overline{t_2}$}}

Combining 3 and 4 with 1 and 2 and applying the minimal candidate generation algorithm on the datum and the label update algorithm on the label gives:

from 1 and 3:

5: [$\overline{t_1}$:ab], {{$\overline{t_1}$abT2}}

from 1 and 4:

6: [$\overline{t_1 t_2}$:], {{$\overline{t_1 t_2}$}}

from 2 and 3:

7: [$\overline{c}$:ab], {{ $\overline{c}$abT1T2}}

from 2 and 4:

8: [$\overline{at_2}$:],[$\overline{bt_2}$:],[$\overline{ct_2}$:], {{$\overline{at_2}$T1},{$\overline{bt_2}$T1},{$\overline{ct_2}$T1}}

Thus by filtering out the temporal assumptions (T1 and T2) and separating the positive and negative assumptions, the label yields the datum in the expected form.

Using the label to determine the value of the datum is normally unthematic in an ATMS. This stems from the fact that the datum part of the node, once calculated is fixed and should never need to be updated. On the other hand a node can be rejustified, which will give it a new label. If the datum is calculated from the label, that would imply recalculating the datum every time the label is updated. In the diagnostic problem however a node will never be rejustified. This is because the translation of a test into the suspects always includes a statement about the test (either test working or test not working). Test findings from a different test, even if they concern the same system components, cannot be treated

as the same piece of datum because the test statement will refer to a different test.

The only problem remaining is that repeated tests (performed if a state change is suspected) could be seen as a rejustification for a datum. However this is not the case. If the repeated result is not the same, there is obviously no rejustification. If the repeated result is the same, the only action the problem solver takes is to redefine its interpretation of the temporal assumption of the test. In other words, the problem solver will change its knowledge about the temporal sequence of the tests.

5.2.3. Optimising the creation of ATMS assumptions

Since assumptions are a major ATMS resource, it is important to optimise their creation and usage.

Creating all the assumptions in advance

As far as the making of the assumptions is concerned, the first possible solution is to explicitly make all the possible assumptions about components working and not working at the beginning of the diagnosis. For a system of n components, this would immediately lead to 2n assumptions. Furthermore, since each assumption that an element is working is contradictory to the assumption that the element is not working, it would immediately create n 'nogoods'. This methodology can be rejected straight away because it is too expensive in terms of assumptions; for a system of one hundred components, there would be 200 assumptions. This would be extremely wasteful, as the symptoms and diagnosis cover only a subsection of the system, and most of the created assumptions would never be used.

Creating positive and negative assumptions just in time

A better approach is to create the working and non working assumptions about an element the first time that element is encountered in a test. The 'nogoods' $(a_i \overline{a_i})$ would then also be created dynamically when necessary. This approach requires two assumptions per component encountered during diagnosis.

5.2.4. Optimising the mapping by creating negative assumptions only

The point of the diagnosis is to try and pinpoint all the faulty elements. Therefore, at the end of the diagnosis there is no real interest in the elements that seemed to be working during the diagnosis. The only reason the positive assumptions were included in the diagnosis was to combine them with the negative assumptions and eliminate the incompatible

elements thus created. That effect can be achieved without making all those assumptions. Currently a successful test t_1 on set {a,b,c,d} (t1 /a,b,c,d/) is translated as $[\overline{t_1}:]$ or [:abcd], with the 'nogoods' $(a\bar{a}),(b\bar{b}),(c\bar{c}),(d\bar{d})$. Instead, it can be translated as $[\overline{t_1}:]$ or $[:t_1]$ with the 'nogoods' $(\bar{a}t_1)$, $(\bar{b}t_1)$, $(\bar{c}t_1)$, $(\bar{d}t_1)$. The effect is exactly the same but the number of assumptions needed is almost halved. Since assumptions are an expensive ATMS resource, this reduction is very beneficial. The only positive assumptions necessary are thus for the tests. Incidentally, this is not an implementation trick for improving performance; it has a semantic significance. When test t_1 finds that a,b,c and d are working, it is logically impossible that at the same time t_1 is working and that a is down, hence the 'nogood' $(\bar{a}t_1)$.

This algorithm is by far the most efficient and is the one used in the implementation (discussed in Chapter 6).

5.2.5. State changes and the ATMS

As far as the ATMS is concerned, state changes are recorded by invalidating the temporal assumptions of the test results affected by the state change.

5.2.6. Summary

The main characteristics of the mapping of diagnosis to the ATMS are:

- the assumptions made are about whether the components are working or not and about the temporal values of the tests;
- the label update algorithm is used to calculate the minimal candidates;
- only negative assumptions about components are made (except for the tests).

5.3. Problem solving; the blackboard perspective

As seen in Section 2.6, the blackboard architecture consists of an overall control module , and 5 levels (communication, translation, filtering, combination and testing/diagnosis) with associated level control modules and knowledge sources. The mapping of the diagnostic problem to the blackboard will thus be discussed in terms of control levels.

5.3.1. The communication level

At the communication level requests for test execution are made and reports about the state of the system (test reports) are received. The requests can be of two types; either a request for a new test to be performed, during normal diagnosis or a request for a

repetition of tests if a state change is suspected. In the first case the test is requested and the result put on the blackboard. In the second, the tests are repeated until one of them finds a state change or until they have all been repeated. The result is then passed to the overall control module which is in charge of the temporal reasoning.

The communication level is also the level at which the information about nodes known to be down is received. This information is used at the filtering level to eliminate test reports that refer to expected faults.

5.3.2. The translation level

Knowledge sources

At the translation level the test data are translated into the implicated objects as described. The way a test result is translated depends on the meaning of the test result which is system dependent. The next Chapter describes an implementation of a typical knowledge source that would be attached to this level, in the field of computer networks.

Translation control

Normally, there are several knowledge sources associated with this level, each one specialising in a specific type of test data. Therefore the triggering can be done by checking the type of the data received. Normally only one knowledge source is triggered at this level, and it can therefore be automatically scheduled.

5.3.3. The filtering level

Knowledge sources

It is often overlooked that some faults may be expected. For example if a component is known to be down for maintenance reasons, tests on it will show that it is down but should not start any alarms [Knig86]. Furthermore, if a component is scheduled to be down, but a test shows that it is up, that may be an indication of a test fault or false belief in the state component. The filtering level makes sure that neither of these events have happened. Additionally, this level is used to filter out the tests whose result have become suspect as a result of a state change. Therefore at this level, there are 5 knowledge sources.

The first knowledge source is triggered by a report of an expected failure. A report refers to an expected failure if the intersection between the list of components involved in a

failed test and the list of components known to be down is not empty. This knowledge source justifies the translated test data with the premise that one of the involved components is down, thereby turning the test data into a premise (if A implies B and A is true, then B must be true).

The second knowledge source is triggered by a report of an unexpected working component. A report refers to an unexpected working component if the intersection between the list of components involved in a successful test and the list of components known to be down is not empty. Such a report normally means that the sensor is faulty or that the information about components known to be down is faulty. The second case would occur for example if a component was scheduled to be down for maintenance but the maintenance procedure began later than expected, or if a component was brought back into service without informing the fault manager. The knowledge source will deduce that the sensor is faulty because the knowledge about components being down is treated as externally given premises which cannot be questioned.

The third knowledge source is triggered by a report of an unexpected failure. Its action is to translate the report into the implied components. In fact, as seen in the section on the ATMS mapping, the minimal candidates algorithm is implemented using the facilities of the ATMS. Therefore at this level, the relevant ATMS assumptions are created and used (together with the test report) to justify the candidates.

The fourth knowledge source is triggered by a report of a successful test which is not contradicted by any premises. Its action is also to translate the report into the implied components, and that is again done using the ATMS.

The fifth knowledge source eliminates the results of tests which have become suspect as a result of a state change. It is given the test whose findings have changed and the test results to be eliminated are determined using the algorithm described in Section 3.7.4.3. The tests are invalidated by contradicting their temporal assumptions.

Filtering control

A single report can trigger only one of the five filtering knowledge sources. Therefore the control will simply schedule the knowledge source that was triggered.

5.3.4. The combination level

Knowledge sources

At the combination level, there is only one knowledge source. It is triggered when newly translated data have arrived. Its duty is to combine these new data with the currently believed data. Since all the possible explanations are being explored in parallel, the new data have to be combined with all the currently believed explanations. As seen in Section 3.6.3, each datum from a test is translated into two (mutually incompatible) explanations; either the test is faulty or the test is working and its findings are translated into implied working or non-working components. These two incompatible explanations are kept in two separate ATMS nodes. They have to be combined with the currently known data which reflect all the test information received so far (known as the maximal contexts). The new data are not combined with any of the data previously deduced which do not reflect all the known test data.

The meaning of this can be illustrated by showing how the maximal contexts are calculated. Let us imagine that tests t_1, t_2 and t_3 are performed in that chronological order. After t_1, there are two maximal contexts t_1 and $\overline{t_1}$. t_2 is translated into t_2 and $\overline{t_2}$, and combined with the maximal contexts t_1 and $\overline{t_1}$ giving the contexts $(t_1 t_2)$, $(\overline{t_1} t_2)$, $(t_1 \overline{t_2})$ and $(\overline{t_1}\,\overline{t_2})$. After t_3, there are eight maximal contexts $(t_1 t_2 t_3)$, $(\overline{t_1} t_2 t_3)$, $(t_1 \overline{t_2} t_3)$, $(\overline{t_1}\,\overline{t_2} t_3)$, $(t_1 t_2 \overline{t_3})$, $(\overline{t_1} t_2 \overline{t_3})$, $(t_1 \overline{t_2}\,\overline{t_3})$ and $(\overline{t_1}\,\overline{t_2}\,\overline{t_3})$. The nodes created after t_3 are not combined with the nodes created after t_1 alone, because these contexts are not maximal. Given the set of performed tests, it is very easy to build those maximal contexts. Since each test has a corresponding assumption, it is also easy to ask the ATMS to retrieve the nodes containing the environments which correspond to the current maximal contexts.

As the final action of the combination phase, the data from the different contexts are unified into one node which is supported by the temporal assumptions only. This enables the testing modules to examine solely this one node, thus facilititating their task. This unification into one node is also semantically correct; if there has been no state change, the explanation for the fault must be contained in that one node.

Combination control

Each combination performed corresponds to a triggering (i.e a consumer is created). Since the algorithm looks at all the maximal contexts, all the consumers will be scheduled.

5.3.5. The testing/diagnosis level

Knowledge sources

At the testing/diagnosis level, there are three knowledge sources: a diagnoser that makes the final diagnosis, a normal tester that determines the next best test, and a repeat tester that requests tests to be repeated when a state change is suspected.

The diagnoser is triggered by the arrival of newly combined data that satisfy the chosen terminating condition for the diagnostic process as discussed in Section 4.8. The diagnoser states the final diagnosis and informs the layer control (which will inform the overall control) that the process has been finished.

The state change tester is triggered by the arrival of newly combined data that seem to suggest a state change. The different methodologies for raising the suspicion were discussed in Chapter 3. The action performed is to request a repetition of all the tests.

The normal tester calculates the next best test according to the algorithms developed in Chapter 4. It is triggered by the arrival of newly combined data that satisfy neither the terminating conditions nor the state change suspicion conditions.

Testing control

Normally only one of the knowledge sources on this level is triggered; therefore the control simply schedules the created consumer. If there is a final diagnosis, the overall controller is also informed that the diagnosis is terminated.

5.3.6. Overall control

The overall control supports two control loops:

- one is the data path through the different layers if no state change is suspected;

- the other control loop is the temporal reasoning control loop which coordinates the 3 phases of the state change algorithm (suspecting a state change, confirming a state change and recovering from a state change), which are performed at different blackboard levels, as depicted in Figure 5.1.

The overall controller keeps a record of all the tests that have been performed so far with their results and of the tests' chronological ordering. If a state change is suspected at the

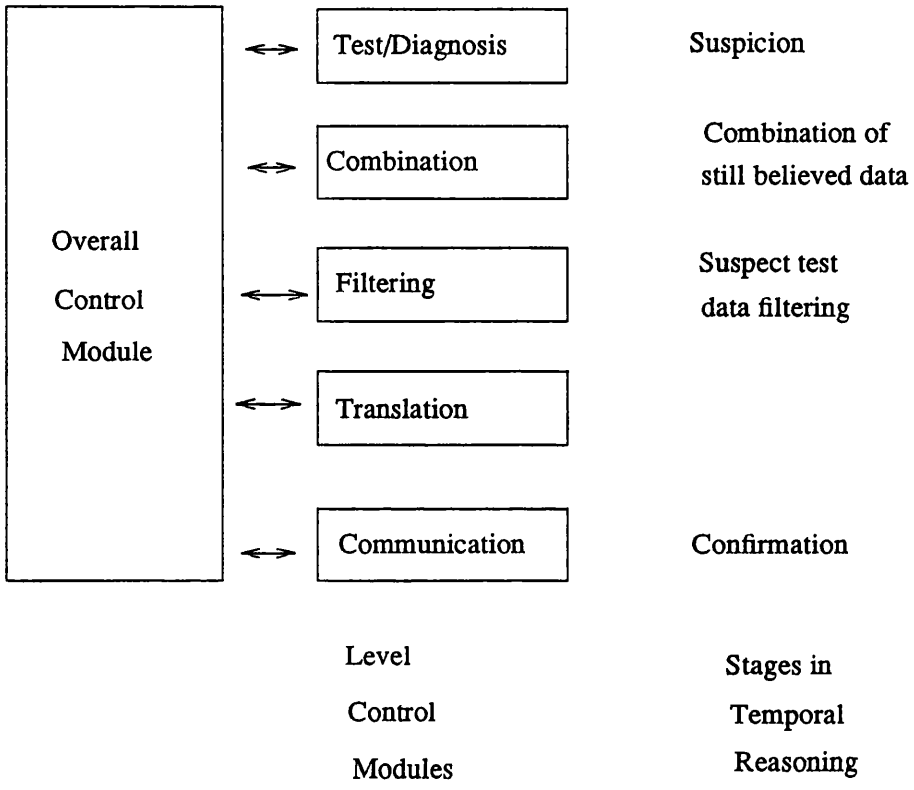


FIGURE 5.1 State change algorithm and the blackboard levels

testing level, the overall controller passes the tests to be repeated and their results to the communication layer. The reply is either that all tests still give the same result or that test t_i has changed its findings. In the first case, the overall controller simply returns the control to the testing/diagnosis control level. In the second case, the controller passes the identity of t_i and its new result to the translation level (where it is translated). Then, the result of t_i is passed to the the filtering level where the suspect test results are filtered out. The believed test identities are then passed to the combination level, which uses them to perform the combinations. Finally, the overall controller changes its chronological interpretation of the tests, and gives control to the testing level, thus terminating the temporal reasoning loop.

5.4. ATMS and blackboard cooperation

The previous sections showed the mapping of the diagnostic problem from the point of view of the ATMS and from the point of view of the blackboard. This section shows how the two cooperate and complement each other in the search for the solution.

5.4.1. The normal data path through the ATMS/blackboard environment

This section describes the data path when no state changes are suspected.

Initially the test data arrive at the communication layer. These data are passed to the ATMS to be justified with a temporal assumption. The ATMS returns the node number N1 in which that data was stored. N1 is then passed to the translation level.

At the translation level, the knowledge sources translate the data from N1 into the implicated objects (suspects) and that is given to the ATMS which stores it in node N2. N2 is passed to the filtering level. For example, the data from test t_1:noroute (A-B) could be translated to t_1(A,B,C,D,E) (reflecting the intermediate components between A and B).

At the filtering level, the knowledge sources determine whether the data are expected and whether they are contradictory to the known facts. In the first case, they are ignored, in the second an alarm is raised. Otherwise, the knowledge sources translate the test data into the two possible hypotheses:

- test not working or

-test working (with an interpretation of the test result)

and determine which assumptions have to be created in order to support these hypotheses. The ATMS is then requested to put these hypotheses into nodes, to create the requested assumptions, and to justify the candidates with the newly created assumptions and the test result. The node numbers of the newly created hypotheses (two of them, N3 and N4, corresponding to the "test working" context and "test not working" context) are returned by the ATMS and passed to the combination layer. For example, t_1<a,b> would be translated into:

N3:[],$\{\overline{t_1}\}$

N4:[],$\{\{\overline{a}T1\},\{\overline{b}T1\}\}$

The datum part of the node is empty because it is filled later once the data have been combined, using the label as described in Section 5.2.2. The temporal assumption T1 appears in the label of N4 because N4 is justified by the test result (which contained T1 in its label).

At the combination level, the combination knowledge source first calculates the existing maximal contexts, as described in Section 5.3.4. It then asks the ATMS to retrieve the nodes that correspond to these maximal contexts and to create a consumer for each possible combination of N3 or N4 with a maximum context node. Once the ATMS has performed that phase (the triggering phase) and returned the node numbers of all the newly created consumers, they are all scheduled, and the action performed consists of calculating the datum part of the node from the labels, as described in Section 5.2.2. The candidates from the different contexts are then gathered into one node, which is passed to the ATMS. The ATMS justifies this one node with the temporal assumptions and the node is then passed on to the testing level.

At the testing level, the candidates are examined. If the diagnosis termination conditions are satisfied, the user is given an explanation for the faults, otherwise the knowledge sources calculate the next best test and the ATMS is asked to put it onto the communication level.

This interaction is summarised in Figure 5.2.

5.4.2. The state change data path through the ATMS/blackboard environment

This section describes the data path when a state change is suspected.

The state change suspicion phase occurs when a node N5 containing the different hypotheses is passed from the unification level to the testing level which decides that there may have been a state change. A request for test repetition is then passed to the overall controller which asks the communication level to repeat the tests. When a test t_i finds a state change, it is processed normally until the filtering level. There,the ATMS is instructed to eliminate suspect test results by contradicting their temporal assumptions. Following that, the interactions with the ATMS are the same as in the normal data path.

5.5. Other issues

5.5.1. Obtaining continuous diagnosis

The system presented so far does not have an idea of continuous diagnosis; it performs one diagnosis session and stops when it finds an explanation that satisfies all the currently believed data. To turn it into a continuous reasoning system requires:

- changing the system beliefs based on the diagnosis just performed;

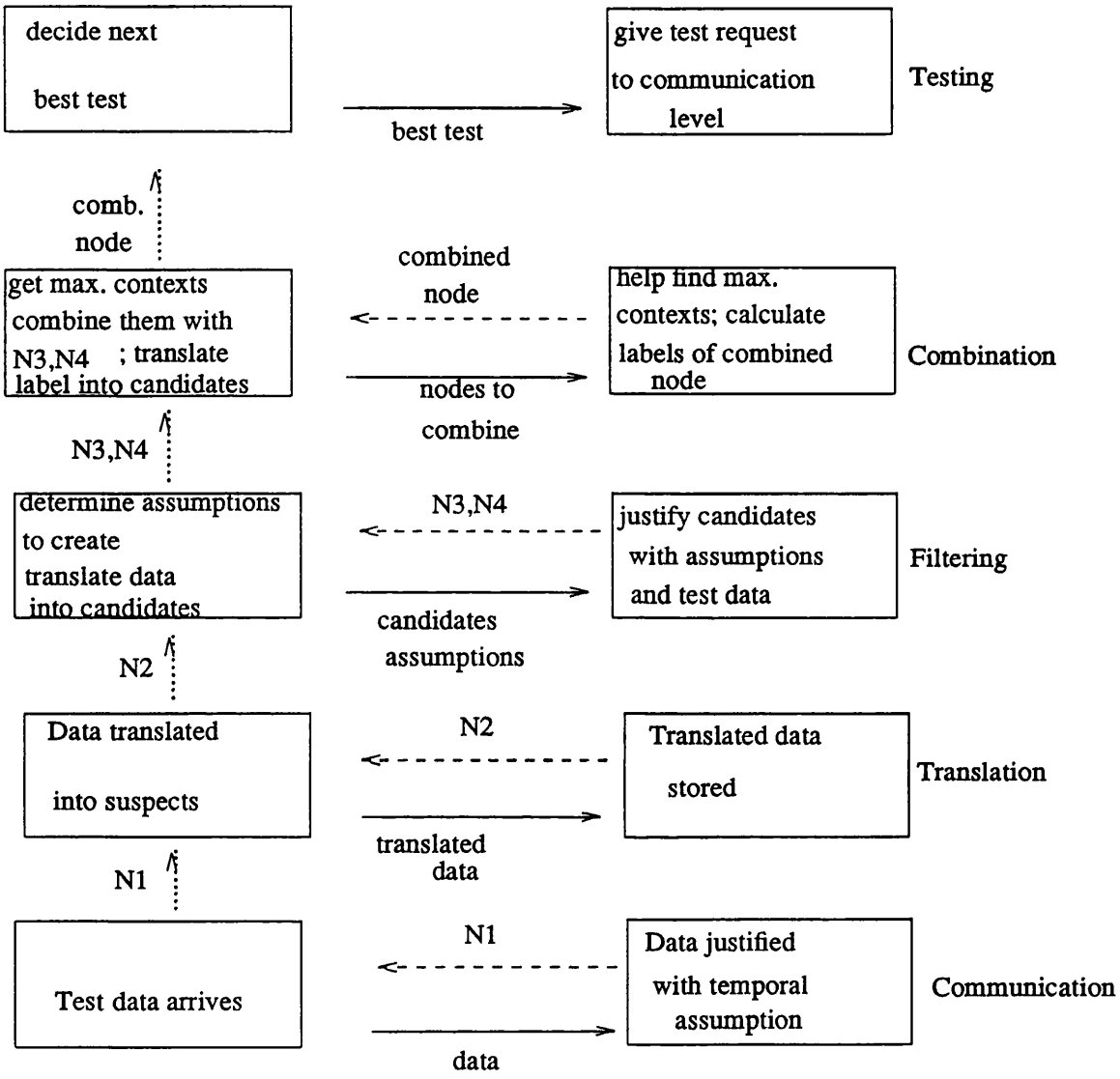


FIGURE 5.2 Blackboard/ATMS interaction

- erasing the latest diagnostic steps from the working memory, once its findings have been incorporated into the current set of beliefs.

Erasing the diagnostic steps can easily be done by deleting all the ATMS nodes created in the course of the diagnosis.

At the beginning of diagnosis, there are 3 types of belief:

- the a priori probabilities of failure of all the components in the system;

- the list of components that are known to be down;
- the (implicit) belief that the other components are up.

At the end of diagnosis, there is normally a high probability tuple that explains all the currently believed test data. This tuple contains a set of one or more components whose failure would explain all the symptoms.

One way to translate this belief is to make the status of all the components of this tuple as "known to be down", and therefore treat it in the same way as externally supplied data about components known to be down. This is not satisfactory because the proposed diagnostic method is probabilistic; there is no proof that the tuple is definitely the cause of the symptoms, only that it is the most probable cause. This solution would be mixing absolute beliefs (externally provided) with relative, probabilistic beliefs (internally generated), and could lead to further propagation of false beliefs in the subsequent diagnosis.

An alternative is to to change the a priori probabilities of failure of the components in the candidate to reflect the new belief that these components are down. This again would be misleading and unthematical. The a priori probabilities of failure are long term beliefs about the components and should not be radically affected by one single diagnosis. The diagnosis ends in a short term belief about the components, the a priori probabilities represent long term beliefs.

Thus, continuous diagnosis requires an additional parameter which would describe the short term result of the diagnosis. The problem stems from the fact that the belief that the components are up is implicit; if a component is not known to be down, it is assumed (without any explicit representation) to be up. If there is no way to explicitly represent the belief that a component is up, there is also no way to explicitly represent the belief that a component is down.

To get around this problem, an additional parameter, associated with each component is needed. This parameter will represent the short term belief about the state of the component. Thus each component will be linked to two probabilities; the long term (a priori) probability of failure LP and the short term probability of failure SP, represented as the tuple (component,LP,SP). Now the short term belief that a component is up can be represented by making SP equal to LP. If at the end of diagnosis, the most probable tuple has probability p1, the SP's of the components included in the non-working part of the tuple can be changed to some very high probability, or even p1 (since that is after all the probability of the candidate anyway). When a component returns to a working state, its SP can be changed back to its LP. The probabilities used by the testing algorithms to

estimate the entropies should obviously be the short term probabilities.

5.5.2. Dealing with the knowledge sources

At the beginning of the diagnosis, all the knowledge sources are inserted into the ATMS. Every time a knowledge source is used to derive some datum, the knowledge source becomes part of the justification for the newly derived datum. This facilitates the tracing of the justifications when explaining the reasoning process.

5.6. Conclusion

This Chapter has explained how the ideas presented in Chapters 2, 3 and 4 can be brought together to provide the system-independent elements of a tool for fault diagnosis in complex dynamic systems. The first part showed how to translate the diagnostic problem into the pure ATMS architecture. The second part described the problem solving paradigm purely from the blackboard point of view and showed how the knowledge sources cooperate to create a solution. The third part showed how the ATMS and the blackboard cooperate in order to achieve a solution. The final part showed a method for obtaining continuous diagnosis. The novel contributions of this Chapter include:

- a method for minimising the number of assumptions created during the diagnostic process in order to optimise the usage of the ATMS;
- the usage of the ATMS label to calculate the problem solver data;
- the cooperation of the ATMS and the blackboard in the search for the solution;
- the adaptation of the blackboard/ATMS architecture to continuous diagnosis.

These improvements optimise the algorithms from the previous Chapters and constitute the final elements of the proposed fault dignosis shell.

CHAPTER VI

IMPLEMENTATION ISSUES, PRELIMINARY VALIDATION AND FUTURE WORK

6.1. Introduction

The previous four Chapters developed a conceptual shell for fault isolation in complex dynamic systems. This Chapter presents the steps taken to perform the preliminary validation of the proposed shell and discusses the work that will still be needed in order to achieve the full validation of the proposed system. The first part summarises the major implementation issues associated with the fault isolation shell. The second part analyses the performance issues relevant to the fault isolation shell. The third part describes the basic principles of computer networks relevant to fault isolation and the development and implementation of a knowledge source to translate routeing failures in a computer network into the implied suspect components. This knowledge source is then used in conjuction with the fault isolation shell to simulate experimental data from a hypothetical network and illustrate the operation of the fault isolation shell. The final part discusses the future work needed in order to further validate the proposed shell and turn it into a fully fledged system.

6.2. Shell implementation status and issues

The fault isolation shell developed in the previous four Chapters is fully implemented in Prolog and all the examples in this Thesis have been tested in the implemented system. Prolog was chosen for its ability to produce rapid prototypes rather than any of its other characteristics.

Since the ATMS is at the heart of the problem solving mechanism, it is important to optimise its implementation as much as possible. Several optimisation issues are presented in [Klee87] concerning the way the data structures are organised and the way the algorithms are implemented. Of those, by far the major one concerns the representation of assumptions and the calculation of labels. This is crucial because the major

function of the ATMS is to create and update the labels. In the implementation, sets of assumptions (forming environments) are represented as bit vectors. Each assumption has a unique position in this bit vector. If a bit is set, then the assumption is present in that particular environment. Therefore the union of two environments can be computed by performing the logical *or* operation on the bit vectors. Furthermore, set1 is a subset of set2 if the result of and-ing the bit vector of set1 with the complement of set2 is zero. Both these operations are very efficient and cover the label creation algorithms described in Section 2.5.4.

This implementation means that assumptions are expensive since they consume valuable bit positions. It thus pays to try and minimise the number of assumptions made; therefore the implementation used the mapping algorithm which only creates the negative assumptions (as described in section 5.2.4).

As far as the blackboard is concerned, all the control modules and knowledge sources are implemented as Prolog predicates. The combination module uses the ATMS label to work out the current suspects (as described in section 5.2.2). Only one of the state change suspicion approaches described in section 3.7.4.1 is implemented, namely the test data inconsistency approach. The test optimisation algorithm of Chapter 4 is fully implemented but none of the performance-enhancing heuristics of section 4.7 have been included in the implementation. However, the performance effects of one of these heuristics (the examination of the most probable candidates only) is evaluated in the next section.

6.3. Performance issues

This section examines the performance issues associated with the fault isolation shell. The aim is to establish the dynamic system parameters which have an important effect on the performance of the proposed shell, see how variations in the parameters affect this performance and establish that the proposed system can scale up to the size of real-world problems.

6.3.1. Fault isolation algorithm characteristics

The basic fault isolation algorithm begins with a set of suspects, makes a succession of test requests and finishes when there is enough information to reach a conclusion. The cost of the algorithm will therefore depend on the number of iterations necessary in order to reach the conclusion and the cost of each iteration.

As seen in figure 5.2 at each iteration the following steps are performed:

a) The newly arrived data is justified with a temporal assumption.

b) The data is translated into implicated objects.

c) The data is filtered for expected faults.

d) The alternative hypotheses are created for each test (i.e test working or test not working).

e) The maximal contexts are created.

f) The new data is combined with existing data and inconsistent, subsumed and duplicated candidates are eliminated.

g) The different contexts are unified.

h) The candidate probabilities are normalised to 1 and the candidates sorted.

i) The tests are evaluated and the best test is selected.

j) The best test is performed.

Phases a), c), d), e), g) and h) can be ignored because they contribute little to the overall cost.

Phase b) is a system specific phase. The time it takes will depend for example on how the system translates the symptoms; translations using deep knowledge will normally be slower than translations using shallow knowledge. We will assume that the average time t1 this phase takes for a specific system is a given parameter.

Similarly phase j),the time t2 it takes to perform the best test is a system-specific parameter and is assumed to be given.

Phases f) (combination of data) and i) (test evaluation) are the main time consuming activities in each iteration, and we will therefore try to estimate their respective time costs t3 and t4.

Section 6.3.2 provides a methodology for calculating the expected number of iterations I in a given system for a given number of initial suspects, and the subsequent sections 6.3.3 and 6.3.4 estimate the costs of t3 and t4, that is the total cost of the minimal candidate generation algorithm and the total cost of the test evaluation algorithm.

Assumptions

All the elements have the same probability of failure p. This is the worst case assumption.

p is a quite small number (of the order 10^{-3} or less).

Since each iteration involves a number of expensive operations, we assume that the aim is to minimise the number of iterations. Therefore, it is important to look at all the available tests in order to choose the best one.

6.3.2. Average number of iterations

The unit of measure used in evaluating the set of suspects is the entropy. The starting point of the diagnosis is a set of N_0 suspect elements of equal probability. The sum of these probabilities is normalised to 1 and therefore each one of them becomes equal to $1/N_0$. Hence the starting point of the diagnosis is the entropy of that set, which is:

$$H_B = -\sum_{i=1}^{i=N_0} \frac{1}{N_0} \log\left(\frac{1}{N_0}\right) = \log(N_0)$$

The diagnosis is finished when one of the suspect elements has a probability of failure greater than α where α is given for a specific system. The entropy of such a suspect set can be approximated by:

$$H_F = -[\alpha \log(\alpha) + (1-\alpha) \log(1-\alpha)]$$

Therefore the goal of the diagnosis is to reduce the entropy of the set of suspects from H_B to H_F, i.e by:

$$\Delta H = \log(N_0) + [\alpha \log(\alpha) + (1-\alpha) \log(1-\alpha)]$$

For example, if the initial number of suspects is 100 and a suspect is accepted if its probability is greater than 0.95, the aim of the diagnosis is to reduce the entropy by:

$$\Delta H = \log(100) + 0.95\log(0.95) + 0.05\log(0.05) = 4.6 - 0.2 = 4.4$$

Given N_0 (the initial number of suspects) and α (the suspect probability that ends the diagnosis, which is given), the number of necessary iterations is equal to:

$$I = \Delta H \ \text{DIV} \ (\overline{H}_t) + 1$$

where DIV is the operation which gives the integer part of the result of the division of two numbers and $\overline{H}_t$ is the average entropy change per chosen test.

Average entropy change per chosen test

The average entropy change per chosen test is a system specific characteristic and therefore strictly speaking outside the fault isolation model. We could just assume that it is a given parameter for the system under observation. However, since it is not a common system value, we will attempt to give one possible methodology for calculating this value.

As seen in section 4.5.1, the expected entropy to which a test will lead is obtained by combining the expected entropies of the failed and successful test, as shown below:

$$H_e(t_i) = p(t_i=f)\ H(t_i=f) + p(t_i=w)\ H(t_i=w)$$

Intuitively, two major parameters determine by how much a given test will reduce the current entropy. One is the way in which the test splits the current list of suspects into two groups of suspects with associated probabilities. The other is the number of new elements of the system that the test is bringing in. The new elements are those which were not contained in any of the current suspects. Therefore, the two parameters are defined as follows:

X, which is the sum of the probabilities of the suspects which will remain if the test fails.

C which is the number of new components the test is bringing in.

Theoretically, the best test that can be chosen is one that performs binary search on the list of suspects [Klee87] and that does not bring in any new elements (i.e C=0). Binary search means that the test splits the list of suspects into two groups of equal probability, i.e X=0.5. Therefore of all the tests available, the best test is the one where X is as close

as possible to 0.5. If several tests yield the same X, the best one is the one that minimises C (since it minimises the number of components involved).

Since we want to estimate the average entropy change caused by choosing the best available test at any time, X will be the average probability of the suspects remaining if the best test is chosen and it fails and C will be the average number of new elements brought in by the best test. These two parameters can be statistically obtained from the system under observation, and are assumed to be given.

Failed test entropy

Since:

- there are N elements of probability p;
- the best test will split the N elements into two groups of respectively XN elements and (1-X)N elements,

the best test will lead to XN elements of probability p and (1-X)CN elements of probability p^2.

For example, if the suspects are $[\overline{t1}:]$, $[\overline{a}:]$, $[\overline{b}:]$, $[\overline{c}:]$ and the best test is t2(a,b), the new suspects if t2 fails are: $[\overline{a}:]$ and $[\overline{b}:]$ with probability p and $[\overline{t1t2}:]$,$[\overline{ct2}:]$ with probability p^2. The new test (in this case t2) always counts as a new component; therefore, C is always greater than or equal to 1.

Since p is quite small, the (1-X)CN elements of probability p^2 can be replaced with 1 element of probability $(1\text{-}X)CNp^2$. To normalise the probabilities of all the components to 1 (which has to be done in order to calculate the entropy), the following equation has to be solved for β:

$\beta XNp + \beta(1\text{-}X)NCp^2 = 1$, i.e :

$$\beta = \frac{1}{XNp + (1-X)NCp^2}$$

Thus if the test fails, there will be XN elements of probability:

$$\frac{1}{XN + (1-X)NCp}$$

and 1 element of probability:

$$\frac{(1-X)Cp}{X+(1-X)Cp}$$

Given these probabilities, and the entropy formula which states that

$H=-\sum p_i \log(p_i)$

it can be shown that the entropy of a failed test is equal to:

$$H_f = \frac{X}{X+(1-X)Cp}\log(N) - \frac{(1-X)Cp}{X+(1-X)Cp}\log(1-X)Cp + \log(X+(1-X)Cp)$$

Successful test entropy

If a test shows that a set elements is working, either the test is faulty or the set of elements is working. The assumption that the test (and the elements) are working produces (1-X)N elements of probability p. The assumption that the test is faulty produces XN elements of probability p^2, which can be approximated to one element of probability XNp^2.

To normalise the probabilities of all the components to 1 the following equation has to be solved for β:

$\beta(1\text{-}X)Np + \beta XNp^2=1$, i.e :

$$\beta = \frac{1}{(1-X)Np + XNp^2}$$

Thus if the test succeeds, there will be (1-X)N elements of probability:

$$\frac{1}{(1-X)N + XNp}$$

and 1 element of probability:

$$\frac{Xp}{(1-X) + Xp}$$

Given these probabilities, and the usual entropy formula, it can be shown that this entropy is equal to:

$$H_w = \frac{(1-X)}{Xp+(1-X)}\log(N) - \frac{Xp}{Xp+(1-X)}\log(Xp) + \log(Xp+(1-X))$$

Expected entropy

Since

$H_e(t_i) = p(t_i=f)H_f + p(t_i=w)H_w$,

$p(t_i=w) = (1-X)(1-p)^C$, and

$p(t_i=f) = 1 - (1-X)(1-p)^C$

(by the equations given in section 4.5.2), it is possible to calculate the average entropy change during diagnosis per test performed $H_e(t_i)$. It is equal to:

$$H_e(t_i) = [1 - (1-X)(1-p)^C] \left[\frac{X}{X+(1-X)Cp}\log(N) - \frac{(1-X)Cp}{X+(1-X)Cp}\log(1-X)Cp \right.$$

$$\left. + \log(X+(1-X)Cp) \right]$$

$$+ (1-X)(1-p)^C \left[\frac{(1-X)}{Xp+(1-X)}\log(N) - \frac{Xp}{Xp+(1-X)}\log(Xp) + \log(Xp+(1-X)) \right]$$

The expected number of iterations I to reach a solution, given an initial number of suspects N, X, C and p is then given by the previously mentioned formula:

$I = \Delta H \quad DIV \quad H_e(t_i) + 1$

Examples of the expected number of iterations

Assuming that a suspect is accepted when its probability is greater than 0.95, Table 6.1 gives some simple examples of the expected number of iterations I, given selected N_0 (initial number of suspects), X (average probability of the suspects remaining if the best test is chosen) and C (average number of new elements brought in by the best test). The probability of failure p of each component was assumed to be 0.00001.

N0	C	X	I
(initial elements)	(new elements)	(probability split)	(iterations)
100	5	0.5	8
100	5	0.4	9
100	5	0.3	9
100	5	0.2	11
100	5	0.15	13
100	5	0.1	16
100	5	0.05	26
50	5	0.3	8
100	10	0.3	9
100	30	0.3	9
10	5	0.3	6
20	5	0.3	7

TABLE 6.1; Required number of iterations

Three main points can be deduced from this table:

- The number of iterations necessary to reach the solution increases as X diverges from 0.5 (binary search) but remains of the same order as binary search between X=0.5 and X=0.2. It then begins to increase quite substantially.

- The number of iterations necessary to reach the solution increases logarithmically as N_0 increases.

- The number of new elements C does not have any effect on the number of iterations; by comparing lines 3,9 and 10 of the table, it can be seen that for the same initial number of elements (N_0=100) and the same probability split (X=0.3), an increase in the number of

new elements C from 5 to 30 has no effect on the number of iterations I, which remains equal to 9. This is because the effect of C on the expected entropy change is too small for the examples given and for the chosen probability p. C will however affect the cost of each iteration.

6.3.3. Minimal candidate generation cost (t_3)

The algorithm to calculate the minimal candidates is known to be NP complete [Karp72] and thus potentially expensive in computational terms, since it does not grow linearly. We will briefly attempt to estimate the cost of the algorithm in the average case, using a reasonably efficient implementation of the algorithm.

Estimating the average size of the chosen tests

Let $\overline{t_c}$ be the average number of elements tested by the best test during diagnosis (called the size of the test). When the first symptom appears, the size of the best test t1 would be approximately equal to: XN+C (X,N and C are as defined in the previous section) since all the elements N are of equal probability and t1 is supposed to split N into 2 groups of probability XN and (1-X)N. As diagnosis progresses, one would expect the size of the best test to become smaller and smaller, since it has to distinguish mainly between the most probable elements. By the end of the diagnosis, a test size of 1 will usually suffice to confirm the most probable candidate. We will therefore assume that the average size of the chosen test is approximately equal to:

$$\overline{t_c} = \frac{XN}{2} + C$$

Estimating the number of suspect elements left after each iteration

Let N_i be the number of currently suspect elements and let t_i be the chosen test. We will assume that the chosen test has an intersection with approximately XN_i elements of the currently suspect elements (i.e that the size of S_{ip} as defined in Chapter 4 is XN_i).

If t_i succeeds, the new number of suspects left N_{i+1} will be equal to N_i. This is easily seen in an example. Let the current suspects be: $[\overline{a}:]$, $[\overline{b}:]$ and $[\overline{c}:]$ and let the result of t_i be /a,b/ with interpretations $[\overline{t_i}:]$ and [:ab]. The new minimal candidates left after this test are: $[\overline{c}:]$, $[\overline{at_i}:]$ and $[\overline{bt_i}:]$; their number is the same as the current number.

If t_i fails, the elements of S_{ip} will stay suspect (as seen in Chapter 4), giving XN_i suspects. The C new elements of the test will combine with the remaining $N_i - XN_i$ elements of the current list of suspects, giving $C*(1-X)*N_i$ new suspect elements. This

corresponds to the $C_{i,U_{ip}}$ elements of Chapter 4. Most of the other theoretically possible elements will not be minimal. Thus the number of suspect elements left if the test fails is approximately: $N_{i+1} = XN_i + C(1-X)N_i$.

Assuming that the test has an equal probability of failing or succeeding 0.5, the expected number of elements is:

$$N_{i+1} = \frac{N_i + XN_i + C(1-X)N_i}{2}$$

$$= \frac{(1+X) + C(1-X)}{2} N_i$$

Minimal candidate generation cost for one iteration

The simple minimal candidate algorithm consists of three phases; combination of all the suspects, checking the 'nogoods' and elimination of the subsumed and duplicated elements. This algorithm is expensive because it does not exploit the properties of the set of suspects to minimise the number of operations to be performed. A more efficient algorithm first intersects the $\overline{t_c}$ and N_i elements, finding the new elements brought in by the test (C of them) and separating the current suspects into subsets S_{ip} and U_{ip} (as defined in Chapter 4). The S_{ip} elements remain minimal candidates and do not need to be combined. The C elements of $\overline{t_c}$ are combined with the U_{ip} elements which are automatically minimal candidates. The other elements of $\overline{t_c}$ are also combined with the U_{ip} elements and then checked with the S_{ip} elements to eliminate subsumed and duplicated elements. Therefore, given $\overline{t_c}$ and N_i, the cost of this algorithm is approximately:

$\overline{t_c}\ N_i$ intersection operations;

$\overline{t_c}\ (1-X)N_i$ union operations (since we assumed that the size of S_{ip} is XN_i in the previous section);

$X\ N_i\ (\overline{t_c} - C)\ (1 - X)\ N_i$ subset operations.

Out of these operations, the major cost are the subset operations for large N_i. Each subset operation corresponds to 3 basic computer operations: complementing a bit vector, and'ing the bit vector with another bit vector and checking the result. Therefore the total number of basic operations is of the order of:

$3\ X\ N_i\ (\overline{t_c} - C)\ (1 - X)\ N_i$

Examples of the total cost of the minimal candidate generation algorithm

The parameters N_0 (initial number of elements),X,C and p determine the expected number of iterations, the growth in the number of suspects and the average size of the chosen test. These in turn determine the total number of operations necessary to perform the minimal candidate generation during a diagnosis. Table 6.2 gives some simple examples of the total number of basic operations performed during the minimal candidate generation phases of the diagnosis.

N0	C	X	I	O
(initial elts)	(new elts)	(prob. split)	(iterations)	no. of basic operations
10	3	0.3	6	10^4
20	3	0.3	7	$2.32*10^5$
50	3	0.3	8	$1.05*10^7$
100	3	0.3	9	$2.43*10^8$
50	3	0.5	7	$2.41*10^6$
50	3	0.4	8	$8.3*10^6$
50	3	0.2	10	$1.05*10^8$
50	3	0.1	14	$1.14*10^{10}$
50	2	0.3	8	$9.44*10^5$
50	5	0.3	8	$5.22*10^8$
50	10	0.3	8	$3.27*10^{11}$

TABLE 6.2; Total cost of minimal candidate operations

These figures mainly give the order of the cost of these operations. The main point is that this cost increases exponentially as the number of iterations increases (which as seen before is determined mainly by N_0 and X) and as the size of C increases, which affects the number of suspect elements in each iteration. Each of these operations corresponds to basic computer instructions and is therefore extremely fast. This cost corresponds to the previously defined parameter t_3.

6.3.4. Calculating the cost of the test evaluation algorithm (t_4)

Test cost for one iteration

As seen in Chapter 4, each test is evaluated first by comparing every component examined by the test with every current suspect to determine the subgroups S_{ip}, U_{ip}, A_i, B_i and C_i, and second by calculating the expression:

$$\left[p(B_i=f) + p(C_i=f) + \left(p(C_i=w)\, p(U_{ip}=f)\right)^2 - 1 \right] \sum_{i=1}^{i=u} p(x_{iu}) \log p(x_{iu})$$

$$-\left[1 - p(U_{ip}=f) + p(U_{ip}=f)\, p(B_i=f) + p(U_{ip}=f)\, p(C_i=f) \right] \log p(t_i=f)$$

$$+ \left[\sum_{i=1}^{i=b} p(x_{bi}) \log p(x_{bi}) + \sum_{i=1}^{i=c} p(x_{ci}) \log p(x_{ci}) \right.$$

$$\left. + \left(p(C_i=w)\, p(U_{ip}=f)\right)^2 \log(p(C_i=w)\, p(U_{ip}=f)) \right] p(U_{ip})$$

Let t_s be the size of the test and N_i the number of suspect elements in the current iteration.

The classification of the components into subgroups is going to take $t_s\, N_i$ steps. This is because each test component has to be compared with each suspect component. For each test component, the comparisons are trying to determine if the test component is equal to any suspect component (i.e belongs to A_i), is a subset of any suspect component (i.e belongs to B_i) or is neither (i.e belongs to C_i). The equality test is a basic operation, and the subset test involves three basic operations; complementing, anding and checking. Thus there are:

$4\, t_s\, N_i$ basic operations to classify the components. this cost can be ignored.

In the estimation of the entropy, the operations performed are additions, multiplications and logarithmic calculations. Since the logarithmic calculations take much longer than the others (of the order of 10^3 times longer than basic operations such as addition), the other operations can be ignored.

The number of times that the logarithmic operation has to be performed is:

$u + 1 + b + c + 1 = 2 + u + b + c \tilde{=} u + b + c$

u can be approximated with N_i and b+c can be approximated with t_s. Both of these approximations are overestimates (since they assume that S_{ip} and A_i are empty).

Thus the cost of estimating a test is approximately:

1000 t_s N_i basic operations.

Examples of the total cost of the test evaluation phase

Let $\overline{t_s}$ be the average test size for the system under observation ($\overline{t_s}$ is a measurable parameter for any system) and let n_t be the total number of tests in the system. The number of operations at each iteration will clearly be n_t times the number of operations necessary to estimate the average test. The number of current suspects N_i (which determines the cost) varies from iteration to iteration and depends on other parameters like X, C and N_0. Table 6.3 shows some examples of the total cost of one test evaluation, for selected values of t_s, N_0, C and X.

N0	C	X	$\overline{t_s}$	I	O
init. elts	**new elts**	**prob. split**	**av. test size**	**iter.**	**cost of test eval.**
10	3	0.3	5	6	$9.42*10^5$
10	3	0.3	10	6	$1.88*10^6$
10	3	0.3	20	6	$3.77*10^6$
20	3	0.3	5	7	$3.3*10^6$
50	3	0.3	5	8	$1.42*10^7$
100	3	0.3	5	9	$4.91*10^7$

TABLE 6.3; Total cost of basic test evaluation for the average test

Since the table shows the total evaluation cost for one test only, the time taken to evaluate all the tests would be unacceptable. The main reason for this is that the basic test evaluation algorithm looks at *all* the current suspects, even though most of the elements contribute very little to the entropy. A much faster method (already mentioned in Chapter 4) is to constrain the test evaluation algorithm to the most probable elements only. Table 6.4 shows the total cost of evaluation per test for the same parameters as Table 6.3, with the difference that the evaluation algorithm only looks at the N_0 most probable elements (which is still an overestimate of the number of suspect elements that have to be taken into acount for test evaluation).

N0	C	X	$\bar{t_s}$	I	O
init. elts	new elts	prob. split	av. test size	iter.	cost of test eval.
10	3	0.3	5	6	$2.5*10^5$
10	3	0.3	10	6	$5*10^5$
10	3	0.3	20	6	10^6
20	3	0.3	5	7	$6*10^5$
50	3	0.3	5	8	$1.75*10^6$
100	3	0.3	5	9	$4*10^6$

TABLE 6.4; Total cost of improved test evaluation for the average test

The main characteristics of the test evaluation algorithm is that the costs increase linearly with the size of the average test and that they are proportional to the number of iterations. The total cost of the evaluation algorithm t_4 is obtained by multiplying the average test cost by the number of tests.

6.3.5. Average total time taken to perform a diagnosis

The previous sections showed how to calculate the expected number of iterations I, the total cost t_3 of the minimal candidate generation algorithm and the total cost of the test evaluation algorithm t_4 given the parameters N_0 (initial number of suspects), C (number of new elements introduced on average by the chosen test), X (the average probability split of the best test), p (the components' probability of failure) and $\bar{t_s}$ (the average test size). Given the two other system-specific parameters t_1 (average time to translate symptoms into suspects) and t_2 (average time to perform a test), the total time to perform a diagnosis is:

$$I*t_1 + I*t_2 + t_3 + t_4$$

For example, for a system with 100 components (medium sized system in the case of computer networks) where C=3 and X=0.3 and $\bar{t_s}$=5 the total number of operations needed for the fault isolation algorithm is of the order of 10^8. This translates into several

seconds of real time, which is a perfectly acceptable total diagnostic time in computer networks.

6.3.6. Issues of parallelism

Both the minimal candidate generation algorithm and the test evaluation algorithm can be fully parallelized at every iteration. This is because both of them consist of a large number of identical and independent operations, which can thus be performed at the same time. The effect of parallelization to divide the cost of these phases at each iteration by the number of parallel processors. This is very important because those two operations formed the major cost of the fault isolation algorithm. The rest of the operations have to be performed sequentially.

6.3.7. Maximum time limit for a diagnosis

The maximum time limit for a diagnosis is system dependent. In time critical systems, this limit may be very low and vice versa. However, there is one absolute limit on the time taken to perform a diagnosis. The average time to perform a diagnosis must be less than the mean time between failures. In a system of N components, each with a probability of failure p, the mean time between failures is $\frac{1}{Np}$. The average time to perform a diagnosis therefore has to be substantially smaller than $\frac{1}{Np}$.

6.3.8. Time taken to deal with state change suspicions

When a state change has been established, one can consider the diagnosis to have been restarted on a new system state and thus reset the time counter to 0. In other words, it is not necessary to add the time taken to deal with established state changes to the total time taken for the diagnosis.

If a state change is suspected but not confirmed, the time taken to check whether a state change occurred has to be taken into account. This time is equal to $I*t_2$, where I is the number of iterations performed before the state change was suspected (i.e the number of tests performed so far) and t_2 is the average time taken to perform a test. The time $I*t_2$ thus corresponds to the time it takes to repeat all the tests. The actual time added to the total diagnostic time by the state suspicions depends mainly on the state suspicion policy and the stage in the diagnosis at which the state change suspicions occur. It is difficult to exactly quantify this time, but two remarks can be made:

- The difference between the mean time between failures and the average time to perform

a diagnosis (ignoring state change suspicions) must be sufficient to compensate for several state change suspicions.

- Dealing with state change suspicions does not involve the two most expensive operations, i.e test selection and candidate combination.

6.3.9. Performance-enhancing heuristics

The performance of the algorithm can be improved in several ways:

- The maximum number of elements that can be simultaneously faulty can be limited; this will reduce the growth in the number of suspect components and thus affect both the test evaluation cost and the minimal candidate generation cost.

- The maximum number of simultaneous sensor faults can be restricted to an even smaller number since they are normally less likely to occur. This will have the same effect as the previous improvement.

- The tests can be evaluated against the most probable suspects only. This will reduce the test evaluation cost.

- The test evaluation could be limited to tests of a certain size only. For example, if there are 12 suspects of equal probability, the theoretically optimal test would be of size 6 and thus the evaluation algorithm could be limited to tests whose size is between 3 and 9 for example. This again reduces the test evaluation cost.

- For large systems, the performance of the proposed troubleshooting model can be enhanced by a hierarchical decomposition of the system under observation. Then the troubleshooting algorithm will be applied to only one hierarchy layer at a time; thus it will have to deal with only a fraction of the total number of components and a fraction of the total number of tests, with a corresponding improvement in performance.

6.4. Computer Networks: a brief introduction

This section briefly introduces computer networks which is the domain chosen as the preliminary testing ground for the proposed diagnostic shell. The following description of computer networks, based on [Tane81], concentrates on the aspects of computer networks which are relevant to fault diagnosis in general and the proposed diagnostic model

in particular. Computer networks have several dynamic characteristics; changing traffic patterns, changing queues etc, but the main interest here is their dynamic behaviour from a diagnostic point of view; new faults can appear at any time, and existing faults may disappear at any time (if they were intermittent faults).

6.4.1. Network definition

A computer network is a dynamic system whose function is to transmit data between and allow access to geographically separated computers. The goal is to make data and other resources available to anyone on the network regardless of their physical location and to provide alternative services when faults occur.

6.4.2. Network structure

The network structure encompasses the types of components found in a network, together with the way they are interlinked. Most networks have a collection of machines (hosts) for running user (application) programs. Each host will offer a string of application programs (services) to the users. The hosts are connected by a communication subnet whose function is to carry information between the hosts. In nearly all networks, this subnet consists of 2 basic components: switching elements (also known as gateways or bridges) and transmission lines (circuits, channels etc). Switching elements normally have several alternative connections and can actively make decisions as to which particular connection to use to send information. Transmission lines are normally completely passive and simply transmit data from one side of the line to the other.

6.4.3. Network architecture

To simplify the communication between network components, most networks are organised as a series of layers. Each layer provides services to the layers above it and uses the services of the layers below it. In a communication between two hosts, peer layers communicate using a set of rules called protocol. A protocol is defined for each layer of the network. The standardised seven layer model [ISO84] is depicted in Figure 6.1.

The data do not travel directly between peer layers. Instead, each layer passes data and control information to the layer immediately below it, until the lowest layer is reached. At the lowest layer there is physical communication between the two sides (as opposed to virtual communication between peer layers).

Each layer has a certain function. Typical functions are:

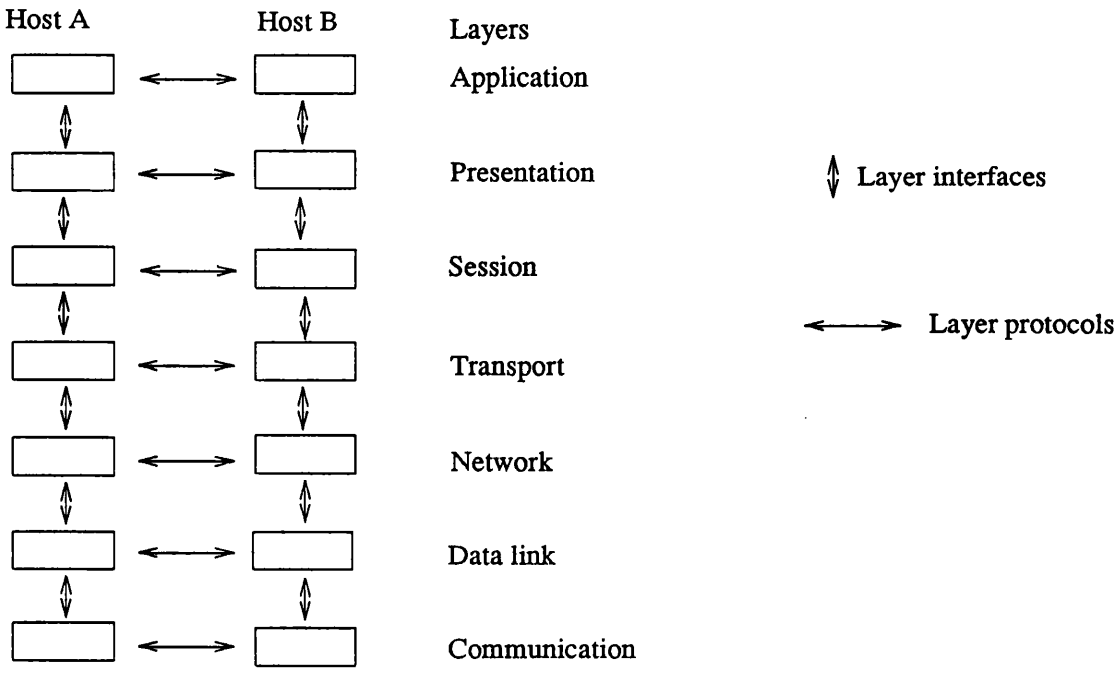


FIGURE 6.1 Layers, protocols and interfaces

- transmitting the data;
- making sure that the data are error-free;
- performing flow control between the two ends;
- opening and closing calls;
- authenticating users;
- routeing the data.

The routeing functions are normally performed at two levels [ISO84]; the application layer (corresponding approximately to routeing by the services) and the network level (corresponding approximately to routeing by the hosts and the switches).

6.4.4. Network topologies

There are several basic topologies for computer networks:

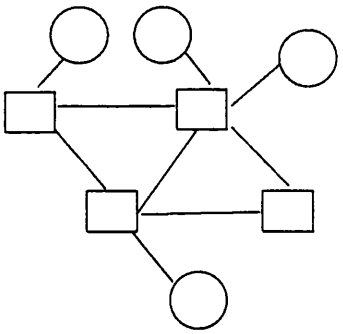
- **Point-to-point networks** consist of cables or leased lines that connect pairs of switching elements. The hosts are usually accessed via the gateways.

- In the **Star topology**, all the hosts are linked to a common central switch.

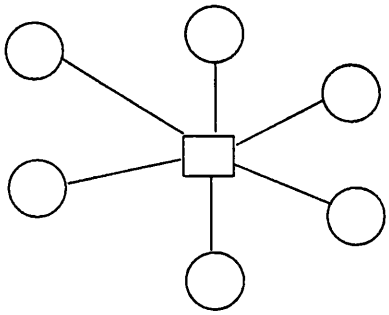
- In the **Ring topology**, all the hosts are linked in a circular fashion.

- In the **Bus topology**, the network itself is just a passive transmission medium. All the hosts are attached directly to the medium (or bus). The data is transmitted along the medium and can be received by all stations.

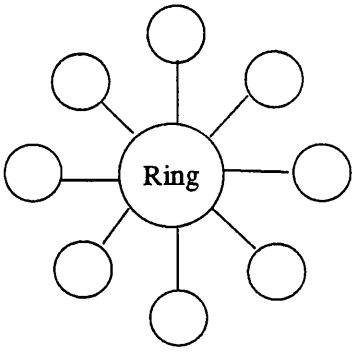
Networks will typically be more complex, consisting of several basic type subnetworks. The different network topologies are depicted in Figure 6.2.



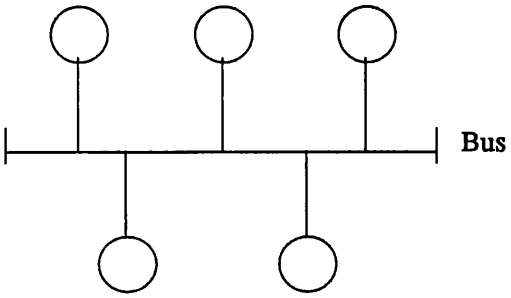
Point-to-point network



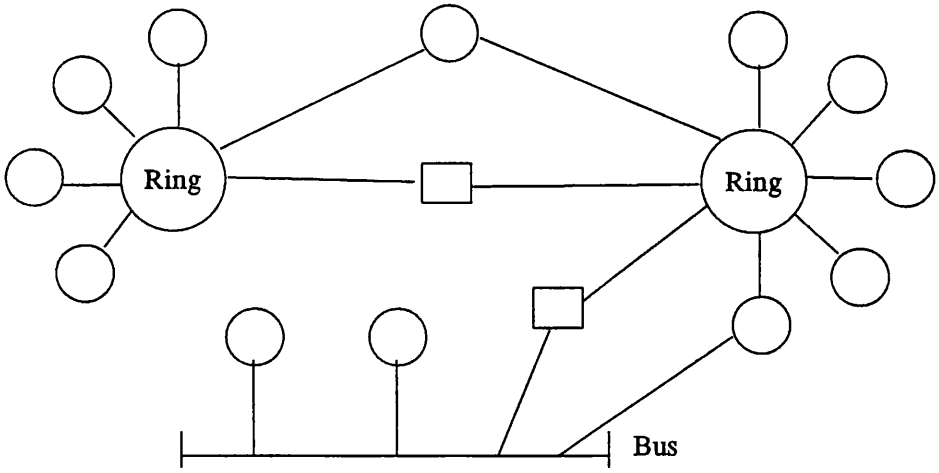
Star topology



Ring topology



Bus topology



Complex topology

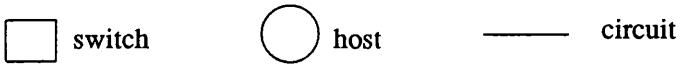


FIGURE 6.2 Computer network topologies

6.4.5. Routeing

The routeing algorithm determines the path that data are going to take between the source and the destination. The routeing behaviour has two aspects:

- the direct aspect: a network component is directly linked to n other components and can only send data to those n components. This direct aspect of routeing is implicit in the network topology;
- the indirect aspect: data to be sent to non-adjacent components have to be sent via the adjacent components. The goal is to determine how to guide information from its source to its destination. Thus the function of the routeing algorithms is to determine which route to choose when there are several possible alternatives. In other words, routeing algorithms can be seen as a set of routeing constraints imposed on top of the implicitly available routes. In terms of network structure, routeing decisions can be made by switches, hosts, services and subnetworks.

Fixed routeing: In fixed routeing, each routeing agent will always send the data for the same destination via the same direct link, even when alternative direct links are available.

Static routeing: In static routeing, a routeing agent can theoretically send data to the same destination using different direct links. However, at any stage, only one of those direct links will be used. In this type of routeing, changes in the routeing algorithms are performed only infrequently and (often) manually.

Centralised adaptive routeing: A centralised adaptive routeing mechanism depends on a routeing control centre. Periodically the routeing agents send status information (data patterns, delays etc) to the routeing control centre which then computes the new optimal routes for each routeing agent and forwards the relevant information to each one of them.

Distributed adaptive routeing: In distributed routeing, the routeing agents themselves exchange status information, and update their routeing tables on the basis of information obtained from other routeing agents.

6.4.6. Network testing tools

This section briefly surveys some typical network on-line testing tools. These are the tools which would typically be asked to provide information about the status of the network.

The probe type tools: These operate by actively trying to reach a network component and asking it to respond. A failure to receive a response will normally mean that that component is down or that it is unreachable (because some other component located between the source and the destination is down). Frequently, the probe can be forced to take a particular route to the destination, thus yielding more information about the network.

Threshold reports: Normally a component's behaviour should conform to certain acceptable standards. Components can thus be monitored to see if their behaviour is acceptable. For example a transmission link may have an acceptable error rate (say 10%) and an error rate above 10% could be treated as a symptom. Other thresholds can exist on the amount of traffic, data queue sizes, delays etc.

Event reports: Many networks have a mechanism for reporting unusual events. For example a connection between two points that breaks (as opposed to being closed normally) could be reported. Such events may be symptoms of faults. Similarly reports of normal events may be used to decide that selected components are working.

User complaints: Since the network is used by human beings, it is quite common for a user to be the first one to notice that something is wrong, because he is not able to perform some action. This information should be used as symptom information.

6.4.7. Important network characteristics

Computer networks have two characteristics typical of complex dynamic systems.

Testing tool characteristics: Most of the tools mentioned are part of the system under observation and use the system under observation to transmit their findings. Therefore, from a fault management point of view, the testing tools may go wrong and thus the data received may be wrong.

Fault characteristics: The basic fault principle in computer networks is that any component can go wrong at any time (before, during or after the current diagnosis session), and that faulty behaviour by a component may disappear at any time (if the fault was intermittent).

6.5. Deep knowledge translation of routeing failures into suspect components

This section describes an implementation of a deep knowledge algorithm for connection based networks, which from the description of the network structure (topology) and the network routeing behaviour and given symptoms related to failed routes, deduces the

components which could have caused the symptom to appear.

The routeing behaviour in computer networks is different from the behaviour of other systems that have been used to illustrate deep knowledge based reasoning. Typically, these other systems have been in the field of electronic troubleshooting of digital circuits [Davi84] , [Klee87] , [Gene84] , [Shir83] where the behaviour of each individual component can be seen in terms of the black box approach; several inputs, a transformation (characterising the behaviour) and an output as shown in Figure 6.3.

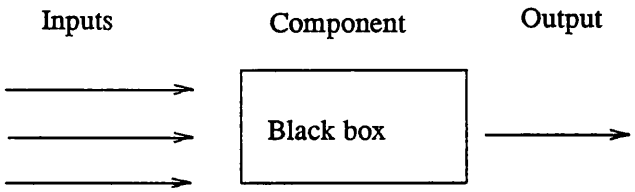


FIGURE 6.3 Black box behaviour

As will be seen later, the routeing behaviour does not necessarily produce a single output. Furthermore, routeing by an element entails knowledge about other elements which is incompatible with the black box definition. This difference makes the use of a deep knowledge algorithm particularly interesting.

6.5.1. Symptom information and routeing

Typically symptom information offered by the routeing entities are of the type: an attempted connection between A and B failed or an existing connection broke down. For example a user may have tried to open a connection to a remote machine or a probe may have attempted to connect to a remote element and failed. These problems are typical of the problems encountered in connection based computer networks [Tane81]. This information may already be offered to the fault manager by the routeing entities but:

- not all networks provide that information;

- the information may not be able to reach the fault manager or may be corrupted en route; therefore the fault manager will always need this knowledge as a back-up to the information provided by the network.

Two separate aspects have to be described in order to characterise a system; its structure and its behaviour [Davi84]. The structure is the information about the modules that form

the system; it contains mainly information about their types and their interconnection. The behaviour is the information about the action performed by each module.

6.5.2. The statics: Describing the network structure

The first decision to be made at this stage is as to which modules will appear in the description of the network's structure. A computer network can be made of many different components which in turn are made of many different subcomponents. For example a computer network may be seen as made of subnetworks, bridges/gateways (linking the subnetworks) and hosts. The hosts could be decomposed into Hardware and Software. The Hardware could then be decomposed into Input/Output, Memory and CPU. Memory could then further be decomposed into Storage devices and interfaces and so on. This decomposition is shown in Figure 6.4.

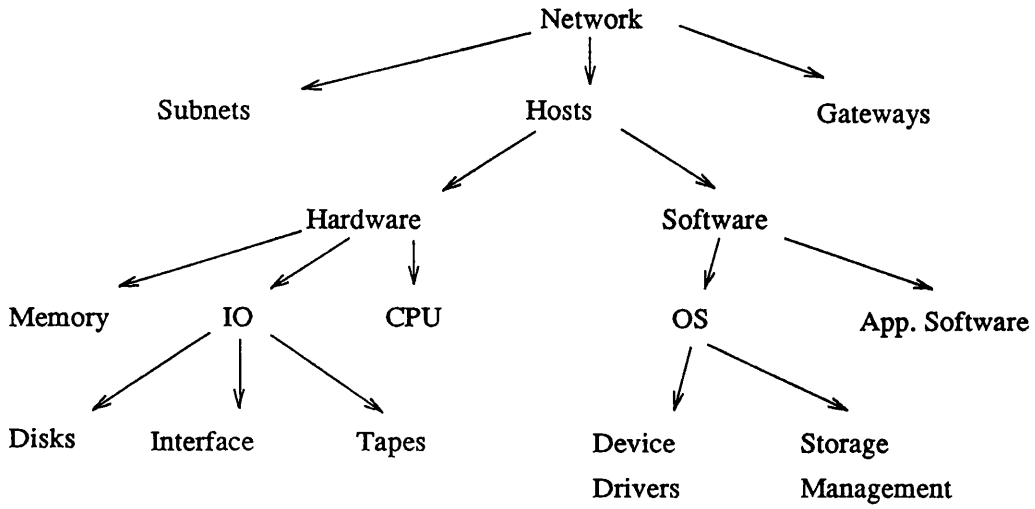


FIGURE 6.4 Hierarchical decomposition of a system

The following description will be restricted to the top level of this hierarchy which corresponds to fault isolation in computer networks. This description could then easily be extended to cover the hierarchical decomposition using for example frame-based hierarchical description languages.

In order to capture the behaviour of the routeing algorithms, the (top level) description should contain all the elements involved in routeing. For this purpose the network will be modeled using four types of entities: hosts, switches (bridges/ gateways), circuits/

subnetworks and services. The reasons for this choice are twofold:

- first, the location of all the routeing decisions made in a network can be simulated using these components;
- second, all the routeing decisions as to where the data are going are normally made about these components.

Routeing decisions in a network are normally done at two layers in the protocol hierarchy: the layer 3 (the network layer) and the layer 7 (application layer) as described in section 6.2.3. The host entity will make layer 3 decisions and the service entity will make the layer 7 decisions. Gateways also make layer 3 routeing decisions. Circuits do not normally make routeing decisions; they just transport data from one of their ends to the other. However subnetworks may make routeing decisions; for example in a Cambridge Ring topology, every ring has a name server whose function is to keep track of all the services available and decide how to reach them [Need82], which is also a routeing function.

Similarly routeing decisions are made about these 4 entities (hosts, gateways, subnetworks and services); in other words routeing elements and algorithms will know about these components and how to reach them. For example a service may know that to reach a different service, it has to send its data via a particular subnetwork. Or a gateway may know that another gateway is reachable via a particular subnetwork.

Thus all the routeing decisions can be modeled using the four entities service, host, gateway and subnetwork.

The structure description allows three types of relationships:

- the offers relationship which links a service entity and the host entity on which it is offered;
- the on relationship which links the host entity with the entity on which it is located;
- the bridgelink relationship which links bridges / switches / gateways entities to the adjacent circuits or subnetworks.

In Prolog (which was used for the implementation), the three relationships are:

offers(Service,Host).

on(Host,Subnetwork).

bridgelink(Bridge,Subnetwork).

Normally a service is always offered by a host, a host can be on a subnet or on a switch, and a bridge is linked to a circuit or a subnetwork. For example the network of Figure 6.5 would be mapped as follows:

bridgelink(B1,R1).	on(H1,R1).	on(H4,R2).
bridgelink(B1,R2).	on(H2,R1).	on(H4,E1).
bridgelink(B2,R2).	on(H2,R2).	on(H5,E1).
bridgelink(B2,E1).	on(H3,R2).	

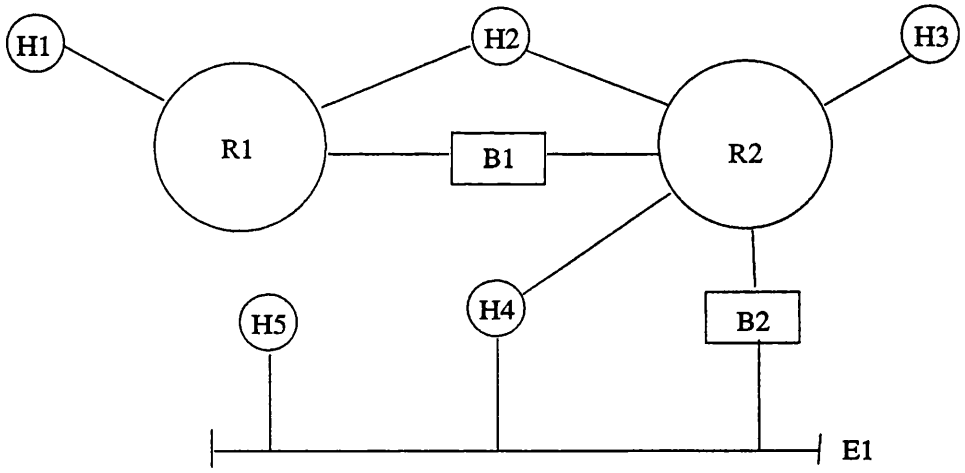


FIGURE 6.5 Computer network

6.5.3. The dynamics: Describing the routeing behaviour

Describing the routeing constraints

One particular type of behaviour is already implicit in the description of the network; data can travel bidirectionally between adjacent entities. Therefore a relationship between two entities implies the possibility of data flow between the two.

A second type of routeing behaviour is when a routeing entity knows that all the traffic to one particular entity has to go via another entity. This routeing behaviour is built on top of the first one and takes precedence over the first one. For example, given the simple topology in Figure 6.6 the implicit routeing algorithm would allow two routes between B1 and H1; [B1,L1,B2,H1] and [B1,L2,B2,H1]. If however, B1 has the additional constraint that all the traffic to H1 has to be sent via L1, then the only legal route is [B1,L1,B2,H1].

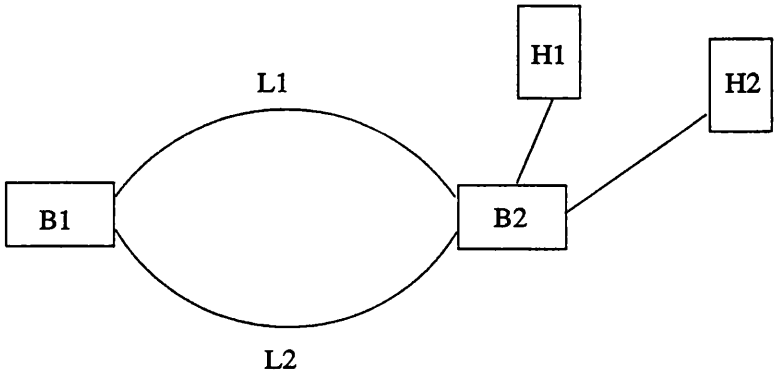


FIGURE 6.6 Simple network topology

This higher level routeing algorithm will be described as a system of constraints; a constraint takes the form: to go from entity A to entity B we have to go via entities C, D etc. In Prolog this is a term of the form: constraint(A-B,[List]).

Two additional constraints are applied:

- No route between two elements can go via a third element more than once; this blocks the infinite loops that would be created otherwise; furthermore once an element is included in a route, if that route becomes suspect, there is no point in having an element in the route more than once as its membership of the route makes it suspect already.

- Data can go to a host only if that host is the final destination or offers the service that is the final destination. If this constraint was not imposed, a host that is on two subnetworks (such as H2 or H4 in Figure 6.5) would also be performing the function of bridge, i.e would be transferring data from one subnetwork to another, which is not normally a host's function.

Generating the legal routes

The next stage is to generate all the legal routes between any two points A and B, given the information about the entities in the system, and the routeing constraints. This problem is akin to the general common problem of finding a path between two points in a connected graph, where the nodes of the graph are the services, hosts, bridges and subnetworks, and the edges are the links between those entities. Specific aspects of the routeing problem are that:

- all the legal paths between two points are wanted, rather than just one;
- a host is only acceptable as the next hop in the route if it is the second hop, the last hop or the last but one hop;
- there are additional routeing constraints.

Without these specific aspects, any simple graph walking algorithm will do.

To find all the legal paths using Prolog is very easy; backtracking is enforced every time a solution is reached until no new solution can be found.

Dealing with the difference between hosts and bridges is also quite easy. The route searching algorithm can easily eliminate the next hop hosts which do not conform to the acceptability definition for hosts.

As seen before a routeing constraint is of the form: to go from A to B, go via [C,D,...]. This is equivalent to breaking the problem into subproblems:

go(A,C), go(C,D), go(D,...), go(...,B)

where all the subproblems have to be solved *(and* type problem)

If there is no constraint, the next hop can be any of the adjacent entities. For example to go from A to B (assuming that C,D and E are the adjacent entities to A), the problem can be broken into subproblems:

{go(A,C) and (C,B)} or {go(A,D) and (D,B)} or {go(A,E) and (E,B)}

where it is sufficient to solve only one of the problems *(or* type problem).

The fact that there is a combination of *and* and *or* type problems suggests an AND/OR tree type solution for the route finding algorithm. In an AND/OR tree, the nodes of the tree correspond to the problems to be solved. The root of the tree is the initial problem (in this case to find the paths from A to B), and the following levels are the corresponding subproblems. An example is depicted in Figure 6.7. A curved arc denotes an *and* type relationship, meaning all the subtrees have to be solved. Otherwise, only one of the

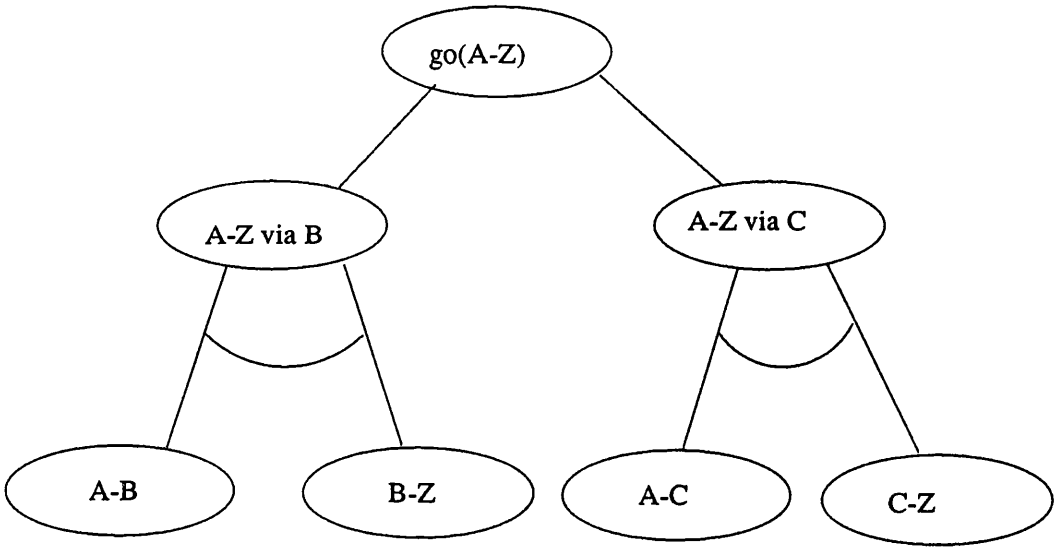


FIGURE 6.7 AND/OR problem decomposition

subtrees has to be solved. A goal node is a trivial subproblem of the type go(B-B). Each solution is a tree T defined recursively as follows:

- The initial problem P is the root of T;
- If P was decomposed into OR subtrees, then one of its successor subproblems together with its own solution tree is in T.
- If P was decomposed into AND subtrees, then all of its successor subproblems together with their solution trees are in T.

6.5.4. Using the network structure and behaviour description to generate suspects

The description of a network and of its routeing constraints allows the generation of all the legal routes between any two points of the network. This makes it possible to generate suspect elements for any information of the type: "a connection between points A and B failed". This includes information from users who cannot reach a machine, from monitors which attempt to bounce data off some other element (probes), broken connections and the like. Some systems may already have the routeing information when a failure of that type occurs but:

- the fact that the system has the information does not mean the fault manager has the

information. That information is normally held by the routeing entities;

- if the routeing algorithm does make that information available to the fault manager, it is not sure that the information will be able to reach it on time (since that information may come from several sources and there may be unpredictable delays and intermediate failures);

- not all systems provide that information;

- the fault manager should always be able whenever possible to back-up information it receives with deep knowledge mechanisms for obtaining that information.

If the routeing algorithm tries only one route between two points, then a union operation will give the initial list of suspects. If the routeing algorithm tries all the routes before declaring that it can't get through, then an intersection operation will give the initial list of suspects.

After the initial list of suspects has been established, additional tests will be performed to try and determine the fault. These additional tests will normally use routes that have been forced (i.e the route used by the test is known). For example loopback tests [Down77] are normally performed over a specified route. This is because only such specific information makes it possible to reduce the list of suspects. These tests can also be described using this methodology; the test can be seen as a service and the specified route a routeing constraint composed of several elements.

The algorithm described does not take into account the likelihood of the individual routes; it can be extended by giving costs to the connections of the graph if such information exists (for example if a subnetwork is more likely to use bridge1 than bridge2). This information could then be used to attach probabilities to the suspects. However it seems that such information is a priory difficult to get, since routeing preferences will depend on instantaneous data like traffic patterns, data queues and the like.

6.5.5. Illustrative runs

This section gives some very simple examples of fault isolation in a hypothetical network. The aim of the section is to illustrate the qualitatively different aspects of the fault model. Thus the 6 examples correspond to the following fault patterns:

- Example 1 shows a diagnostic session with a single fault in the system, ended when there are no more good tests to perform.

- Example 2 shows a diagnostic session with a single fault in the system, when one of the components becomes very probable.

- Example 3 shows a diagnostic session with a single fault in the system, where a state change suspicion (proved to be wrong) occurs.

- Example 4 shows a diagnostic session with a multiple fault in the system.

- Example 5 shows a diagnostic session with a sensor fault in the system.

- Example 6 shows a diagnostic session where a state change has occurred.

Assumptions

- The examples make the assumption that there is at most one test fault in the system.

- The sensors are assumed to communicate with the fault manager outside the network, using secure links.

Furthermore, the system stops when one of the suspects has a probability of failure greater than 0.99. State changes are suspected when the probability of the most probable candidate has been reduced or when all suspects imply a sensor fault.

The hypothetical network

The simple network used for these examples is shown in Figure 6.8. It is not dissimilar to a configuration that once existed at UCL. The main characteristics are as follows:

- t0 is a user who tries and fails to connect to u44e. This attempt may go either via the universe ring or via the service ring. When it fails, both routes become suspect.

- t1 is a test probing bridge1, via u44e and universe.

- t2 probes u44d via the universe/bridge1/microlab route.

- t3 probes bridge2 via universe.

- t4 probes u44a via the universe/bridge2/service route.

- t5 probes u44b via the universe/bridge2/service route.

- t6 probes u44c via the universe/bridge2/service route.

- t7 probes bridge3 via service.

- t8 probes u44d via the service/bridge3/microlab route.

- t9 probes bridge2 via service.

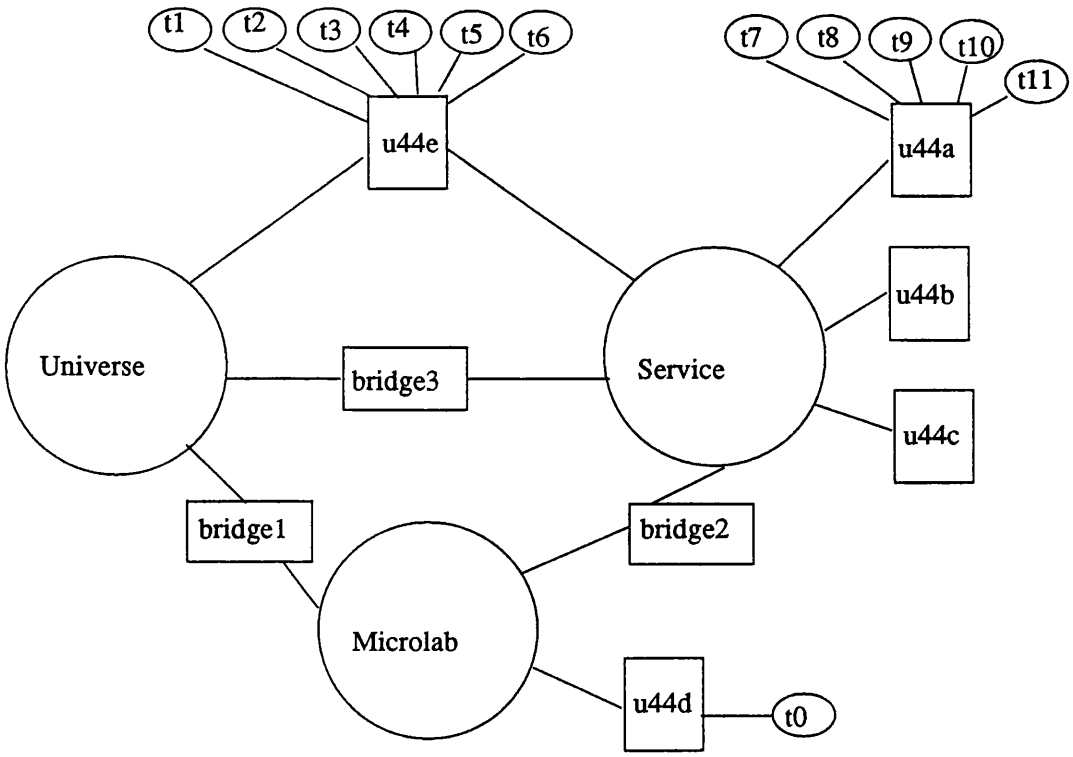


FIGURE 6.8 The simulated network

- t10 probes u44e via the service/bridge2/universe route.
- t11 probes u44b via service.

In the examples, answers to the fault manager are typed in Italics and comments are delimited by /*..*/. The answer noroute(A-B) means that the connection between A and B failed, and route(A-B) means that the connection between A and B succeeded.

Example 1

All test failure probabilities are 0.001. The other probabilities of failure are 0.01 apart from universe which is 0.1.

/* Beginning of diagnosis */

Input Test Id: *t0* **Input Test Result:** *noroute(t0-u44e)*

Current suspects: [([universe], .625), ([service], 0.0625), ([bridge3], 0.0625), ([u44e], 0.0625), ([bridge1], 0.0625), ([microlab], 0.0625), ([u44d], 0.0625)]

Perform test t3: *noroute(t3-bridge2)* /* new suspects t3, u44e, universe, bridge2 */

Current suspects: [([universe], .90449), ([u44e], 0.09044), ([service,bridge2], 0.0009), ([bridge3,bridge2], 0.0009), ([bridge1,bridge2], 0.0009), ([microlab,bridge2], 0.0009), ([u44d,bridge2], 0.0009), ([t0,bridge2], 9.04e-05), ([service,t3], 9.04e-05), ([bridge3,t3], 9.04e-05), ([bridge1,t3], 9.04e-05), ([microlab,t3], 9.04e-05), ([u44d,t3], 9.04e-05)]

No more good tests

/* Because no test can distinguish between the two most probable suspects*/

Trying to confirm the most probable suspect

Perform test t1: *noroute(t1-bridge1)* /* New suspects t1, u44e, universe, bridge1 */

Current suspects: [([universe], .90818), ([u44e], 0.090818), ([bridge1,bridge2], 0.00090), ([bridge1,t3], 9.0818e-05), ([service,bridge2,t1], 9.08e-07), ([bridge3,bridge2,t1], 9.08e-07), ([microlab,bridge2,t1], 9.08e-07), ([u44d,bridge2,t1], 9.08e-07)]

The most probable fault is [universe] with probability .90818

The other suspects were: [([u44e], 0.09081), ([bridge1,bridge2], 0.0009), ([bridge1,t3], 9.08e-05), ([service,bridge2,t1], 9.08e-07), ([bridge3,bridge2,t1], 9.08e-07), ([microlab,bridge2,t1], 9.08e-07), ([u44d,bridge2,t1], 9.08e-07)]

Example 2

All test failure probabilities are 0.001. The other probabilities of failure are 0.01 apart from bridge1 which is 0.1.

Input Test Id: *t0* **Input Test Result:** *noroute(t0-u44e)*

Current suspects: [([bridge1], .625), ([service], 0.0625), ([bridge3], 0.0625), ([u44e], 0.0625), ([universe], 0.0625), ([microlab], 0.0625), ([u44d], 0.0625)]

Perform test t1: *noroute(t1-bridge1)* /* New suspects t1, u44e, universe, bridge1 */

Current suspects: [([bridge1], .83305), ([u44e], 0.0833), ([universe], 0.0833), ([service,t1], 8.33e-05), ([bridge3,t1], 8.33e-05), ([microlab,t1], 8.33e-05), ([u44d,t1], 8.33e-05)]

Perform test t3: *route(t3-bridge2)* /* Working elements t3, u44e, universe, bridge2 */

Current suspects: [([bridge1], .9994), ([service,t1], 9.99e-05), ([bridge3,t1], 9.99e-05), ([microlab,t1], 9.99e-05), ([u44d,t1], 9.99e-05), ([u44e,t3], 9.99e-05), ([universe,t3], 9.99e-05)]

The most probable fault is: [bridge1] with probability .9996

The other suspects were: [([microlab,t1], 9.99e-05), ([u44d,t1], 9.99e-05), ([u44e,t3], 9.99e-05), ([universe,t3], 9.99e-05)]

Example 3

All test failure probabilities are 0.001. The other probabilities of failure are 0.01 apart from bridge1 which is 0.1.

Input Test Id: *t0* **Input Test Result:** *noroute(t0-u44e)*

Current suspects: [([bridge1], .625), ([service], 0.0625), ([bridge3], 0.0625), ([u44e], 0.0625), ([universe], 0.0625), ([microlab], 0.0625), ([u44d], 0.0625)]

Perform test t1: *route(t1-bridge1)* /*Working elements t1, u44e, universe, bridge1*/

Current suspects: [([service], .24319), ([bridge3], .24319), ([microlab], .24319), ([u44d], .24319), ([t0], 0.02431), ([bridge1,t1], 0.00024), ([u44e,t1], 0.00024), ([universe,t1], 0.00024)]

Suspecting a state change /*Reduction in the probability of the most probable suspect*/

Please repeat all tests sequentially. Write "same." if they all repeat their findings, else give the test number of the first test to change its findings: *same*

Current suspects: [([service], .24319), ([bridge3], .24319), ([microlab], .24319), ([u44d], .24319), ([t0], 0.02431), ([bridge1,t1], 0.00243), ([u44e,t1], 0.00024), ([universe,t1], 0.00024)]

Perform test t2: *route(t2-u44d)*

/* Working elements t2, u44e, universe, bridge1, microlab, u44d */

Current suspects: [([service], .47573), ([bridge3], .47573), ([t0], 0.04757), ([microlab,t2], 0.00047), ([u44d,t2], 0.00047)]

Perform test t11: *route(t11-u44b)* /* Working elements t11, u44a, service, u44b */

Current suspects: [([bridge3], .90661), ([t0], 0.09066), ([microlab,t2], 0.00090), ([u44d,t2], 0.00090), ([service,t11], 0.00090)]

Perform test t7: *noroute(t7-bridge3)* /* New suspects t7, u44a, service, bridge3 */

Current suspects: [([bridge3], .999), ([service,t11], 0.00099)]

The most probable fault is: [bridge3] with probability .999

The other suspects were: [([service,t11],0.00099)

Example 4

All test failure probabilities are 0.001. The other probabilities of failure are 0.01 apart from bridge3 which is 0.1.

Input Test Id: *t0* **Input Test Result:** *noroute(t0-u44e)*

Current suspects: [([bridge3], .625), ([service], 0.0625), ([u44e], 0.0625), ([universe], 0.0625), ([bridge1], 0.0625), ([microlab], 0.0625), ([u44d], 0.0625)]

Perform test t7: *route(t2-u44d)*

/* Working components t2, u44e, universe, bridge1, microlab, u44d */

Current suspects: [([bridge3], .90049), ([service], 0.09004), ([t0], 0.009), ([u44e,t2], 9.004e-05), ([universe,t2], 9.004e-05), ([bridge1,t2], 9.004e-05), ([microlab,t2], 9.004e-05), ([u44d,t2], 9.004e-05)]

Perform test t11: *noroute(t11-u44b)* /* New suspects t11, u44a, service, u44b */

Current suspects: [([service], .82501), ([bridge3,u44b], 0.08250), ([bridge3,u44a], 0.08250), ([bridge3,t11], 0.00825), ([t0,u44b], 0.00082), ([t0,u44a], 0.00082), ([u44e,t2,u44b], 8.25e-06), ([universe,t2,u44b], 8.25e-06), ([bridge1,t2,u44b], 8.25e-06), ([microlab,t2,u44b], 8.25e-06), ([u44d,t2,u44b], 8.25e-06), ([u44e,t2,u44a], 8.25e-06), ([universe,t2,u44a], 8.25e-06), ([bridge1,t2,u44a], 8.25e-06), ([microlab,t2,u44a], 8.25e-06), ([u44d,t2,u44a], 8.25e-06)]

Suspecting a state change

Please repeat all tests sequentially. Write "same." if they all repeat their findings, else give the test number of the first test to change its findings: *same*

Current suspects: [([service], .82501), ([bridge3,u44b], 0.08250), ([bridge3,u44a], 0.08250), ([bridge3,t11], 0.00825), ([t0,u44b], 0.00082), ([t0,u44a], 0.00082), ([u44e,t2,u44b], 8.25e-06), ([universe,t2,u44b], 8.25e-06), ([bridge1,t2,u44b], 8.25e-06), ([microlab,t2,u44b], 8.25e-06), ([u44d,t2,u44b], 8.25e-06), ([u44e,t2,u44a], 8.25e-06), ([universe,t2,u44a], 8.25e-06), ([bridge1,t2,u44a], 8.25e-06), ([microlab,t2,u44a], 8.25e-06), ([u44d,t2,u44a], 8.25e-06)]

Perform test t6: *route(t6-u44c)*

/* Working elements t6, u44e, universe, bridge2, service, u44c */

Current suspects: [([bridge3,u44b], .46935), ([bridge3,u44a], .46935), ([bridge3,t11],

0.04693), ([t0,u44b], 0.00469), ([t0,u44a], 0.00469), ([service,t6], 0.00469), ([bridge1,t2,u44b], 4.69e-05), ([microlab,t2,u44b], 4.69e-05), ([u44d,t2,u44b], 4.69e-05), ([bridge1,t2,u44a], 4.69e-05), ([microlab,t2,u44a], 4.69e-05), ([u44d,t2,u44a], 4.69e-05)]

Suspecting a state change

Please repeat all tests sequentially. Write "same." if they all repeat their findings, else give the test number of the first test to change its findings. *same*

Current suspects: [([bridge3,u44b], .46935), ([bridge3,u44a], .46935), ([bridge3,t11], 0.04693), ([t0,u44b], 0.00469), ([t0,u44a], 0.00469), ([service,t6], 0.00469), ([bridge1,t2,u44b], 4.69e-05), ([microlab,t2,u44b], 4.69e-05), ([u44d,t2,u44b], 4.69e-05), ([bridge1,t2,u44a], 4.69e-05), ([microlab,t2,u44a], 4.69e-05), ([u44d,t2,u44a], 4.69e-05)]

Perform test t10: *route(t10-u44e)*

/* Working components t10, u44a, service, bridge2, universe, u44e */

Current suspects: [([bridge3,u44b], .89984), ([bridge3,t11], 0.08998), ([t0,u44b], 0.00899), ([bridge3,u44a,t10], 0.00089), ([bridge1,t2,u44b], 8.99e-05), ([microlab,t2,u44b], 8.99e-05), ([u44d,t2,u44b], 8.99e-05)]

No more good tests. Trying to confirm the most probable suspect.

Perform test t5: *noroute(t7-bridge3)* /* New suspects t7, u44a, service, bridge3 */

Current suspects: [([bridge3,u44b], .90826), ([bridge3,t11], 0.09082), ([bridge3,u44a,t10], 0.00091)]

The most probable fault is: [bridge3,u44b] with probability .90826

The other suspects were: [([bridge3,t11], 0.09082), ([bridge3,u44a,t10], 0.00091)]

Example 5

All test failure probabilities are 0.001. The other probabilities of failure are 0.01.

Input Test Id: *t0* **Input Test Result:** *noroute(t0-u44e)*

Current suspects: [([service], .14285), ([bridge3], .14285), ([u44e], .14285), ([universe], .14285), ([bridge1], .14285), ([microlab], .14285), ([u44d], .14285)]

Perform test t1: *route(t1-bridge1)* /*Working components t1, u44e, universe, bridge1*/

Current suspects: [([service], .24372), ([bridge3], .24372), ([microlab], .24372), ([u44d], .24372), ([t0], 0.02437), ([u44e,t1], 0.00024), ([universe,t1], 0.00024), ([bridge1,t1], 0.00024)]

Perform test t2: *route(t2-u44d)*

/* Working components t2, u44e, universe, bridge1, microlab, u44d */

Current suspects: [([service], .47573), ([bridge3], .47573), ([t0], 0.04757), ([microlab,t2], 0.00047), ([u44d,t2], 0.00047)]

Perform test t11: *route(t11-u44b)* /* Working components t11, u44a, service, u44b */

Current suspects: [([bridge3], .90661), ([t0], 0.09066), ([microlab,t2], 0.00091), ([u44d,t2], 0.00091), ([service,t11], 0.00091)]

Perform test t7: *route(t7-bridge3)* /* Working components t7, u44a, service, bridge3 */

Current suspects: [([t0], .97087), ([microlab,t2], 0.00971), ([u44d,t2], 0.00971), ([bridge3,t7], 0.00971)]

Suspecting a state change.

Please repeat all tests sequentially. Write "same." if they all repeat their findings, else give the test number of the first test to change its findings: *same*

Current suspects: [([t0], .97087), ([microlab,t2], 0.00971), ([u44d,t2], 0.00971), ([bridge3,t7], 0.00971)]

The most probable fault is: [t0] with probability .97087

The other suspects were: [([microlab,t2], 0.00971), ([u44d,t2], 0.00971), ([bridge3,t7], 0.00971)]

Example 6

All test failure probabilities are 0.001. The other probabilities of failure are 0.01.

Input Test Id: *t0* **Input Test Result:** *noroute(t0-u44e)*

Current suspects: [([service], .14285), ([bridge3], .14285), ([u44e], .14285), ([universe], .14285), ([bridge1], .14285), ([microlab], .14285), ([u44d], .14285)]

Perform test t1: *route(t1-bridge1)* /*Working components t1, u44e, universe, bridge1*/

Current suspects: [([service], .24372), ([bridge3], .24372), ([microlab], .24372), ([u44d], .24372), ([t0], 0.02437), ([u44e,t1], 0.00024), ([universe,t1], 0.00024), ([bridge1,t1], 0.00024)]

Perform test t2: *route(t2-u44d)*

/* Working components u44e, universe, bridge1, microlab, u44d */

Current suspects: [([service], .47573), ([bridge3], .47573), ([t0], 0.04757), ([microlab,t2], 0.00047), ([u44d,t2], 0.00047)]

Perform test t11: *route(t11-u44b)* /* Working components u44a, service, u44b */

Current suspects: [([bridge3], .90661), ([t0], 0.09066), ([microlab,t2], 0.00091), ([u44d,t2], 0.00091), ([service,t11], 0.00091)]

Perform test t7: *route(t7-bridge3)* /* Working components u44a, service, bridge3 */

Current suspects: [([t0], .97087), ([microlab,t2], 0.00971), ([u44d,t2], 0.00971), ([bridge3,t7], 0.00971)]

Suspecting a state change

Please repeat all tests sequentially. Write "same." if they all repeat their findings, else give the test number of the first test to change its findings: *t0*

All the symptoms have disappeared and the diagnosis is finished.

6.6. Future work on the validation of the proposed shell

In order to fully develop and validate a proposed artificial intelligence system, several milestones have to be satisfied:

1) A high-level architecture has to be developed, the major algorithms worked out and the two put together into a unified entity.

2) The performance of the system has to be evaluated in order to ensure that it can deal with the required problems and that it is worth building it.

3) The system has to be implemented (possibly in a fairly coarse way) in order to ensure that the high level concepts can be turned into a computational entity.

4) Simple runs with simulated data should be executed on the system in order to perform a preliminary validation of the system and ensure that the system is behaving a priory as expected. At this stage, all the main features of the system should be tested and the residual algorithmic and design problems should be eliminated.

5) The system has to be tested against real data; this may imply a rewrite of the software so that it can deal with real-time data.

6) The system has to be tested in the various environments to which it is supposed to be applicable in order to test its generality.

7) The final problems (such as software bugs) have to be eliminated and user- friendly interfaces should be added.

Steps 1) to 4) lead to the existence of a software research prototype on which a preliminary validation has been performed. The aim of this research prototype is to show that AI and expert system technology can be applied to the problem, to test ideas about problem definition, scoping and representation of the domain and establish that the model will work for a variety of cases [Wate85]. Steps 5) to 7) constitute subsequent refinements of the initial demonstration prototype; this leads to further validation and development of the proposed system and eventually may turn the system into a fully fledged field prototype or production model.

The system proposed in this thesis has undergone most of the preliminary development and validation stages:

- Chapters 2,3,4 and 5 developed the high-level architecture and the necessary algorithms.

- The system is fully implemented.

- The system has undergone an initial (mathematical) performance evaluation.

- Preliminary tests have been performed on all the types of faults that the system has to solve.

These categories suggest the direction which future validation and research work should follow.

From an implementation point of view, the system should be reimplemented in a more efficient way and/or a more efficient language such as C (the particular version of Prolog used is in fact only interpreted). This may require further research into the optimisation of the proposed model and methods such as parallel programming. The current implementation was sufficient for prototyping purposes; as seen above any further validation requires a more efficient implementation.

The performance evaluation should be enhanced in two major ways. First a model of the proposed algorithms based on queuing theory should be built. This would give a much more precise performance figures than the current mathematical model (which is at best approximate). Second, real performance figures should be obtained once the model is efficiently implemented and being tested on a real system.

From the testing point of view, the system should be tested with a much larger variety of symptoms and against a real system. That means that a particular, existing computer network should be selected, and knowledge sources developed to deal with all the symptoms produced by the sensors on that network. Only a prolonged run on such a network would finally show that the proposed system is definitely applicable to a real system. Additional validation runs should also be performed on fault dignosis in other complex dynamic systems (such as nuclear power stations for example) since the model should be applicable to any complex dynamic system.

In the longer term, several enhancements should also be performed on the architecture and algorithms:

- The proposed system should be expanded to deal with a hierarchical decomposition of the system under observation. This would have a beneficial effect on the performance of the system as a whole and also make it compatible with existing hierarchical system description languages.

- The ATMS should be expanded to incorporate a temporal logic. This would have the

advantage of providing the temporal reasoning as part of the high level architecture and thus facilitate the task of the problem solver. The proposed system built its own temporal reasoning inside the problem solver but fortunately the temporal issues were relatively simple. However it is not clear whether the inclusion of a complex temporal logic would make the performance of the ATMS unacceptable.

- The proposed high level blackboard/ATMS architecture could be further enhanced to provide a truly general problem solving environment. This would incorporate user interface modules, various tools for developing control modules, debugging tools etc. The proposed system developed only the aspects of the ATMS/blackboard architecture which were directly relevant to fault diagnosis in complex dynamic systems.

6.7. Summary

This Chapter described the preliminary validation performed on the proposed system and discussed the proposed further validation issues. The preliminary validation included an implemented research prototype, an initial performance validation and the testing of the main characteristics of the implemented system using simple, simulated data. The proposed further validation discussed the implementation, performance and testing steps needed to fully validate the system. Finally future work directions were discussed.

CHAPTER VII

CONCLUSION

7.1. Thesis summary

The main thesis of this dissertation is that a general fault diagnostic environment has to encompass all the relevant stages of the diagnostic cycle and not just the translation of symptoms into suspect components. At the beginning of the dissertation, four major goals were set, which have been achieved as follows:

The building of a high level architecture for fault diagnosis in complex dynamic systems

Chapter 2 analysed the requirements of fault diagnosis in complex dynamic systems relevant to the architecture design and developed a new high level architecture based on the coupling of a layered blackboard and an Assumption-based Truth Maintenance System, which fulfills all the architectural requirements.

The development of a general fault model for complex dynamic systems

Chapter 3 analysed the requirements of fault diagnosis in complex dynamic systems from the point of view of the fault model. It then proposed a new fault model which fulfills these requirements. The fault model has two aspects: a unified probabilistic methodology for dealing with single, multiple and sensor faults, and a temporal fault model aspect to deal with state changes, based on suspicion about state changes, confirmation of state changes and efficient recovery from state changes.

The development of a general testing strategy for complex dynamic systems

Chapter 4 argued for the use of algorithmic techniques for determining the next test to perform. It then developed a new algorithmic test selection mechanism, based on the concept of entropy, which requires only a list of the tests available and the a priory probabilities of failure of the system components. In this methodology, each test is evaluated according to the change in entropy (in the list of suspect candidates) it is expected to

cause. Some heuristic methods for reducing the cost of the methodology were also covered.

The mapping of the fault diagnosis and test generation algorithms to the high level architecture

Chapter 5 discussed the development of a fault diagnostic shell obtained by integrating the high level blackboard/ATMS architecture and the fault diagnosis and test generation algorithms. It covered the mapping of these algorithms onto the ATMS in an efficient way, the mapping of these algorithms onto the blackboard with a discussion of the required control modules and knowledge sources, the cooperation of the ATMS and the blackboard in solving a problem and a methodology for extending the model to deal with continuous diagnosis.

Chapter 6 discussed the preliminary validation performed on the proposed system and proposed further validation methods and directions for future work.

7.2. Critical evaluation

The approach to the research

For every new problem analysed in this dissertation, the general approach was to look at existing solutions to similar problems and try to improve them or adapt them to the new problem. Thus, the proposed architecture contains elements of existing architectures, the fault model is partly based on an existing minimal candidate algorithm and the test generation algorithm is based on the entropy measure which has already been used in similar contexts. The alternative research method is to try to generate a solution solely by looking at the problem space, without being influenced by existing methodologies. Both these approaches are valid. The advantage of the first approach and the reason it was used here is that modifications and extensions of existing tools are very likely to work in similar domains since the underlying tools have already been tested and shown to be working. The second approach will probably lead to a better solution for a problem which is not similar to any existing problem.

Generality of the proposed solution

The proposed solution is general in two distinct ways:

- It is applicable to the most general category of systems, namely the complex dynamic

systems. This generality is embodied in the lack of restrictive assumptions about the nature, number and timing of the faults and the chosen test generation algorithm. As such, it represents a step forward from the existing models which all make restrictive assumptions.

- It provides all the stages in the fault diagnostic cycle that can be generalized, not just one stage as most existing systems do.

The dissertation assumes that generality of the solution is a desirable property. As was discussed in the introduction, this assumption seems to be supported by the historical development of research in computational tools, but it cannot be proven right. The opposite view that generality has to be sacrificed for performance optimisation is also defensible but unprovable.

The high level architecture

The proposed blackboard/ATMS architecture is not only suitable for fault diagnosis, but also for any system that needs to perform opportunistic reasoning and reasoning in multiple contexts. The blackboard offers very flexible and modular data organisation and control mechanisms, while the ATMS offers an indexing to all the reasoning performed so far, a consistent approach to multiple context searching, and help for the blackboard control functions. This inherent complementarity between the blackboard and the ATMS yields a very general, efficient and elegant high level architecture.

The fault model

The main feature of the proposed fault model is the lack of restrictive assumptions about the nature of the fault. It allows single faults, multiple faults, sensor faults and intermittent faults (state changes), whereas existing fault models are restricted to a subset of these. The first three fault types are grouped and dealt with in a unified probabilistic approach, while the temporal reasoning is contained in a separate control loop.

The probabilistic approach to the fault model is not the only possible approach to modelling uncertainty. Fuzzy logic for instance offers a perfectly acceptable alternative, which has been used with success in other systems. Choosing between these two alternatives (and other uncertainty models) is not always clear-cut, as they all normally have a sound theoretical basis. In this case however, the probabilistic model has an inherent advantage over other uncertainty models; it can be extended to perform test generation using the entropy function, which is a theoretically sound way. For this reason, it was used in

preference to other uncertainty models.

At least two alternative approaches to the temporal reasoning problem are possible. The first is to try and develop a fault model which integrates the temporal reasoning and the reasoning about single, multiple and sensor faults. The difficulty with such an approach is the duality between sensor faults and state changes. A second solution is to incorporate a temporal logic into the ATMS. Such an approach would allow the problem solver to ask the ATMS questions about temporal consistency, but may be expensive in terms of performance overhead.

The reason the current fault model was chosen is that it incorporates a very simple temporal logic, whose sole purpose is to determine and maintain state consistency. Given the very limited amount of temporal information during fault diagnosis in a complex dynamic system, this temporal logic seemed sufficient for this particular problem. For systems where the amount of temporal information is bigger, this temporal logic is inadequate because it is too restricted.

The test selection algorithm

The test selection phase is an excellent example of the trade-off between performance on one side and adaptability/ flexibility/ generality on the other side. In artificial intelligence terms this trade off is between heuristic and algorithmic approaches. The proposed test selection mechanism based on the entropy function is definitely algorithmic. An algorithmic approach was chosen so that it could be incorporated into a general diagnostic tool. The satisfactory performance characteristics of the proposed test selection algorithm (independent of the system size) together with the general advantages of algorithmic mechanisms were the other deciding factors.

The possibility of making multiple test requests at the same time was not analysed. The reason is that it would have been incompatible with the chosen fault model, which requires sequential testing in order to perform temporal reasoning and in order to distinguish between sensor faults and state changes. Multiple test requests can be used in a static system, or even in a system where the sensors are completely reliable, but when both state changes and test faults can occur, keeping track of all the possible alternatives would soon lead to a combinatorial explosion.

The performance characteristics

General problem-solving tools, such as the one developed here, invariably have a worse performance than problem-specific tools. At the limit, the performance of the general tools becomes too slow to be acceptable. This dissertation did not attempt to establish the

absolute performance limits of the proposed environment. The reason is that this was not taken as a research issue; the main aim of the thesis was to develop a fault diagnostic shell for complex dynamic systems and to build an experimental research prototype to perform the preliminary validation of the proposed shell. This lack of absolute performance analysis is a drawback. However the general performance analysis of the system suggested that there is no a priori limit in the size of the system to which the fault diagnostic environment can be applied as the execution time of the proposed algorithms does not depend directly on the size of the system under observation. Furthermore, performance enhancing heuristics were developed for all the major stages in the diagnostic process. A full performance analysis would have to look into parallel computer architectures as well as sequential ones , which was beyond the scope of this dissertation. However, the two most expensive parts of the solution, namely the calculation of the maximal contexts and the evaluation of tests can both be done in a parallel fashion (as seen in Chapter 6), thus eliminating the major potential bottlenecks.

The dissertation has therefore provided several novel, general purpose contributions to the field of fault diagnosis in complex dynamic systems:

- a suitable architecture based on the coupling of a layered blackboard with an ATMS;

- a suitable general fault model to deal uniformly with single, multiple and sensor faults and to react and recover from state changes in the system;

- a fully adaptive test optimisation algorithm based on the entropy function, which requires only the list of tests available and the a priori probabilities of failure of the components.

These three elements must be present and must be given equal importance in any general fault diagnostic environment for complex dynamic systems. This dissertation is therefore a step towards the goal of automated diagnosis in complex dynamic systems.

REFERENCES

[Addi80.]
T. R. Addis, "Expert Systems: an Evolution in Information Retrieval," *ICL Technical Journal*, pp. 79-105, 1980.

[Alle83.]
J. F. Allen, "Maintaining Knowledge about Temporal Intervals," *Communications of the ACM*, vol. 26, no. 11, pp. 823-843, 1983.

[Balz80.]
R. Balzer, L. D. Erman, P. London, and C. Williams, "HEARSAY-III: A Domain-Independent Framework for Expert Systems," *Proceedings of the First Annual National Conference on Artificial Intelligence*, 1980.

[Bara86.]
J. Barac, "Reasoning About Sensors in Fault Diagnosis," *IFIP WG 6.4a Proceedings of the First Workshop on LAN Management*, pp. 29-33, London, 1986.

[Bigh87.]
J. Bigham, V. H. Khong, G.I Williamson, and S. G. King, *Inference Tools for Knowledge Based Systems in Industrial Application Areas*, 1987.

[Bobr77.]
D. G. Bobrow and T. Winograd, "An Overview of KRL, a Knowledge Representation Language," *Cognitive Science 1*, pp. 3-46, January 1977.

[Brac85.]
R. J. Brachman and J. G. Schmolze, "An Overview of the KL-ONE Knowledge Representation System," *Cognitive Science 9*, pp. 171-216, April 1985.

[Breu76.]
M. Breuer and A. Friedman, *Diagnosis and Reliable Design of Digital Systems*, Computer Science Press, 1976.

[Brow82.]
J. S. Brown, R. R. Burton, and J. de Kleer, "Pedagogical, Natural Language and Knowledge Engineering Techniques in SOPHIE I,II,III," in *Intelligent Tutoring Systems*, ed. D. Sleeman J.S. Brown, Academic Press, 1982.

[Cant83.]
R. R. Cantone, F. J. Pipitone, W. B. Lander, and M. P. Marrone, "Model-Based Probabilistic Reasoning For Electronic Troubleshooting," *Proceedings IJCAI-83*, pp. 207-211, Karlsruhe, 1983.

[Chan87.]
B. Chandrasekaran and W. F. Punch, "Data Validation During Diagnosis, a Step Beyond Traditional Sensor Validation," *Proceedings AAAI-87*, pp. 778-782, 1987.

[Ches84.]
D. Chester, D. Lamb, and P. Dhurjati, "Rule-Based Computer Alarm Analysis in Chemical Process Plants," *Proceedings of the Seventh Annual Conference on Computer Technology, MICRO-DELCON 84, IEEE*, pp. 22-29, 1984.

[Cohn85.]
A. G. Cohn, "Deep Knowledge Representation Techniques," *Expert Systems 85*, pp. 299-306. Brighton, 1985.

[Davi84.]
R. Davis, "Diagnostic Reasoning Based on Structure and Behaviour," *Artificial Intelligence*, vol. 24, pp. 347-410, 1984.

[Deit83.]
H. M. Deitel, *An Introduction to Operating Systems*, Addison Wesley, 1983.

[Down77.]
P. J. Down, *Fault Diagnosis in Data Communications*, NCC Publications, 1977.

[Doyl79.]
J. Doyle, "A Truth Maintenance System," *Artificial Intelligence*, vol. 12, pp. 231-272, 1979.

[Duda79.]
R. Duda, J. Gaschnig, and P. Hart, "Model Design in PROSPECTOR Consultant System for Mineral Exploration," in *Expert Systems in the Microelectronic Age*, ed. D. Michie, pp. 153-167, Edinburgh University Press, 1979.

[Enge88.]
R. Engelmore and T. Morgan, *Blackboard Systems*, Addison-Wesley, 1988.

[Erma80.]
L. D. Erman, F. Hayes-Roth, V. R. Lesser, and D. Raj Reddy, "The HEARSAY-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty," *ACM Computing Survey*, vol. 12, pp. 213-253, 1980.

[Erns69.]
G. Ernst and A. Newell, *GPS: A Case Study in Generality and Problem Solving*, Academic Press, 1969.

[Fike85.]
R. Fikes and T. Keller, "The Role of Frame-Based Representation in Reasoning," *Communications of the ACM*, vol. 28, no. 9, pp. 904-920, 1985.

[Forg81.]
C. L. Forgy, "OPS5 User's Manual," Report CMU-CS-81-135, Computer Science Dept., Carnegie-Mellon, Pittsburgh, 1981.

[Fox83.]
M. S. Fox, S. Lowenfeld, and P. Kleinosky, "Techniques for Sensor-Based Diagnosis," *Proceedings IJCAI-83*, pp. 158-163, Karlsruhe, 1983.

[Fuso83.]
A. Fusoaka, H. Seki, and K. Takahashi, "A Description and Reasoning of Plant Controllers in Temporal Logic," *IJCAI-83*, pp. 405-408, Karlsruhe, 1983.

[Geff87.]
H. Geffner and J. Pearl, "Distributed Diagnosis of Systems with Multiple Faults," *IEEE Conference on AI Applications*, pp. 156-162, 1987.

[Gene84.]
M. R. Genesereth, "The Use of Design Descriptions in Automated Diagnosis," *Artificial Intelligence*, vol. 24, pp. 411-436, 1984.

[Hams83.]
W. Hamscher, "Using Structural and Functional Information in Diagnostic Design," *Proceedings AAAI-83*, pp. 152-156, Washington, 1983.

[Haye85a.]
B. Hayes-Roth, "A Blackboard Architecture for Control," *Artificial Intelligence*, vol. 26, pp. 251-321, 1985.

[Haye85.]
F. Hayes-Roth, "Rule-Based Systems," *Communications of the ACM*, vol. 28, no. 9, pp. 921-932, September 1985.

[ISO84.]
ISO, *Information Processing Systems- Open Systems Interconnection- Basic Reference Model*, ISO 7498, 1984.

[John85.]
L Johnson and E. T. Keravnou, *Expert Systems Technology: a Guide*, Abacus Press, 1985.

[Kahn77.]
K. M. Kahn and A. G. Gorry, "Mechanising Temporal Knowledge," *Artificial Intelligence*, vol. 9, pp. 87-108, 1977.

[Karp72.]
R. Karp, "Reducibility among Combinatorial Problems," in *Complexity of Computer Computations*, ed. J. Thatcher, pp. 85-103, Plenum Press, 1972.

[Kell82.]
V. E. Kelly and L. I. Steinberg, "The CRITTER System: Analyzing Digital Circuits by Propagating Behaviours and Specifications," *Proceedings AAAI-82*, pp. 284-289, Pittsburg, 1982.

[Klee86.]
J. de Kleer, "Problem Solving with the ATMS," *Artificial Intelligence*, vol. 28, pp. 197-224, 1986.

[Klee86a.]
J. de Kleer, "An Assumption-Based TMS," *Artificial Intelligence*, vol. 28, pp. 127-162, 1986.

[Klee86b.]
J. de Kleer, "Extending the ATMS," *Artificial Intelligence*, vol. 28, pp. 163-196, 1986.

[Klee87.]
J. de Kleer and B. C. Williams, "Diagnosing Multiple Faults," *Artificial Intelligence*, vol. 32, pp. 97-130, 1987.

[Knig86.]
G. Knight, "A Database for Local Network Management," *IFIP WG 6.4a Proceedings of the First Workshop on LAN Management*, pp. 35-40, 1986.

[Kunz84.]
J. C. Kunz, T. P. Kehler, and M. D. Williams, "Applications Development Using a hybrid AI Development System," *The AI Magasine*, vol. 5, no. 3, Fall 1984.

[Laff84.]
T. J. Laffey, W. A. Perkins, and T. A. Nguyen, "Reasoning About Fault Diagnosis with LES," *IEEE Conference on AI Applications*, pp. 267-273, 1984.

[Lind80.]
R. K. Lindsay, E. A. Buchanan, E. A. Feigenbaum, and J. Lederberg, *Applications of Artificial Intelligence for Organic Chemistry: the DENDRAL Project*, McGraw-Hill, 1980.

[Long83.]
W. J. Long, "Reasoning about State from Causation and Time in a medical Domain," *Proceedings of the National Conference on Artificial Intelligence*, pp. 251-254, Washington, 1983.

[Mart77.]
J. Martin, *Computer Data-Base Organisation*, Prentice-Hall, 1977.

[Mart83.]
J. P. Martins and S. C. Shapiro, "Reasoning in Multiple Belief Spaces," *Proceedings IJCAI-83*, Karlsruhe, 1983.

[McAl80.]
D. McAllester, "An Outlook on Truth Maintenance," AI Memo 551, Artificial Intelligence Laboratory, MIT, 1980.

[McDe82.]
D. McDermott, "A Temporal Logic for Reasoning Processes and Plans," *Cognitive Science*, vol. 6, pp. 101-155, 1982.

[McDe83.]
D. McDermott, "Contexts and Data Dependencies: a Synthesis," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 5, no. 3, pp. 237-246, 1983.

[Mich83.]
R. S. Michalski and A. B. Baskin, "Integrating Multiple Knowledge Representations and Learning Capabilities in an Expert System: the ADVISE System," *Proceedings IJCAI-83*, pp. 256-258,

Karlsruhe, 1983.

[Need82.]
R. M. Needham and A. J. Herbert, *The Cambridge Distributed Computing System,* Addison-Wesley, 1982.

[Nels82.]
W. R. Nelson, "REACTOR: An Expert System for Diagnosis and Treatment of Nuclear Reactor Accidents," *Proceedings AAAI-82*, pp. 296-301, Pittsburg, 1982.

[Nii79.]
H. P. Nii and N. Aiello, "AGE: a Knowledge-based Program for Building Knowledge-based Programs," *Proceedings IJCAI-79*, pp. 209-219, Tokyo, 1979.

[Nii82.]
H. P. Nii, E. A. Feigenbaum, J. J. Anton, and A. J. Rockmore, "Signal-to-Symbol Transformation: HASP/SIAP Case Study," *AI Magasine*, vol. 3, no. 2, pp. 23-35, 1982.

[Nii86.]
H. P. Nii, "Blackboard Systems(Part 1)," *The AI Magasine*, pp. 38-53, 1986.

[Pauk76.]
S. G. Pauker, G. A. Gorry, J. P. Kassirer, and W. B. Schwartz, "Toward the Simulation of Clinical Cognition: Taking a Present Illness by Computer," *The American Journal of Medicine 60*, pp. 981-995, 1976.

[Peng86.]
Y. Peng and J. A. Reggia, "Plausibility of Diagnostic Hypotheses: the Nature of Simplicity," *Proceedings AAAI-86*, pp. 140-145, Philadelphia, 1986.

[Popl75.]
H. E. Pople, "Dialog: a Model of Diagnostic Logic for Internal Medicine," *Proceedings IJCAI-75*, pp. 848-855, 1975.

[Rebo81.]
R. Reboh, "Knowledge Engineering Techniques and Tools in the PROSPECTOR Environment," SRI Technical Note 243, Stanford Research Institute, Menlo Park, 1981.

[Redd82.]
Y. V. Reddy and M. S. Fox, "KBS: An Artificial Intelligence Approach to Flexible Simulation," Report CMU-RI-TR-82-1, Robotics Institute, Carnegie-Mellon University, Pittsburgh, 1982.

[Reit87.]
R. Reiter, "A Theory of Diagnosis from First Principles," *Artificial Intelligence*, vol. 32, pp. 57-96, 1987.

[Sevi83.]
R. E. Seviora, "Field Debugging of Telecommunication Switching Machines," *Globecom 83 (IEEE Global Telecommunication Conference)*, vol. Volume 3, pp. 1480-1485, 1983.

[Shan49.]
C. E. Shannon and Warren Weaver, *The Mathematical Theory of Communication*, University of Illinois Press, 1949.

[Shir83.]
M. H. Shirley and R. Davis, "Generating Distinguishing Tests Based on Hierarchical Models and Symptom Information," *Proceedings IEEE International Conference on Computer Design*, pp. 455-458, 1983.

[Shor76.]
E. H. Shortliffe, *Computer-based Medical Consultations,* American Elsevier, 1976.

[Shub82.]
H. Shubin and J. W. Ulrich, "IDT: an Intelligent Diagnostic Tool," *Proceedings AAAI-82*, pp. 290-295, Pittsburg, 1982.

[Slag71.]
J. R. Slagle and R. C. T. Lee, "Application of Game Tree Searching Techniques to Sequential Pattern Recognition," *Communications of the ACM*, vol. 14, pp. 103-110, 1971.

[Smit83.]
R. G. Smith and J. D. Baker, "The DIPMETER Advisor System," *Proceedings IJCAI-83*, pp. 122-129, Karlsruhe, 1983.

[Stef79.]
M. J. Stefik, "An Examination of a Frame-Structured Representation System," *Proceedings IJCAI-79*, pp. 845-852, Tokyo, 1979.

[Szol77.]
P. Szolovits, L. B. Hawkinson, and W. A. Martin, "An Overview of OWL, a Language for Knowledge Representation," MIT/LCS/TM-86, Laboratory of Computer Science, MIT, 1977.

[Tane81.]
A. S. Tanenbaum, *Computer Networks*, Prentice Hall, 1981.

[VanM79.]
W. VanMelle, "A Domain-Independent Production Rule System for Consultation Programs," *Proceedings IJCAI-79*, pp. 923-925, Tokyo, 1979.

[Veso83.]
G. T. Vesonder, S. J. Stolfo, J. E. Zielinski, F. D. Miller, and D. H. Copp, "ACE: an Expert System for Telephone Cable Maintenance," *Proceedings IJCAI-83*, pp. 116-121, Karlsruhe, 1983.

[Wate85.]
D. A. Waterman, *A Guide to Expert Systems*, Addison Wesley, 1985.

[Weis74.]
S. M. Weiss, "A System for Model-based Computer-aided diagnosis and therapy," PhD Thesis, Computers in Biomedicine, Dept. of Computer Science, Rutgers University, CBM-TR-27-Thesis, 1974.

[Weis79.]
S. M. Weiss and C. A. Kulikowski, "EXPERT: a System for Developing Consultation Models," *Proceedings IJCAI-79*, pp. 942-947, Tokyo, 1979.

[Weis83.]
S. M. Weiss and C. A. Kulikowski, *A Practical Guide to Designing Expert Systems*, Chapman and Hall, 1983.

[Will83.]
T. L. Williams, P. J. Orgren, and C. L. Smith, "Diagnosis of Multiple Faults in a Nationwide Communications Network," *Proceedings IJCAI-83*, pp. 179-181, Karlsruhe, 1983.

[Will87.]
G. I. Williamson, J. Bigham, J. W. Butler, and S. G. King, "Using a KBS in telecommunications 2," *Proceedings of Esprit Technical Week IV*, 1987.

[Zwas81.]
V. Zwass, *Introduction to Computer Science*, Barnes and Noble, 1981.

APPENDIX 1

ENTROPY CALCULATIONS

Calculating the prior probability of $[\overline{x_{bi}}, \overline{x_{uj}}:]$

This and the following section makes the assumption that the probability calculated for a minimal candidate is the sum of the probabilities of all the candidates that contain the components included in the minimal candidates. For example, if the system has 3 components a,b and c, the minimal candidate $[\overline{a}:]$ stands for the candidates: $[\overline{a}:bc]$, $[\overline{ab}:c]$, $[\overline{ac}:b]$ and $[\overline{abc}:]$ and the sum of the probabilities of those 4 elements is equal to the probability of the minimal candidate $[\overline{a}:]$, calculated as in section 3.4.6.4. This sum is in fact only an approximation as will be discussed in section 4.10. However, it makes it possible to avoid the combinatorial explosion that would occur in a complex system if all the suspects where represented.

When calculating the prior probability of $[\overline{x_{bi}}, \overline{x_{uj}}:]$, the prior probabilities of x_{bi} and x_{uj} are known. The prior probability of minimal candidate x_{uj} covers all the candidates containing the components included in x_{uj} as described in the assumption above.

Let NW be the set of elements not known to be working initially: NW = $\{nw_1, nw_2, .. , nw_{nw}\}$. Then the minimal candidate x_{uj} really stands for a set of candidates LS which comprises all the possible permutations of $nw_1 .. nw_{nw}$ together with x_{uj}. Within each permutation, each element nw_i can have two states; either $\overline{nw_i}$ (meaning element nw_i not working) or nw_i (meaning element nw_i working). In other words:

$$LS = \{[\overline{x_{uj}}, nw'_1, .. , nw'_{nw}] \}$$

where nw'_i is nw_i or $\overline{nw_i}$.

Let us suppose without loss of generality that $x_{bi} = nw_1$. Then for each possible permutation of $[\overline{x_{uj}}, nw'_2 .. nw'_{nw}]$, nw'_1 can be equal to nw_1 or $\overline{nw_1}$ Therefore in probability

terms, each $p_i * p(nw_1)$ has a corresponding $p_i * p(\overline{nw_1})$ (where p_i stands for the probability of a permutation $[\overline{x_{uj}}, nw'_2 .. nw'_{nw}]$). The sum of all the probabilities which include $p(\overline{nw_1})$ and which is the sum we are interested in is therefore:

$$\sum_{i=1}^{i=2^{(nw-1)}} p_i \, p(\overline{nw_1})$$

The sum of the other elements is :

$$\sum_{i=1}^{i=2^{(nw-1)}} p_i \, p(nw_1)$$

and the sum of those two expressions is of course $p(x_{yj})$. So:

$$\sum_{i=1}^{i=2^{(nw-1)}} p_i \, p(\overline{nw_1}) + \sum_{i=1}^{i=2^{(nw-1)}} p_i \, (1 - p(\overline{nw_1})) = p(x_{yj})$$

i.e

$$\sum_{i=1}^{i=2^{(nw-1)}} p_i = p(x_{yj})$$

and

$$\sum_{i=1}^{i=2^{(nw-1)}} p_i \, p(\overline{nw_1}) = p(\overline{nw_1}) \, p(x_{yj})$$

i.e

$$p'([\overline{x_{bi}}, \overline{x_{ui}}]) = p(x_{bi}) \, p(x_{ui})$$

The same also applies to the elements obtained by correlating C_i and U_{ip}.

Calculating the entropy of a successful test

If the test t_i is successful, all the minimal candidates of U_{ip} remain suspect. However they represent a different (reduced) set of non minimal candidates. This is intuitively true because t_i found that some elements were working and thus the minimal candidates do not represent any more the candidates containing those elements. For example in a system with two components a and b, $[\overline{a}:]$ represents $[\overline{a}:b]$ and $[\overline{ab}:]$. If b is found to be working, it only represents $[\overline{a}:]$. Therefore it is necessary to find out how the new and the old probabilities of the minimal candidates are related.

Again, every suspect must contain a permutation of all the elements of the system, where

the individual value of an element can be either element working (x_i) or element not working ($\overline{x_i}$). The elements of U_{ip} can therefore be divided into two parts: those that contain the permutation [x_{a1} .. x_{aa} x_{b1} .. x_{bb} x_{c1} .. x_{cc}] of A_i B_i C_i (together with some permutation of the other elements of the system), and those that contain some other permutation of the A_i B_i C_i elements (together with some permutation of the other elements of the system). The sum of the probabilities of all the permutations of the elements of A_i B_i C_i must be equal to 1 since within each permutation an element can have either probability α or $(1-\alpha)$. Therefore the probability of any element of U_{ip} can be approximated as a combination of two probabilities:

$p(X)\ p(\text{perm. of } P_i) + p(1-X)\ p(\text{perm. of } P_i)$

where

$p(X) = p([\ x_{a1} .. x_{aa}\ x_{b1} .. x_{bb}\ x_{c1} .. x_{cc}\ ])$

$= p(U_{ip}=f)\ p(C_i=w)$

(This is only an approximation because the second term of the expression is in reality not one element but a series of elements).

The second element of the expression will disappear since its posterior probabilities (that is the probability given that the test is successful) must be equal to 0. Therefore for each element of U_{ip} we have kept only the portion $p(U_{ip}=f)\ p(C_i=w)$. The portion $1 - p(U_{ip}=f)\ p(C_i=w)$ has disappeared. If the old entropy of U_{ip} was :

$H2 = \sum_{i=1}^{i=u} p(x_i) \log p(x_i)$ the new entropy must be:

$$H'2 = \sum_{i=1}^{i=u} p(U_{ip}=f)\ p(C_i=w)\ p(x_i) \log (p(U_{ip}=f)\ p(C_i=w)\ p(x_i))$$

$$= p(U_{ip}=f)\ p(C_i=w) \sum_{i=1}^{i=u} [H2 + p(U_i=f) \log (p(U_{ip}=f)\ p(C_i=w))]$$

GLOSSARY

Blackboard Level Control Module: The blackboard control entity associated with each control level of the blackboard.

Blackboard Overall Control Module: The blackboard control entity in charge of the coordination of the Blackboard Level Control entities.

Candidate or Candidate tuple: A set of system components whose failure would explain the observed symptoms.

Dynamic System: System whose state can change while it is under observation.

Failed test: Synonym for unsuccessful test

Filtering of symptoms: The ignoring of expected symptoms (i.e. symptoms caused by known faults)

Fault: Malfunction of a system component.

Fault Manager: The component of the system under observation which is in charge of discovering, repairing and recovering from faults in the system.

Implicated objects: Sinonym for suspects.

Intermittent fault: Malfunction of a system component, which manifests itself only periodically.

Minimal candidates: The smallest sets of system components whose failure would explain the observed symptoms.

Multiple fault: Simultaneous ocurrence of several faults in the system under

observation.

Performance Manager: The component of the system under observation which is in charge of monitoring, analysing and improving the performance of the system.

Sensor: Component of the system under observation whose function is to monitor the system by performing tests or observations.

State Change: From a fault management point of view, a state change occurs when a working system component becomes faulty or a faulty component reverts to a working state.

Successful test: Test which observes a correct, working behaviour.

Suspect combination: The combining of the sets of suspect components, performed in order to obtain the minimal candidates.

Suspects: Elements of the system whose failure can lead to the observed symptom(s).

Symptom: The manifestation of a fault through a malfunction.

Symptom translation: The transformation of a symptom into the set of components whose failure would explain the symptom.

Test Fault: Test result which does not represent correctly the true state of the components it is testing.

Test optimisation: The action of selecting the best posssible test among the available tests.

Topology manager: The component of the system under observation which is in charge of keeping track of all the elements of the system.

Unsuccessful test: Test which observes a faulty behaviour.