

DIAGNOSING PROBLEM-SOLVING
SYSTEM BEHAVIOR

EVA HUDLICKÁ

COINS Technical Report 86-03
February 1986

Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

Submitted to the Graduate School of the
University of Massachusetts in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

February 1986

Computer and Information Science

This research was sponsored, in part, by the National Science Foundation under Grants NSF# DCR-8500332 and NSF# DCR-8318776 and by the Defense Advanced Research Projects Agency (DOD) monitored by the Office of Naval Research under Contract N00014-79-C-0439, P00009.

Eva Hudlická

©

1986

All Rights Reserved

To my parents and YF

“What have you lost, Mulla?”
“My key,” said Nasrudin.
“Where did you drop it?”
“At home.”
“Then why, for heaven’s sake, are you looking here?”
“There is more light here.”

- A Sufi Parable

Acknowledgements

I would like to thank my committee:

Victor Lesser for his direction; this dissertation has been greatly influenced by him.

Daniel Corkill for time, advice, and criticism, as well as for the Clisp and ImageTalk environments, and for figures 7 and 9. Dan made many valuable suggestions, both during the development of the ideas presented here and during the writing.

John Stankovic for careful reading of the early drafts and for his down to earth comments.

Al Hanson for asking the right questions about this work.

I also want to thank Alex Wolf for T_EX advice and his thesis macros.

David Stemple and M for helping me keep the correct perspective on AI.

Srini for encouragement and proposal reading.

Alan for the Macintosh, for showing me that while this may have been useful, it was not necessary, and for hastening, in many ways, the writing of this dissertation.

K for reminding me that there is life after (or even during) graduate school, and for encouragement and support during this past year.

And, last but not least, I thank my cats for providing continuous entertainment. I am sure they will find their own uses for this document!

A úplně nakonec, ale vůbec ne nejméně, chci poděkovat rodičům, za jejich rozhodnutí před skoro osmnácti lety a za jejich trpělivost těchto dlouhých šest let, a tetě Daně, za to, že Československo je pořád ještě domovem!

ABSTRACT

Diagnosing Problem-Solving System Behavior

February 1986

Eva Hudlická

BS, Virginia Polytechnic Institute and State University

MS, The Ohio State University

PhD, University of Massachusetts

Directed by: Professor Victor Lesser

The complexity of man-made systems is rapidly increasing to the point where it is becoming difficult for us to understand and maintain the systems we build. AI problem-solving systems are particularly susceptible to this information overload problem, due to their often ad hoc design, large knowledge-bases, and decentralized control mechanisms. This has recently resulted in a trend towards more autonomous systems; systems that can explain their behavior, aid the developers with debugging, and monitor and adapt their behavior, in order to function well in a changing environment. Central to all these functions is the ability of the problem-solving system to reason about its own behavior.

This dissertation describes a system component, the Diagnosis Module (DM), that enables a problem-solving system to reason about its own behavior. The

problem-solving system being diagnosed is a distributed interpretation system, the Distributed Vehicle Monitoring Testbed (DVMT). The DM uses a causal model of the expected behavior of the DVMT to guide the diagnosis. The aim of the diagnosis is to identify inappropriate control parameters or faulty hardware components as the causes of some observed misbehavior. The DM has been implemented and successfully identifies faults in the DVMT.

Causal model based diagnosis is not new in AI. What is different about this work is the application of this technique to the diagnosis of problem-solving system behavior. Problem-solving systems are characterized by the availability of the intermediate problem-solving state, large amounts of data to process, and, in some cases, lack of absolute standards for behavior. We have developed diagnostic techniques that exploit the availability of the intermediate problem-solving state and address the combinatorial problem arising from the large amounts of data to analyze. We have also developed a technique for dealing with cases where no absolute standard for correct behavior is available. In such cases the system selects its own "correct behavior criteria" from objects within the DVMT which did achieve some desired situation.

Table of Contents

ACKNOWLEDGEMENTS	v
LIST OF FIGURES	xiv
CHAPTER	
I. OVERVIEW	1
§1. Motivation: Need for more Autonomous Systems	1
§2. Problem Definition: What is Diagnosis?	5
§2.1 Diagnosis of Problem-Solving Systems	8
§2.2 Types of Faults Found in a Problem-Solving System	10
§3. Selection of a Diagnostic Method	12
§4. Implementation	19
§5. Assumptions	22
§5.1 Practicality and Feasibility of Model Construction	23
§6. Contributions of this Work	27
§7. A Guide for the Reader	33
§8. Summary	34
II. THE DVMT AND EXAMPLES OF FAILURES	37
§1. The Distributed Vehicle Monitoring Testbed	37
§1.1 The Task of the DVMT	39
§1.2 Node Architecture	43
§1.3 DVMT System Organization	50
§2. Unique Characteristics of the DVMT System	51
§2.1 The Complexity of the DVMT System	53

§2.2 Availability of the Intermediate DVMT States	54
§3. Examples of DVMT Failures DM Can Diagnose	55
§3.1 Example I: Four-Node System without Communica- tion Knowledge Sources	56
§3.2 Example II: Single-Node/Two-Sensor System with a Failed Sensor	60
§4. Summary	65
III. MODEL STRUCTURE	68
§1. Introduction	68
§2. Hierarchical State Transition Diagrams	73
§2.1 Predicate States	74
§2.2 Relationship States	77
§2.3 Primitive States	78
§2.4 Transition Arcs Linking the States	79
§2.5 Data-Dependent Choice of Neighboring States	83
§2.6 Organizing the States into a Hierarchy	84
§3. Abstracted Objects	89
§4. Constraint Expressions among the Abstracted Objects	95
§5. Object-State Links	97
§6. Model Instantiation	97
§7. Summary	102
IV. MODEL USE	104
§1. Introduction	105
§2. A Dammed Metaphor	107
§3. Backward Causal Tracing	114
§4. Forward Causal Tracing	126
§5. Comparative Reasoning	132
§6. Situation Matching Reasoning	138

§7. Unknown Value Derivation	142
§8. Inconsistency Resolving	148
§9. Use of Underconstrained Objects	150
§10. Checking Global Consistency via Path Values	152
§11. Summary	156
V. EXAMPLES OF MODEL USE FOR DIAGNOSIS	158
§1. Introduction	158
§2. EXAMPLE I: Missing Communication Knowledge Sources .	163
§2.1 Diagnosis for Node #1	166
§2.2 Diagnosis for Node #2	171
§2.3 Diagnosis for Node #3	179
§2.4 Diagnosis for Node #4	180
§2.5 Diagnosis for the Pending Symptoms at Node #2 . .	180
§3. EXAMPLE II: Low Sensor Weight and Malfunctioning Sen- sor	185
§4. Reducing the Complexity of the Diagnosis	202
§4.1 Object Grouping	204
§4.2 Underconstrained Objects	208
§4.3 Hierarchy	210
§5. Summary	212
VI. BUILDING A CAUSAL MODEL	214
§1. Choosing the States	215
§2. Organizing the States into a Hierarchical Model	219
§3. Choosing the Abstracted Objects	225
§4. Constructing the Constraint Expressions Among the Objects	229
§5. Types of Abstraction Necessary to Reduce the DVMT Sys- tem Complexity	232
§6. Summary and Conclusions	245

VII. RELATED WORK	248
§1. Introduction	248
§2. Medicine: CASNET	252
§3. Medicine: ABEL	256
§4. Digital Circuits: DART	262
§5. Hardware: Davis's circuit analyzer	266
§6. Summary of Comparisons of AI Diagnostic Systems	266
VIII. IMPLEMENTATION DETAILS	269
§1. Introduction	269
§2. Abstracted Objects	271
§3. Constraint Expressions	278
§4. The State Transition Diagram	284
§5. State Transition Arcs	297
§6. Model Instantiation: An Example	308
§7. Summary	317
IX. CONCLUSIONS AND CONTRIBUTIONS	318
§1. Introduction	318
§2. Defining the Problem	320
§3. Solving the Problem	322
§4. Major Difficulties Encountered	324
§5. Contributions of this Research	325
§6. How Domain Independent is this Work?	328
§7. Where Do We Go from Here?	330
X. FUTURE RESEARCH	331
§1. The Best of All Possible AI Systems	331
§2. Extensions to the DM	336

§2.1	Detection	336
§2.2	Diagnosis	338
§2.3	Correction	340
§2.4	Distribution of the Diagnostic Interpreter	342
§2.5	Availability of the DVMT Intermediate States	343
§3.	Augmenting the System Behavior Model	343
§3.1	Extend the Degree of DVMT Behavior Represented in the Model	344
§3.2	Automatic Hierarchy generation	344
§3.3	Automatic Extension of Model	345
§3.4	Extending the Type of Knowledge Represented by the Model	346
§4.	Summary	346
BIBLIOGRAPHY		348
APPENDIX		
A. GLOSSARY		352
B. THE SYSTEM BEHAVIOR MODEL		360
§1.	Introduction	360
§2.	Answer Derivation Cluster	361
§3.	Ksi Scheduling Cluster	370
§4.	Communication Cluster	377
§5.	Ksi Rating Derivation Cluster	383
C. DETECTION OF A PROBLEM		388
§1.	The Detection Problem	388
D. THE DIAGNOSIS MODULE TRACES FOR THE EX- AMPLES		392
§1.	Introduction	392
§2.	Trace Format Description	393

§3. Traces for Example I	400
§3.1 Node #1: PT to MESSAGE-ACCEPTED	400
§3.2 Node #1: MESSAGE-ACCEPTED to MESSAGE- SENT	403
§3.3 Node #2:PT1-VT5	407
§3.4 Node #2: PT3-PT1 True-False Pair	410
§3.5 Node #2: Pending Symptoms	415
§4. Traces for Example II	422

List of Figures

1. Architecture of a Fully Fault Tolerant Problem-Solving System	3
2. Problem-Solving Control Errors	6
3. Difference Between a Fault Dictionary and a Causal Model Representations of System Behavior	14
4. Architecture of the Diagnosis Module	20
5. Relationship Between the Diagnosis Module and the DVMT System	21
6. Alternative Points of Model Construction in the System Development Cycle	24
7. The Distributed Vehicle Monitoring Testbed	38
8. A High-Level View of the Data Transformation	40
9. System Signal Grammar	42
10. Graph of the Data Transformation	44
11. The Structure of Objects in the DVMT System	46
12. The Structure of Processing at Each Node in the DVMT System	49
13. Relationship between DVMT and DM Systems and Their Domains	52
14. Scenario for Example I	57
15. Scenario for Example II.	61
16. A High Level View of the Modeling Formalism	71
17. Legend for Figures in the Dissertation	72
18. Illustrating the Modeling Formalism	76

19. State Transition Diagram Illustrating Relationship States	78
20. Relationships Among the State Neighbors	82
21. Linking Clusters at Different Levels of the Hierarchy	86
22. The Different Types of Neighbors a State May Have	87
23. Relationship Among the DVMT system, the Abstracted Objects, and the State Transition Diagrams	91
24. Different States Can Refer to the Same Object	92
25. Dummy Double page figure	98
25. The Model Clusters Representing the DVMT Behavior	99
26. A Canal System Analogy for the DVMT Diagnosis	108
27. The Types of Reasoning Used by the DM	113
28. Causal Relationships Among States	116
29. Causal Pathways in the Instantiated SBM	118
30. BCT Search Strategy	119
31. Algorithm for Backward Causal Tracing	120
32. BCT Terminating Conditions	122
33. BCT-Constructed Causal Pathways	125
34. Search Strategy Imposed on the SBM Instantiation by FCT	129
35. Algorithm for Forward Causal Tracing	130
36. Algorithm for CR	135
37. Example of CR	136
38. Example of Unknown Value Derivation	145
39. A Situation Where UVD Fails	146
40. List of Abbreviations Used Throughout This Chapter	160
41. Interest Areas for Example I	164

42. Instantiated SBM for Diagnosis in Node #1	167
43. Instantiated SBM for Diagnosis in Node #2-Part I	172
44. Instantiated SBM for Diagnosis in Node #2-Part II	175
45. Instantiated SBM for Diagnosis of Pending Symptoms in Node #2	182
46. Instantiated SBM for Example II-Part I	188
47. Instantiated SBM for Example II-Part II	190
48. Instantiated SBM for Example II-Part III	192
49. Reduction of the Number of States by Object Grouping	205
50. Maximum # of States Created for Each Cluster	208
51. Strategies for Choice of States in the SBM	217
52. Two Versions of the SBM Modeling the Same Process	220
53. Construction of Causal Pathways	223
54. The Types of Abstractions Used in Model Construction and Reasoning	234
55. Representing a Class of Objects in the Uninstantiated Model	236
56. Representing a Series of Steps in the Uninstantiated Model	239
57. The Iterative Object Grouping Process	243
58. DVMT Object and its Abstracted Object Representation	273
59. A List of the Abstracted Object Fixed Attributes	274
60. Variable Object Attributes	276
61. The Grammar for the Constraint Expression Syntax	280
62. The Types of Dependencies Among Attributes	281
63. Illustrating the Use of Path Expressions	282
64. Constraint Expressions	284
65. Multiple Path Values of a State	288
66. Illustrating the Path Value Calculation	291

67. Syntax of the Object Grouping Specification	295
68. State Attributes	298
69. Grammar for the Neighbor List Specification	299
70. Motivating the Need for the PrefOR Logical Operator	303
71. Data Dependent State Neighbors in Ksi Scheduling Cluster	306
72. Illustrating the Specification of the Dynamic Neighbor Names	307
73. Uninstantiated VT State	309
74. Uninstantiated VT-HYP-OB Abstracted Object	310
75. Instantiated VT-HYP-OB Abstracted Object	311
76. Instantiated VT State	312
77. Instantiated SBM	315
78. A Possible Architecture of an Autonomous Knowledge-Based System .	333

Chapter I

OVERVIEW

**'You may call it "nonsense" if you like,' she said,
'but I've heard nonsense, compared with which
that would be as sensible as a dictionary!'**

- Lewis Carroll, in *Through the Looking Glass*

This chapter is a high-level overview of the research described in this dissertation. It discusses the motivation for this research, defines the problem, and justifies the approach selected for solving it. The contributions are outlined at the end of the chapter.

§1. Motivation: Need for more Autonomous Systems

The role of man-made systems is rapidly shifting, from that of mere tools, performing simple, tedious tasks, to that of partially autonomous assistants. As the system complexity increases, so does the amount of information that needs to be processed, both by the system developer and by its users. It would therefore be of great help if the system could take on greater responsibility for its own behavior, including more sophisticated control, some system maintenance tasks, and the ability to explain and justify its behavior.

Artificial Intelligence (AI) systems have followed this trend towards greater complexity. Problem-solving systems encoding expertise in some domain¹ are called upon to help make decisions and to perform complex interpretation tasks. As the difficulty of the tasks increases, more knowledge is necessary to solve them. In addition to task specific knowledge (also called domain knowledge), knowledge about the problem-solving process itself becomes necessary, in order to manage the various resources, including the domain knowledge. This problem of partitioning the task, deciding which portion to work on next, and what knowledge is applicable at each stage constitutes the control problem in AI. The type of knowledge necessary for making control decisions is called control knowledge.² It is often difficult to decide on a suitable control strategy for some problem-solving task a priori. For this reason, AI systems requiring complex control are often parametrized to allow easy modification of their control strategies. In some applications, in addition to controlling the problem-solving strategies, the parameters may control various hardware components, such as processors, communication channels, sensors, or effectors.

Ideally, such sophisticated problem-solving systems would function in an independent manner, working on their tasks without the help of a human overseer. They could monitor their own behavior and adjust the various parameters as necessary, in order to function optimally in a changing environment. At one extreme of this self-sufficiency spectrum is a fully adaptive system; one that has

¹This includes knowledge-based or expert systems.

²The control problem has received much attention recently [8,16,35].

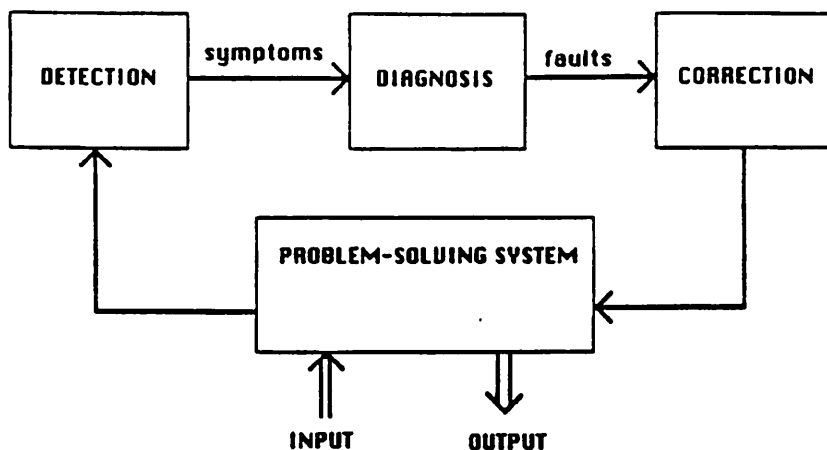


Figure 1: Architecture of a Fully Fault Tolerant Problem-Solving System

A fully fault tolerant problem-solving system would contain a detection, a diagnosis, and a correction component. The detection component would monitor the problem-solving system behavior and would detect any deviations from expected behavior. The diagnosis component would receive reports of such misbehaviors and would identify the faults which caused them. The correction component would correct the faults in the problem-solving system.

some expectations for its own behavior, is capable of monitoring its behavior, and can detect, diagnose, and correct inappropriate behavior. Such systems are called fault tolerant in the computer hardware literature, because they can tolerate some degree of component failure. Figure 1 shows the system components in a fully fault-tolerant architecture.

Making a system fully fault tolerant requires three things:

1. First, the criteria for appropriate system behavior must be developed. These allow the system to monitor itself and to detect discrepancies between the actual and the expected behavior. Such criteria are often difficult to determine. In a problem-solving system, for example, one does not know the correct answer with which to compare the system's answer. These criteria must therefore be based on indirect methods of judging appropriate behav-

ior, such as internal system consistency.

2. Second, once a problem is detected, the system must diagnose what caused it. The causes are some predefined categories of faults. Usually these are related to the available methods for correcting the failures. The aim of diagnosis is to find an explanation for the observed misbehavior in terms of repairable components of the system.
3. Finally, the correction process must begin. In some cases, for example traditional distributed processing systems, this phase begins with the restoration of a consistent system state among the various processors. The system then proceeds to adjust its parameters or to replace the faulty hardware component and continues in its normal functioning.

The original intent of this work was to construct a fully fault tolerant problem-solving system. However, this proved to be a task well beyond the scope of a single dissertation. A portion of this task, the diagnosis problem, was therefore selected as the focus of this research. This dissertation describes progress made toward making a problem-solving system capable of diagnosing its own behavior. The rest of this chapter further defines the problem of diagnosis, especially in the context of problem-solving systems. This is followed by a discussion of the current AI approaches to diagnosis and a motivation for our selection of the causal model based approach. Finally, the implementation of the diagnostic component is briefly discussed and the contributions of this research are outlined.

§2. Problem Definition: What is Diagnosis?

The problem of diagnosis is to identify the causes of some inappropriate behavior. Diagnosis begins when some undesirable situation (a **symptom**) is observed, and proceeds by trying to determine the **causes** of that symptom. A successful diagnosis consists of explaining the symptom in terms of causes for which corrective actions are known. In medicine, for example, such causes would be some diseases with known therapies. In hardware, they would be various replaceable components, such as circuit boards or system modules. In a problem-solving system, diagnosis would explain the observed symptom in terms of some faulty control parameter setting, a software bug, a problem with the knowledge-base, or some failed hardware component, such as a processing node, a channel, or a sensor.

The goal of this work was to construct a component of a problem-solving system capable of diagnosing the system's behavior and identifying the causes for inappropriate behaviors. By inappropriate behavior we mean the lack of a complete solution, a solution that does not satisfy some user imposed constraints, (such as time or quality), or a poor problem-solving strategy. Figure 2 illustrates an example of a system in dire need of an ability to monitor and adjust its problem-solving control strategy. The result of the diagnosis is a set of faults, which explain the initial symptom. The types of faults the diagnostic component finds are discussed in detail below. (Note that we are not attempting to solve the detection problem; that is, to automatically determine when diagno-

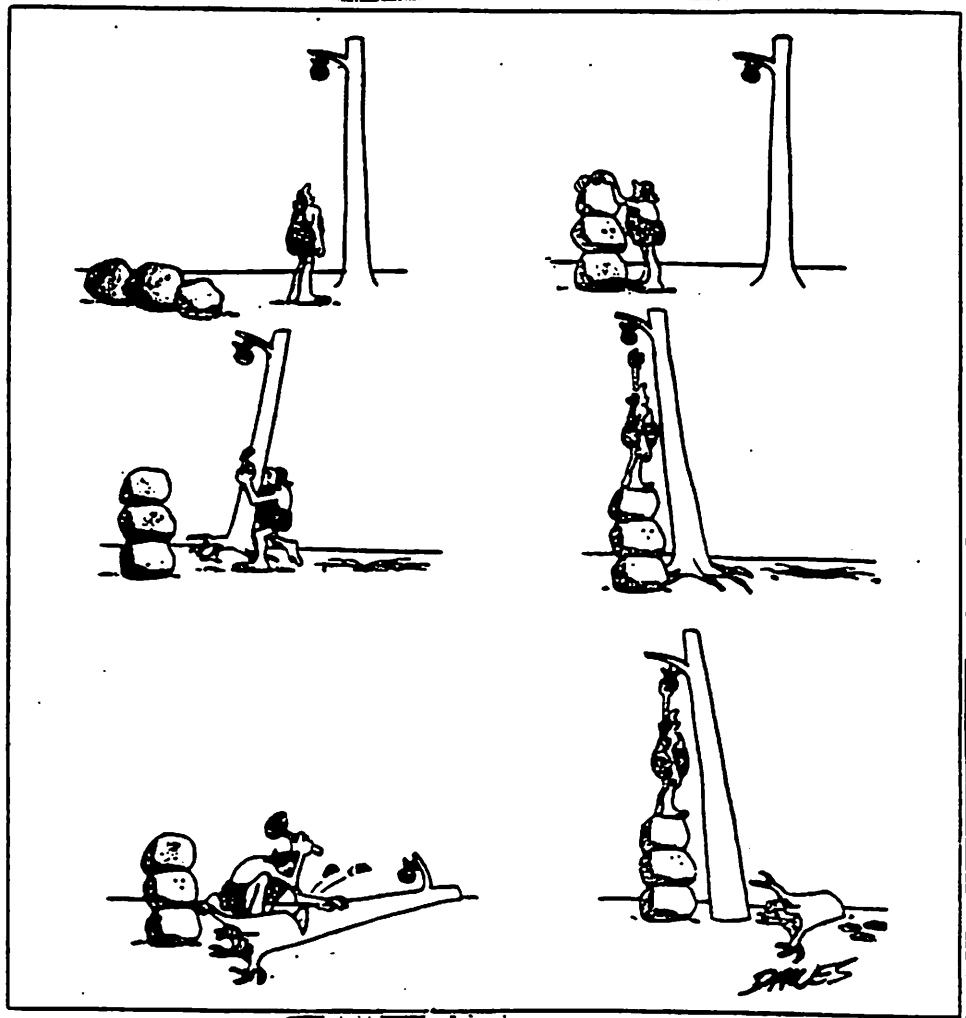


Figure 2: Problem-Solving Control Errors

This figure illustrates a poor control strategy in a particularly primitive problem-solving system.

sis is appropriate. The initial symptoms are provided as data to the diagnostic component.)

A component capable of diagnosing problem-solving system behavior would be useful for a number of applications:

ADAPTIVE PARAMETER CONTROL. Although most problem-solving systems are highly parametrized, they rarely adapt their parameters to changes in the problem-solving environment. The parameters are generally set at the beginning of each task. If either the system or the task characteristics change during the run the system cannot adapt to this change and its performance may degrade. The ability to diagnose its own behavior and to identify the faulty parameters is the first step in adapting to a changing environment. The diagnosis component is thus an integral part of a fully fault-tolerant system architecture

DEBUGGING. Problem-solving systems are becoming more complex and the amount of information that has to be processed by their designers and developers is enormous. A component of a problem-solving system that could diagnose the system's behavior would serve as an assistant during the system development phase.

EXPLAINING SYSTEM BEHAVIOR. Problem-solving systems often generate large traces of their behavior. These traces must be interpreted in order to understand why the system behaved as it did. Due to the system complexity, such trace interpretation is a time-consuming, tedious task. A component capable of explaining the system's behavior would make this task much easier. The diagnostic component described here can also be used to explain system behavior.

§2.1 Diagnosis of Problem-Solving Systems

The original motivation for this research came from our experience with a particular problem-solving system: the Distributed Vehicle Monitoring Testbed (DVMT) [23]. The DVMT is a testbed for investigating how to organize and control distributed problem solving systems. A detailed description of the DVMT is in Chapter II. Briefly, the DVMT's task is to interpret acoustic signals generated by vehicles moving through the environment and to produce a map identifying the types of vehicles and describing the paths they took. The acoustic signals are sensed by sensors which are distributed throughout the environment. The sensors send their signals to processors, which integrate the discrete sensory data into descriptions of the vehicle types and their movements. Since each processor typically senses only a portion of the data, communication among the processors is necessary in order for the system to construct a map of the overall environment. The system thus consists of a number of autonomous processors, each working on its part of the overall task.

Numerous parameters are responsible for establishing the control strategy for a particular interpretation task and system configuration. These parameters determine, for example, which processor works on what data, who communicates with whom and what types of messages are transmitted, and what knowledge is available at each processor. Errors in these parameter settings can cause the cooperating processors to work at cross-purposes or to fail to work on important portions of the developing solution. Examples of such errors are: inappropriate

communication decisions (a processor sending messages to another processor that cannot react to them), inappropriate focus of attention (a processor is externally-directed when internal direction would be more appropriate), or inappropriate task allocation (processors are assigned work in the wrong areas). Since these parameters are responsible for the control of the problem-solving in the DVMT, we refer to their faulty settings as **problem-solving control failures**.

Currently, these control parameters are set at the beginning of each experiment and the system is allowed to run to completion, each processor generating as much of the overall vehicle map as possible, given its data and control parameters. In a fully functional distributed problem-solving system however, these parameters would be set by the control component of the system and would be updated as necessary.

There are many situations in the DVMT where the ability of the system to diagnose its own behavior would be useful:

- The numerous parameters may be set inconsistently to begin with. This is typically not discovered until the experiment is completed, resulting in much wasted effort, both on the part of the experimenter and the machine.
- The parameters are set correctly for the initial data and system configuration, but both of those may change during the experiment; different types of data arrive or some of the hardware components fail. The fixed parameter settings result in poor use of the system resources at best, and lack of the final answer at worst.
- Once the experiment is finished, the results must be analyzed. This analysis is a time consuming process, involving detailed examination of the system

traces. During the development phase, each experiment must be analyzed in detail, in order to make sure the system is behaving correctly. Whenever the system did not perform as expected, for whatever reasons, the experimenter must spend hours examining detailed system traces in order to find out what caused the discrepancy.

All of the above tasks would be made much easier if the system was capable of diagnosing its own behavior and explaining it in terms of various primitive causes, such as inappropriate parameter values or faulty hardware components. These could then be adjusted, either by the system itself, or by a human.

§2.2 Types of Faults Found in a Problem-Solving System

What faults need to be diagnosed in a problem-solving system? There are four categories of faults (primitive causes), which could lead to an observed symptom.

1. Faulty hardware components such as sensors, communication channels, or failed processors; the **hardware failures**.
2. Faulty settings of the parameters controlling the type of processing at each node (e.g., data-directed vs. goal-directed, breadth-first vs. depth-first search), the distribution of data among the processors, the communication among the processors, and the application of knowledge to the problem. Since most of these parameters are responsible for the control of the problem solving process, we call these types of faults **problem-solving control failures**.
3. Software bugs in the problem-solving system code.
4. Insufficient facts to solve the problem. This might be missing data or lack of appropriate knowledge.

Although these are clearly very different types of failures, their manifestations in the DVMT system are indistinguishable. They result in identical symptoms, such as lack of the overall interpretation of the environment, or wasteful problem-solving process. For example, the fact that a processing node failed to derive a hypothesis describing the motion of some vehicle may be due to one or more of the following causes:

- a failed sensor that did not sense the data (hardware failure);
- a failed channel that did not transmit the necessary data sent from another node (hardware failure);
- a missing knowledge source³ that prevented the node from deriving the answer (problem-solving control failure);
- a faulty parameter setting did not allow the node to process part of the input data (problem-solving control failure).

We are primarily interested in diagnosing problem-solving control failures, caused by inappropriate parameter values. However, since the manifestations of these failures are indistinguishable from hardware failures,⁴ such as a failed sensor or channel, we can use the same mechanism to diagnose both failure types.

³A knowledge source is a piece of code performing some well-defined function.

⁴This is also true for hardware and software errors [31]. Although we have chosen not to diagnose software errors, we believe that our approach could handle those as well.

§3. Selection of a Diagnostic Method

Diagnosis is not a new problem for Artificial Intelligence. Much work has been done in medical diagnosis, in diagnosis of computer systems, digital circuits, electronic devices, as well as larger systems such as nuclear reactors. This work, and its relationship to the diagnostic component described here, is discussed in Chapter VII. The approaches to diagnosis fall onto a spectrum. At one end is the approach involving knowledge about the symptoms→fault associations but no knowledge about the structure and expected behavior of the diagnosed system; this is termed **fault dictionary based diagnosis**. A fault dictionary consists of a collection of symptom→fault pairs (rules), which represent the associations between the observed symptoms and the faults that caused them. At the other end is the approach involving knowledge about the system's structure and function, which allows reasoning about a large subset of the possible system behaviors, both correct and faulty. For a given situation, the expected modes of behavior will typically be a small subset of the possible behaviors. Diagnosis using such a representation is termed **causal model based diagnosis**. Causal model based diagnosis does not necessarily differentiate between modeling failure modes and correct processing. Since faulty behavior can occur, it should be represented by a causal model as one of the possible behaviors. What distinguishes a causal model from other representations, is that it represents the structure and function of the system's components and the relationships among them. Unlike fault dictionary based diagnosis, causal model based diagnosis does not view the system as a

black-box with only a few observable inputs and outputs. Figure 3 illustrates the difference between the types of knowledge used by these two approaches.

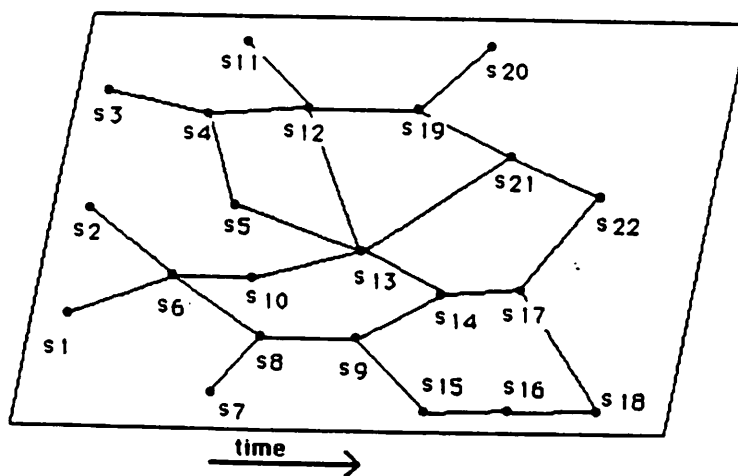
The difference between these two approaches is analogous to the recently discussed difference between shallow and deep models of domain knowledge in knowledge-based systems [9,26]. This difference is best illustrated by the following example. A student can go into an exam anticipating all the questions asked and just recalls the memorized answer. This may be more efficient than actually thinking about the problems, and such a student may appear impressive. However, since he cannot always anticipate all the questions, this student, not having any real understanding of the subject matter, will not be able to answer questions he has not anticipated and will therefore not do well in the long-run. Similarly, a knowledge-based system with shallow knowledge about its task can do very well in a narrow domain of expertise but does not possess any real understanding of the domain and will not be able to handle any new problems for which it does not have an explicit rule.

In contrast, a student who knows the principles behind the subject matter, but not the answers to many specific questions, may take longer to answer a particular question, but will be able to solve a much larger set of problems. This type of knowledge representation and reasoning is what deep (causal) models of a domain strive for. Because of the additional knowledge of the problem, knowledge-based systems with deep models are capable of graceful degradation when they reach the limits of their expertise, unlike their shallow rule counterparts. The difference between the two approaches is the level of information included in the system and

SYMPTOM	FAULT	PROBABILITY
S22	S11	.3
	S3	.2
	S2	.4
	S1	.05
	S7	.05
S20	S11	.7
	S3	.3
S18	S7	.5
	S1	.2
	S2	.3

FAULT DICTIONARY REPRESENTATION OF SYSTEM BEHAVIOR

PART A



CAUSAL MODEL REPRESENTATION OF SYSTEM BEHAVIOR

PART B

Figure 3: Difference Between a Fault Dictionary and a Causal Model Representations of System Behavior

Part A shows a fault dictionary representing the symptom-fault pairs and the associated probabilities. Part B shows a state transition diagram representation of the same system. This causal model representation makes the system structure explicit and can thus support reasoning from first principles about the system behavior.

the amount of reasoning necessary to use the information.

The two approaches can be compared along two dimensions:

1. The amount of work necessary to prepare the diagnostic knowledge-base, either the fault dictionary or the causal model of the possible system behaviors, and ease of modifying the knowledge-base.
2. The effectiveness of the knowledge in diagnosis, including efficiency, the types of reasoning each form of knowledge supports, and how well it deals with cases for which no specific knowledge exists and with multiple failures.

A detailed discussion and comparison of these two approaches to diagnosis is in Chapter VII. Below is a summary of their distinguishing characteristics which serves as a motivation for our selection of the causal model based diagnosis.

Advantages and disadvantages of fault dictionary based diagnosis.

ADVANTAGES:

- Efficient diagnosis if table size is small;
- Does not require understanding of the diagnosed system, only empirical observations of associations between symptoms and faults.
- In its simpler forms, does not require sophisticated reasoning mechanisms.

DISADVANTAGES:

- Requires a long time to collect the symptom→fault pairs necessary to construct the dictionary.
- Requires updating any time the system characteristics change; this becomes costly when the fault dictionary is large.

- If no entry can be matched to an observed symptom, no fault will be found. The dictionary based diagnosis thus does not support graceful degradation.
- Knowledge in the dictionary is limited to diagnosis and prediction of faulty behavior.
- When multiple failures need to be diagnosed, the size of the dictionary quickly grows beyond practical limits, because all possible combinations of failures must be explicitly encoded.

These characteristics of fault dictionary based diagnosis suggest the characteristics of the systems for which it is appropriate.

- Stable systems, whose modes of failures are well known and do not change.
- Systems whose structure is not known and only associations among symptoms and faults can be observed.
- Systems with a relatively small number of symptoms and faults, so that the dictionary size is manageable.

The advantages and disadvantages of causal model based diagnosis.

ADVANTAGES:

- Supports graceful degradation of diagnostic expertise by at least narrowing down the problem when exact knowledge is not available. Can therefore handle unexpected situations more easily than a fault dictionary approach.
- Represents knowledge in a more concise form than a fault dictionary and can reason from first principles about the system behavior.
- Knowledge in the model is not tailored towards a specific use and is therefore not limited to diagnosis.

- Does not require lengthy compilation of symptom→fault rules.
- Can diagnose multiple failures by tracing their causal pathways in the model.

DISADVANTAGES:

- Requires knowledge of the internal system structure.
- The construction of the system model can be time consuming.
- Requires sophisticated reasoning mechanisms.
- Diagnosis may require a long time since reasoning from first principles is necessary.

Again, from these characteristics, we can see the types of systems for which a causal model based diagnosis is appropriate.

- Systems whose modes of failure change rapidly.
- Systems whose structure is known so that a causal model can be constructed.
- Systems with many faults leading to the same symptom or many symptoms having the same underlying fault. This includes system where multiple faults are common.

Which of these diagnostic approaches is more appropriate for problem-solving system diagnosis? Looking at the above system characteristics, it is clear that a problem-solving system falls into the category of systems for which causal model diagnosis is suitable. Let us examine in detail a few of the DVMT characteristics that make causal model based diagnosis appropriate.

Rapidly changing modes of failure: Given the number of DVMT control parameters, it would be impossible to catalog all the possible combinations of their values and the behaviors these combinations would cause. In the context of diagnosis, such a catalog constitutes the modes of failure and would be required for a fault dictionary based diagnosis. It is much more efficient to encode the effect of these parameters on the system behavior and then to determine, on a case by case basis, which parameter values were responsible for some symptom.

Knowledge of the system structure: Since the DVMT is a man-made system, its structure is completely known. Furthermore, its intermediate states can be determined simply by examining the system data structures. This makes the construction and use of a causal model feasible.

Multiple faults: It is very common in the DVMT for a number of parameters to be set inappropriately. In a fault dictionary approach, each such combination would require a new entry in the dictionary. In a causal model approach, since each causal pathway can be traced independently, the same model supports diagnosis of multiple faults.

Mapping between symptoms and faults: Due to the DVMT system's complexity, a relatively small number of faulty parameter settings can result in a very large set of diverse symptoms. Again, use of a causal model based diagnosis is a more efficient way of dealing with this problem, since the causal pathway must be represented just once, instead of having to explicitly represent all possible pairs of symptoms and faults.

It is for these reasons that the causal model based approach was selected for diagnosing the DVMT system behavior. The diagnostic component we have constructed uses a model of DVMT system to determine how some situation was achieved or how it should have been achieved.

§4. Implementation

We have implemented a prototype version of a diagnostic component, called the Diagnosis Module (DM). Figure 4 shows the architecture of the Diagnosis Module. The Diagnosis Module is a knowledge-based problem-solving system whose task is the diagnosis of faults in the DVMT system. The knowledge-base of the DM is the causal model representing the possible behaviors of the DVMT system. (A description of this model is in Appendix B.) The data of the DM is an initial symptom, representing some inappropriate system behavior, and the data structures of the DVMT system. The output of the DM is the set of primitive faults (i.e., faulty parameter settings or hardware failures) which caused the symptom. In addition to identifying the faults, the DM also produces a complete set of causal pathways linking the intermediate states between the faults and the observed symptom. (These traces are in Appendix D.) Figure 5 illustrates the relationship between the DM and the DVMT system. The DM consists of about 5000 lines of code (including comments). It is written in a local dialect of lisp called CLISP and runs on VAX a 11/750 under the VMS operating system.

In order to construct a model of the DVMT, a suitable formalism had to be developed for representing its problem-solving system behavior. Our formalism consists of two parallel networks. One network represents the objects in the DVMT system and the relationships among their attributes. The other network represents the states these objects undergo such as creation, deletion, or various

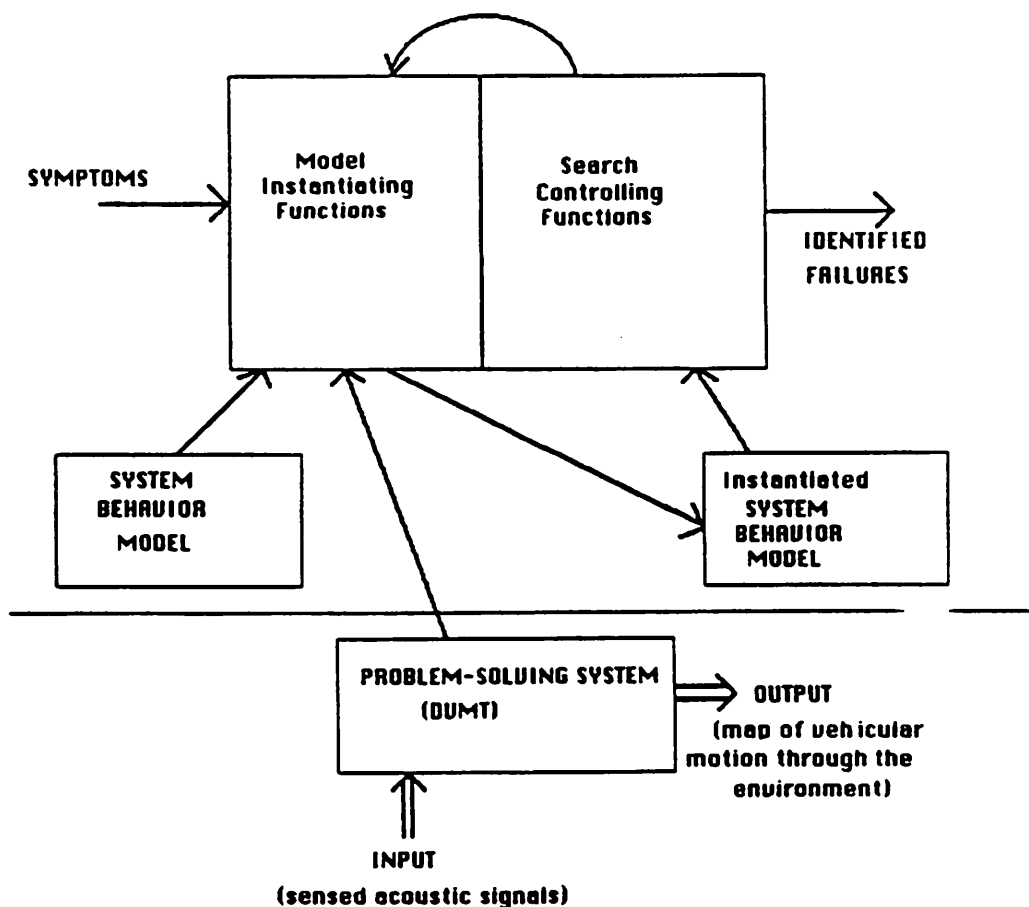


Figure 4: Architecture of the Diagnosis Module

The *Diagnosis Module* receives symptoms of DVMT misbehaviors as its input. Its output are the identified faults responsible for the observed symptom. The *Diagnosis Module* uses a model of the DVMT behavior to guide the diagnosis. A symptom represents a violated expectation. The DVMT model is instantiated to determine how this situation could have been reached by the DVMT. This instantiated model, representing correct behavior, is then compared with the actual behavior of the DVMT, as reconstructed from the DVMT data structures. Any observed discrepancies are traced to the primitive causes which are then reported as failures.

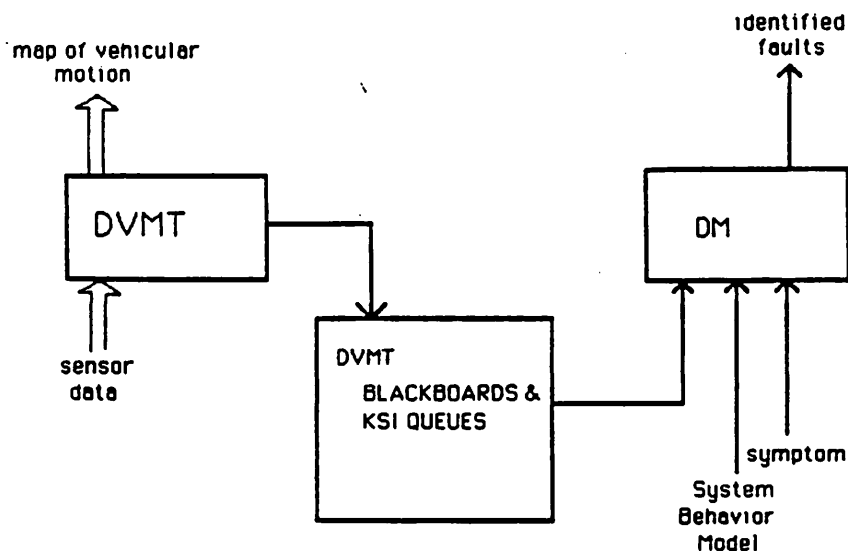


Figure 5: Relationship Between the Diagnosis Module and the DVMT System

This diagram shows the inputs and the outputs of both the DVMT and the Diagnosis Module. The Diagnosis Module communicates with the DVMT system via the DVMT blackboards and ksi queues.

attribute value changes. These two networks are linked together by associating with each object the states it is expected to achieve during the course of a DVMT run. The model thus represents the DVMT structure and possible behaviors. It is divided into small groups of states, called clusters, each representing some aspect of the DVMT behavior. Currently there are 5 model clusters with a total of 39 types of states and 10 types of objects.

Diagnosis begins with the arrival of the initial symptom,⁵ which represents some desired system behavior that was not observed. The symptom used most often is the lack of a hypothesis that spans the entire sensed area of a processor. The model is instantiated in order to determine how such a situation could have

⁵This is currently done by hand, since a detection component has not been implemented.

been achieved by the DVMT system given the current parameter settings and data. This instantiated model is then compared with the actual system behavior, as determined from the DVMT data structures which contain records of the intermediate problem-solving states. Any points where there are discrepancies among the desired and the actual behaviors are then traced to some predefined set of primitive components (i.e., hardware components or faulty parameter settings), which are then reported as the identified faults.

The DM can thus reason about aspects of the DVMT system behavior, using its model of possible system behaviors. It can diagnose a large set of symptoms and explain them in terms of a small set of primitive faults. The DM does not perform any detection or correction. The problems associated with these tasks are not addressed in this dissertation, although some thoughts about detection are discussed in Appendix C. The problems associated with monitoring a problem-solving system and with the balance between monitoring, detection, and diagnosis are not addressed. Currently, the DM functions independently of the DVMT system. In the future, we would like to integrate the DM into the control component of the DVMT.

§5. Assumptions

It is important to make explicit the assumptions underlying this research. The assumptions we make are:

1. The construction of a model representing the correct behavior of a problem-solving system is feasible and practical.
2. The constructed causal model is correct.
3. The information about intermediate states of the problem-solving system is available from the system's data structures.

Each of these assumptions is discussed below.

§5.1 Practicality and Feasibility of Model Construction

Certainly the construction of a causal model of a problem-solving system is feasible, since the structure of the system is known. The question is whether it is practical. In order to answer this question, let us look at the two alternatives for constructing a causal model. Briefly stated, these two alternatives are:

1. is the model constructed "from" the system or
2. is the system constructed from the model?

Figure 6 illustrates the difference between these two methods of model construction.

In the first alternative above, the model is constructed **after** the problem-solving system has been built. The system builder (or a person familiar with the system structure) can construct the model on the basis of his understanding of how the system works. This is what was done in our case. This approach has three shortcomings.

1. It is very time consuming, hence may not be practical.

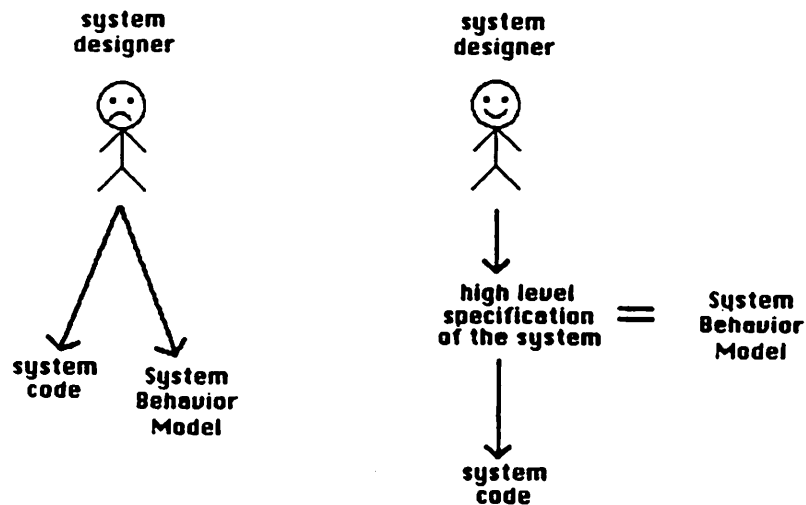


Figure 6: Alternative Points of Model Construction in the System Development Cycle

The left half of the figure illustrates one way of constructing the model of the diagnosed system. Here the model is constructed in parallel with the system code. The right half of the figure shows the preferred way of model construction. Here a high level specification of the system is constructed first, which serves both as the causal model and as the input to an automatic code generation system which then produces the actual system code. The latter method is preferable because it guarantees that the system model and the system code will be consistent.

2. There is no guarantee of the correctness of the model. Although the same knowledge was used for construction of the system and the model, two different processes were used, and it is likely that errors have been introduced and that the system and the model no longer represent the same behavior.
3. There are two objects that must be updated any time a change is made to the system. This is very time consuming and is another source of discrepancies among the system and the system model.

Constructing the causal model after the fact is only practical if the system is fairly stable; i.e., the end of the development cycle has been reached and the causal model then serves to capture the expertise of the person who maintains the system.

The other alternative is to have a high-level specification of the problem-solving system constructed as an intermediate step in the system development, prior to the actual coding. This high-level specification would then serve two purposes: as a preliminary step of the actual implementation and as the causal model for the diagnosis component of the system. This is a logical thing to do since the knowledge necessary for building a software system can also be used to diagnose it, when causal model based diagnosis is used. This approach is taken by Genesereth in his DART program, which uses the manufacturing specifications for VLSI circuits as the causal model for circuit diagnosis [15]. Automatic program generation and program modification from high level specifications is also an active research area [41]. This is the type of environment that would be ideal for causal model based diagnosis of software systems.

Correctness of the Causal Model. The validity of this assumption depends on two things:

1. Whether the model was constructed independently of the system or whether the system implementation was constructed from the model.
2. The level of detail of the causal model (this translates to model size.)

In cases where the model was constructed independently of the system, there is no guarantee that it is correct. The issue of the model size enters into the argument here. The more detailed the model, the more there is a chance that it is inconsistent with the system. Obviously if the model is very simple, it is less likely that the model builder has made an error and it is easier to compare the system with the model. However, if the system is constructed automatically from the model, then as long as we can guarantee that the implementation correctly represents the model, we know that the model is correct.

Availability of the Intermediate Problem-Solving State. The assumption about the intermediate problem-solving state being available is the most problematic one. In principle it is certainly possible for the problem-solving to keep track of all the intermediate data it generates and from these data reconstruct any intermediate system state.⁶ Clearly this becomes impractical/impossible when the system is very large or when it produces large amounts of intermediate results. It is then a matter of tradeoff between the amount of time it takes to deduce what some intermediate state was, from partial information,

⁶Various time stamps would have to be used to make sure only data created before some time was considered.

and the amount of storage and search time required when all intermediate information is stored. When only partial information is maintained, the issue then becomes what to keep and what to throw away. The problem of diagnosis also changes and becomes more that of devising discriminatory tests for specific faulty parts. This is the focus of much research, exemplified by the work of Genesereth and Davis [15,10].

§6. Contributions of this Work

Recently, causal model based reasoning has become popular; both in diagnosis and in attempts to predict all possible behaviors of complex devices based on the knowledge of their individual parts. This latter application is called envisioning [12]. To our knowledge this idea has not been applied to reasoning about problem-solving systems. We believe this to be a sound approach to reasoning about the behavior of complex systems. The primary contribution of this research is a proof by example⁷ of the feasibility of causal model based diagnosis of problem-solving system behavior.

Our work differs from existing AI diagnostic systems in several aspects. The availability of the intermediate problem-solving state of the DVMT means that we do not have to deal with uncertainty nor do we need complex reasoning mechanisms to determine which of several components is faulty. On the other hand, we do require mechanisms to handle the combinatorics due to the complexity

⁷The example being the construction of the Diagnosis Module.

of the DVMT system. The lack of absolute standards for behavior required the development of a reasoning we call Comparative Reasoning, which involves the comparison of two situations in the DVMT. Comparative Reasoning is described in Chapter IV and the details of how we deal with the combinatorics are described in Chapter V and VI. The details of how this work relates to other AI diagnostic systems are in Chapter VII.

While diagnosis is in itself useful for sophisticated debugging and explanation of system behavior, the real excitement comes when diagnosis is coupled with automatic detection and correction of the system behavior. A problem-system system capable of such behavior would become much more autonomous and robust, being capable of adapting its behavior to changes both in the environment and in its own configuration. Recent work in AI has emphasized the necessity of problem-solving systems having deep models of their tasks in order to behave "intelligently". This theme reappears in different guises in many areas. In natural language dialogue understanding, the system must first construct a model of the participants in the dialogue, before it is able to completely understand what is being said. In intelligent interfaces, the system must construct a model of the user in order to disambiguate commands when necessary. In complex problem-solving systems, it is becoming necessary for the system to take part in its own control. Without having a model of itself, a system cannot properly make the necessary control decisions. This type of self-awareness seems to be one of the main characteristics of intelligent behavior [29,13].

As far as the specifics of causal model construction and diagnosis of problem-

solving systems, the contributions fall into four categories:

1. Developing a formalism for representing the causal model and guidelines for representing systems by such models.
2. Identifying the types of reasoning necessary to diagnose problem-solving system behavior.
3. Handling of cases where no absolute standards for behavior exist and the system must compare some problem situation with a selected situation from the past which seemed to be correct. This model situation then becomes the relative standard for appropriate behavior.
4. Dealing with the combinatorial explosion resulting from the large number of objects that must be considered during diagnosis.

Each of these categories is discussed in more detail below.

Developing the Modeling Formalism. Our modeling formalism captures both the structure of the problem-solving system and its behavior. The structure is represented by modeling the different types of objects in the system and the relationships among them. The function is represented by attaching to each objects the states they can undergo; for example, creation, changes in attributes, or deletion. This type of representation has two advantages. First, the objects and states correspond to easily identifiable components of the modeled system. This makes is relatively easy to decide how to map the system onto the model. Second, the model captures the intuitive understanding of the underlying system. This makes it easier to follow the diagnostic reasoning based on the model.

We have found some principles for constructing similar models for problem-solving systems. These principles are discussed in Chapter VI. Statement of these

rules should make it easier for someone to construct a similar model for their own application. Although we have used this modeling formalism to represent a problem-solving system, we believe that it could also represent physical systems. The model is easily extensible to contain more states when the resolution of the representation is increased. It is more difficult to add new types of objects to the model, because of the numerous relationships among attributes of neighboring objects.

Reasoning Mechanisms. We have found the need for several types of reasoning in order to diagnose the DVMT behavior. Two of them are modifications of backward chaining and forward chaining. We call them Backward Causal Tracing and Forward Causal Tracing, since they construct causal pathways linking states and objects in the system model. Backward Causal Tracing is used to determine how some desired situation in the DVMT could have been achieved. By comparing the resulting causal pathways with what the system actually did do, we are able to determine where the faults occurred. Forward Causal Tracing is used to predict the effect of some situation on future system behavior. This type of reasoning has been called projection or simulation in the planning literature. Both of these types of reasoning are made more complex because the model is hierarchical. Additional control is necessary to decide when to move up or down a level in the hierarchy.

In addition to these, we needed to develop two other reasoning mechanisms. One to handle cases where no absolute standard exists for the system behavior, which requires the diagnostic component itself to select an appropriate situation

in the system behavior with which to compare the problematic situation. We call this type of reasoning **comparative reasoning**, since it involves the comparison of two situations in the problem-solving system. The other type of reasoning mechanisms was required due to the large number of objects that must be considered when diagnosing problem-solving system behavior. In our case, these objects were the many intermediate results in the problem-solving process; i.e., partial descriptions of the map of the environment being generated by the DVMT. In order to handle this combinatorial explosion, we developed mechanisms to represent and reason about classes of objects rather than the individual cases.

Lack of absolute standards. As was mentioned above, when diagnosing the behavior of systems with few absolute standards for correct behavior, it becomes necessary to select some "typical" situation in the system as a model on which to base decisions about the correctness of another situation. This is an example of the "majority rules" practice for deciding whether something is appropriate or not. The mechanism we call comparative reasoning handles this type of a situation. Given a situation to analyze, comparative reasoning must first select another situation to use as the basis for deciding what is wrong and what is right. Currently, the knowledge upon which this situation selection is based is provided by the model builder. It is in a declarative form so that it can be easily modified, should the criteria for correctness change. One of the extensions of this research would be to provide the system with the underlying knowledge that would allow it to decide what the criteria for correctness were. We believe this is an important type of reasoning in problem-solving systems since those

systems often deal with problems that have no predefined or fixed answers. It thus becomes necessary for the system to decide whether it is functioning correctly based on the consensus among the system components rather than some externally provided absolute standard or correctness.

Handling combinatorial problems. The problem of combinatorial explosion plagues most AI tasks. In AI approaches to diagnosis, it has only been discussed with respect to decisions about which diagnostic path to follow. We have faced a slightly different problem: that of handling large numbers of objects to diagnose. An example from the DVMT will illustrate this. In order to determine why the system generated only a partial description of some vehicle track, we must consider the track locations in the possible track extensions. There are eight intermediate levels between the raw signals and the final track. Objects (i.e., partial track descriptions) at each of these must be analyzed during the course of a diagnostic session. Given the structure of the data, the analysis of a vehicle track consisting of eight locations can result in the creation of 153 intermediate objects.⁶ Since a typical environment map may contain several tracks, such large numbers of objects soon make the diagnosis impractical. In order to deal with this problem, we needed to develop some techniques for reducing the combinatorics. Some of these are:

- Grouping together similarly behaving objects and reasoning about the resulting classes of objects, rather than the individual cases.
- Choosing a representative object to diagnose.

⁶The exact number depending on the system signal grammar, as well as the number of locations.

- Allowing the existing data to constrain the diagnosis.

The details of these techniques are discussed in Chapter VI.

§7. A Guide for the Reader

This is a large document and chances are that only a portion is relevant for a particular reader. Below is a brief summary of each chapter as well as the relationships among them. Appendix A contains a glossary of the technical terms used in this dissertation.

Chapter II introduces the DVMT structure in enough detail to enable the reader to follow the diagnosis experiments described in Chapter V and the implementation details described in Chapter VIII. This chapter should be read by anyone wishing to understand the details of the DM structure and operation.

Chapter III describes the structure of the System Behavior Model and should be read by people interested in constructing a similar model for a different problem-solving system.

Chapter IV discusses how the System Behavior Model is used in diagnosis. The different types of reasoning are introduced and discussed in detail.

Chapter V describes in detail two experiments where the DM was successful in diagnosing DVMT failures.

Chapter VI discusses some principles we have learned about modeling problem-solving system behavior. This chapter should be read by those interested in adapting the formalism for a different system. Chapters III and IV should be

read before Chapter VI.

Chapter VII is a literature review of diagnostic systems in AI. It can be read independently of the other chapters, although some of the more subtle points can be better appreciated if the reader understands the structure and use of the System Behavior Model.

Chapter VIII describes the implementation details of the DM. It can be safely skipped unless one wants an in-depth understanding of the DM system details.

Chapter IX is look back at this work and highlights the contributions. Chapter X discusses some directions for future work.

§8. Summary

This chapter provided an overview of the research discussed in this dissertation: the construction of a component of a problem-solving system capable of diagnosing the system's behavior. We first discussed the need for more autonomous problem-solving systems, ones that were capable of reasoning about their own behavior. We then motivated the selection of diagnosis as the focus of this research by explaining how it forms an integral part of the higher-level functions that autonomous systems must display; namely adaptation and fault-tolerance. Diagnosis is also useful by itself, as an aid in debugging the system during development and in explaining its behavior once it is functional.

The problem of diagnosis was defined as the task of identifying the primitive causes responsible for some inappropriate system behavior. In the case of

problem-solving systems, inappropriate behavior can be the lack of an overall solution, an unsatisfactory solution, or inefficient system behavior. We briefly introduced the problem-solving system we will be diagnosing, the Distributed Vehicle Monitoring Testbed, and discussed the types of faults that need to be diagnosed. Namely, hardware failures of components such as sensors or communication channels, and faulty settings of the various parameters responsible for determining the problem-solving control strategies. We call such faulty control parameter settings problem solving control failures. It is these types of failures that are the focus of the diagnosis.

The AI approaches to diagnosis were discussed and the selection of a causal model based diagnosis for problem-solving systems was motivated. The system characteristics such as the availability of the system structure and the intermediate problem-solving states, as well as the large set of possibly faulty behaviors, resulting from inappropriate parameter settings, make the use of a causal model based diagnosis more appropriate than the rival fault dictionary based approach. The more complex a system is, the more manifestations a single failure may have as it propagates through the system behavior pathways. In the DVMT system, a relatively small set of failures can have numerous manifestations and the same symptom can be the result of a combination of several failures. A causal model provides a concise representation of these complex relationships and is therefore well suited for representing problem-solving system behavior.

We have developed a formalism for a causal model of a problem-solving system and have encoded a subset of the possible DVMT behaviors within this formal-

ism. The resulting causal model serves as the knowledge-base for a diagnostic component we have implemented. The architecture of this component was briefly described. Finally, the assumptions underlying this approach to diagnosis and the contributions of this work were discussed.

Chapter II

THE DVMT AND EXAMPLES OF FAILURES

Do not be afraid to abandon your faults.

A Chinese fortune cookie.

This chapter describes the structure of the problem-solving system diagnosed by the Diagnosis Module, the Distributed Vehicle Monitoring Testbed (DVMT). The DVMT characteristics that have influenced our approach are discussed and two example DVMT failures are presented, showing a high-level view of how the Diagnosis Module (DM) works to identify the failures. The detailed description of these examples is in Chapter V.

§1. The Distributed Vehicle Monitoring Testbed

The (DVMT) [23] is a distributed problem-solving system where a number of processors (nodes) cooperate to interpret acoustic signals. The goal of the system is to construct a high-level map of vehicular movement in the sensed environment. Each processor in the DVMT system is based on an extended Hearsay-II architecture where data-directed and goal-directed control are integrated [4]. Figure 7

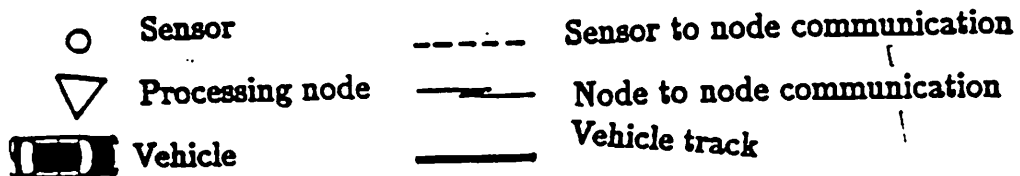
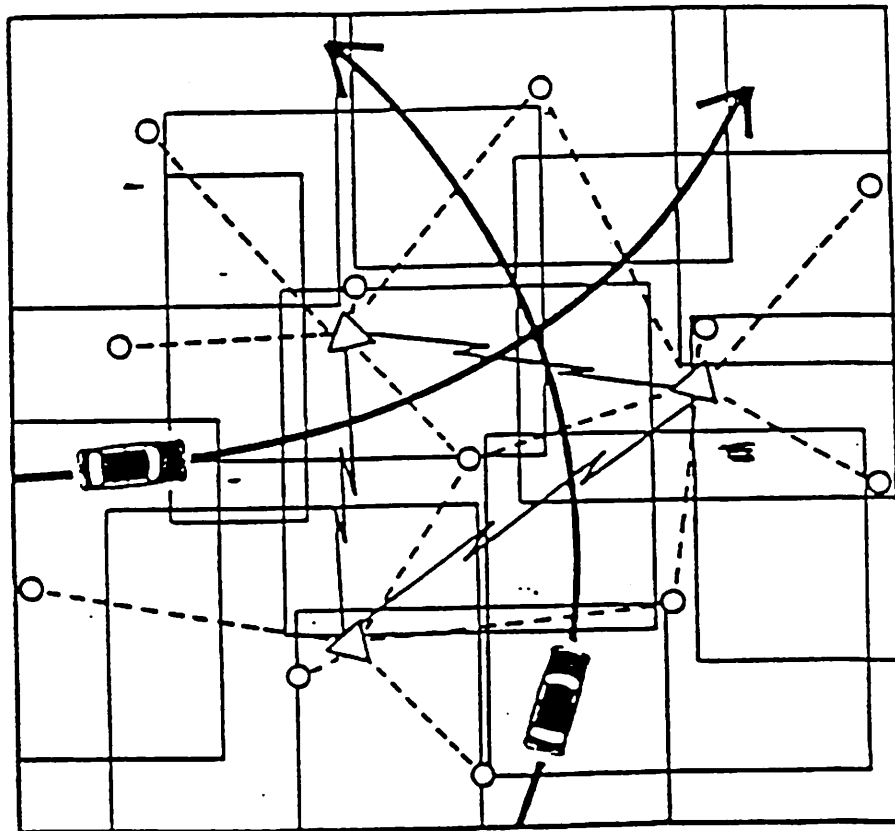


Figure 7: The Distributed Vehicle Monitoring Testbed

The Distributed Vehicle Monitoring Testbed (DVMT) is a distributed problem-solving system whose task is the interpretation of acoustic signals produced by moving vehicles. The DVMT consists of sensors (circles) and processing nodes (triangles). Since each sensor senses only a portion of the environment, the nodes must communicate in order to derive the overall map of vehicular motion.

shows a diagram of the DVMT system and its task. The system will be presented by first describing the task (the interpretation of the acoustic signals), then the architecture and the processing structure of an individual node in the DVMT (goal-directed Hearsay-II architecture), and finally the overall organization of the system.

§1.1 The Task of the DVMT

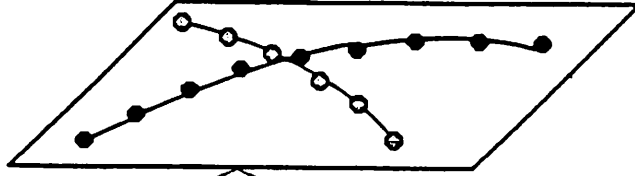
The task of the DVMT system is the interpretation of acoustic signals generated by vehicles moving through the environment. The system senses these signals, combines them appropriately, and produces a map of the environment which identifies the types of vehicles and describes their paths. The raw signals are described by four components: a sensed time, an x-y location, the signal type (event class), and the signal strength (sensed value). These raw signals undergo several types of transformations whose end result is the map describing the vehicular motion. Figure 8 illustrates this process.

In order to generate such a map the DVMT must know:

1. how the individual sensed frequencies combine to identify a particular vehicle and how these vehicles combine into patterns, which move as a unit, and
2. how the individual signals at different times and locations combine to form tracks representing the vehicle paths.

The information about how the individual frequencies are combined to form vehicles and which vehicles form patterns is contained in the system signal gram-

A Map of the Vehicle Motion through the Environment

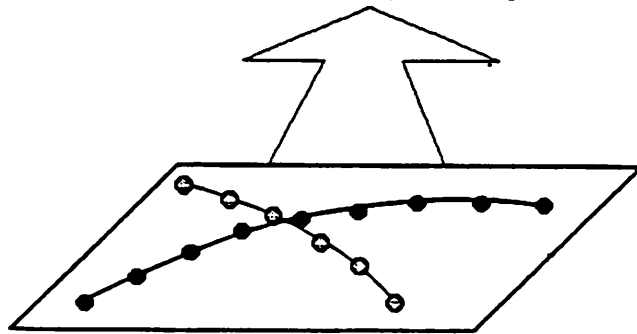


transformation through the levels of abstraction (SL, GL, UL, PL, ST, GT, UT, PT) combining the signal types according to the system signal grammar

combining individual locations into tracks, and track segments into longer tracks at each level of abstraction

$(1(x_1 y_1) \& (2(x_2 y_2))) \text{ ---> } ((1(x_1 y_1)) (2(x_2 y_2)))$

sets of uncorrelated signals specifying TIME and LOCATION
 $(1(x_1 y_1)) (2(x_2 y_2)) (3(x_3 y_3)) \dots$



Vehicles of Different Types Moving through the Environment

Figure 8: A High-Level View of the Data Transformation

This figure illustrates the task of the DVMT: the construction of a map describing the vehicular motion through the environment. The sensors produce sets of uncorrelated signals, describing the time, location, and type of the signal. The DVMT then integrates these signals into sequences representing the motion of the vehicles.

mar. Figure 9 illustrates the structure of the signal grammar. The grammar is a partially ordered graph with four levels and its structure parallels the structure of the data. At the lowest level (signal level) are the individual frequencies sensed as the raw signals, at the **signal** level of the grammar. These are combined into sets of harmonically related signals called groups, at the **group** level. Vehicles are identified by the groups they contain. Groups therefore combine into vehicles, at the **vehicle** level of the grammar. Finally, at the highest level of the grammar, are **patterns**. Patterns consist of two or more vehicles that move through the environment as a unit at some fixed relationship to one another. The patterns can be identified by the types of vehicles they contain. Individual vehicles are thus combined into fixed patterns, where the whole pattern moves along the same path. **We thus have four levels of abstraction: signal, group, vehicle, and pattern.**

In addition to this vertical data transformation, the data undergoes another type of transformation: the aggregation of the individual locations into tracks representing the vehicle paths. This aggregation can occur at any of the four levels of abstraction described above. In order for locations to be combined into a track they must be of similar signal type (signal types are called event classes in the DVMT) and their time and spatial relationship must be consistent with the velocity and acceleration constraints for the different vehicle types.

We thus have eight types of data in the DVMT system: signal location (sl) and signal track (st), group location (gl) and group track (gt), vehicle location (vl) and vehicle track (vt), and finally pattern location (pl) and pattern track

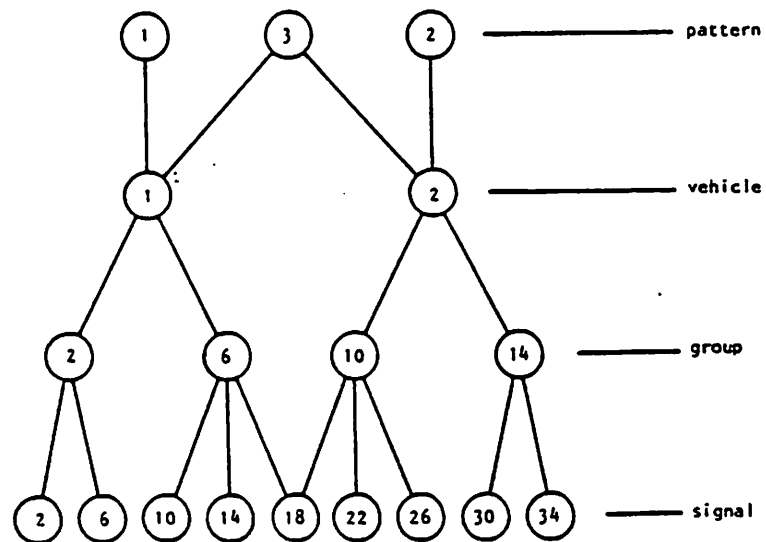


Figure 9: System Signal Grammar

A simple, three pattern grammar, specifying the relationship among the event classes at the various of the data hierarchy. There are two vehicles types (1 and 2), each formed from two different signal groups. Each vehicle can form a single vehicle pattern (1 and 2) or both together (at a specific distance not illustrated here) can form a two-vehicle pattern (3).

(pt). The data comes in at the signal location level and the output of the DVMT system is a map of the environment representing the data at the pattern track level. The input data thus undergoes two types of transformation.

- vertically from the lowest level signals, through the group and vehicle levels, up to the pattern level, and
- horizontally, within each level, aggregating the individual locations to form longer tracks.

Figure 10 illustrates this process. Later we will see how some of the DVMT data structures (the blackboards) parallel this structure of the data transformation.

§1.2 Node Architecture

The task of each node is to process all data available to it by applying its knowledge sources to perform the data transformations. The knowledge sources use the system signal grammar and their knowledge of how signals combine into tracks to perform local (within a node) data transformations. There are several knowledge source types, each responsible for a different type of data transformation or for communication of partial results among the nodes. The node's data can either be available locally, from the node's sensors, or can be received from other nodes.

All dynamic domain data (i.e., information about the vehicle movement) in the system is represented by data structures called **hypotheses**, both the initial sensor inputs and any partial results produced by the node's knowledge sources (see Figure 11, part A). The hypotheses are stored on a **data blackboard**. A

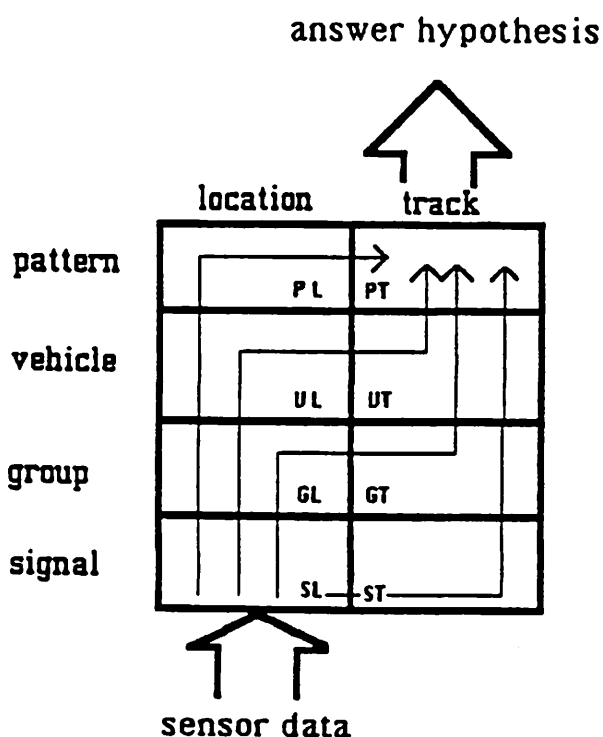


Figure 10: Graph of the Data Transformation

The sensors produce data at the signal location level. The aim of the DVMT is to produce correlated pattern tracks (answer hypothesis). The arrows show four pathways in which the answer hypothesis can be derived. The DVMT interest area parameters control which combination of these pathways is used to derive the answer hypothesis.

blackboard is a shared-memory structure among the different knowledge sources within a node. Each node has its own set of blackboards. There is no centralized blackboard for the entire DVMT system. The structure of the data blackboard parallels the structure of the data transformations (see Figure 10).

The creation of a hypothesis stimulates the creation of a goal (see Figure 11, part B). A goal represents the set of hypotheses which may be created by transforming the one that stimulated the creation of the goal. Goals thus represent predictions of how existing data will be extended. They can be thought of as tasks to be achieved (goals to be satisfied), using the stimulus hypothesis, as well as any other necessary hypotheses, as the data. Goals are stored on a **goal blackboard** whose structure parallels the structure of the data transformations; the goal blackboard levels are the same as the data blackboard levels.

Once a goal has been created, it stimulates the scheduling of a knowledge source. A successful scheduling of a knowledge source results in the creation of a **knowledge source instantiation (ksi)** which assigned a rating and is inserted onto the ksi scheduling queue. Part C of Figure 11 shows an example ksi. A knowledge source instantiation can be thought of as a process, which is scheduled when the necessary data exists. A knowledge source takes its input from and puts its output on the data blackboard, in the form of hypotheses.

The knowledge source scheduling is controlled by the planner component, which controls the ksi scheduling and execution at each node. The planner uses parameters called **interest areas** to determine the ratings of goals and knowledge source instantiations and thus to control which data will be worked on.

Structure of a DVMT Hypothesis Object

time-location-list ((1 (1 1)) (2 (2 2)) (3 (3 3)) (4 (4 4)))
 event-class 1
 level vt
 rating 4500
 stimulated-goals (g0010 g0012 g0013)
 stimulated-ksis (ksi0011 ksi0024)
 satisfied-goals (g0002 g0004)

PART A

Structure of a DVMT Goal Object

active-time-region-list (4 (3 3 7 7))
 inactive-time-region-list ((1 (0 0 4 4)) (2 (1 1 5 5)) (3 (2 2 6 6)))
 level vt
 event-classes (1 2 3)
 stimulus-hyps (h0003 h0005)
 stimulated-ksis (ksi0030 ksi0031)
 rating 3000

PART B

Structure of a DVMT Ksi Object

ks-type merge:vt
 stimulus-hyps (h0020 h0021)
 stimulus-goals (g0025 g0028)
 output-hyps (h0045 h0046 h0047)
 rating 7050

PART C

Figure 11: The Structure of Objects in the DVMT System
This figure shows the structure of three objects from the DVMT system. Only a subset of the object attributes is shown.

There are two types of interest areas: the local interest areas and the communication areas. Local interest areas determine which of the possible data transformation paths seen in Figure 10 the node will take. (Both the local and the communication interest areas must be consistent with the set of knowledge sources at a node. In fact, a common error the DM can diagnose is a mismatch among the two. For example, a node's local interest area might indicate a transformation from *sl* to *gl* but there is no knowledge source that can perform such transformation.) Communication interest areas control the communication among the nodes in the DVMT.

The interest area parameters specify weights for different areas of the environment. They allow the independent specification of times and regions in the environment, levels on the blackboard, and event classes. In the case of the communication areas, they also allow the specification of other nodes in the system. The weights associated with each of the regions in this time/level/event-class/node-space control how the data within that region will be weighted. This allows the planner to control processing by giving some area a lower or higher weight. The knowledge sources working on data in a more highly rated area will be rated higher and will thus have a greater chance of executing.

Processing at each node thus involves the creation and manipulation of three different types of objects: the hypotheses (representing the data and stored on the data blackboard), the goals (representing the predictions of how the data will be processed further and stored on the goal blackboard), and knowledge source instantiations (represented by *ksi*'s and stored on the *ksi* scheduling queues).

Each of these objects has a rating. In the case of hypotheses, this rating is called the belief, to indicate the system's confidence that this hypotheses represents a correct partial interpretation of the environment.

Summary of the Basic Node Processing Cycle. The execution of a knowledge source instantiation and the associated activities comprise the basic problem-solving cycle in the DVMT system. The cycle begins with the execution of the highest rated knowledge-source instantiation on the scheduling queue. The knowledge-source runs and produces one or more hypotheses. The hypotheses are inserted onto the data blackboard. Hypotheses generate goals which represent predictions about how the hypotheses can be extended. A hypothesis together with a goal triggers the scheduling of a knowledge source whose execution will satisfy the prediction (goal) by producing a more encompassing hypothesis (one which includes more information about the vehicle motion). It is the planner component of the DVMT system which controls the creation and rating of the goals and the instantiation, rating, scheduling, and execution of the knowledge sources. Figure 12 illustrates the processing structure at each node.

This constitutes one ksi execution cycle for the node. The next cycle begins by executing the highest rated ksi from the queue, inserting the hypotheses that are produced, creating the appropriate goals, and scheduling any relevant knowledge sources. This process begins with the input data and repeats until a complete map of the environment is generated or until there are no more knowledge source instantiations to invoke.

Additional issues the planner deals with are subgoaling of high level goals and

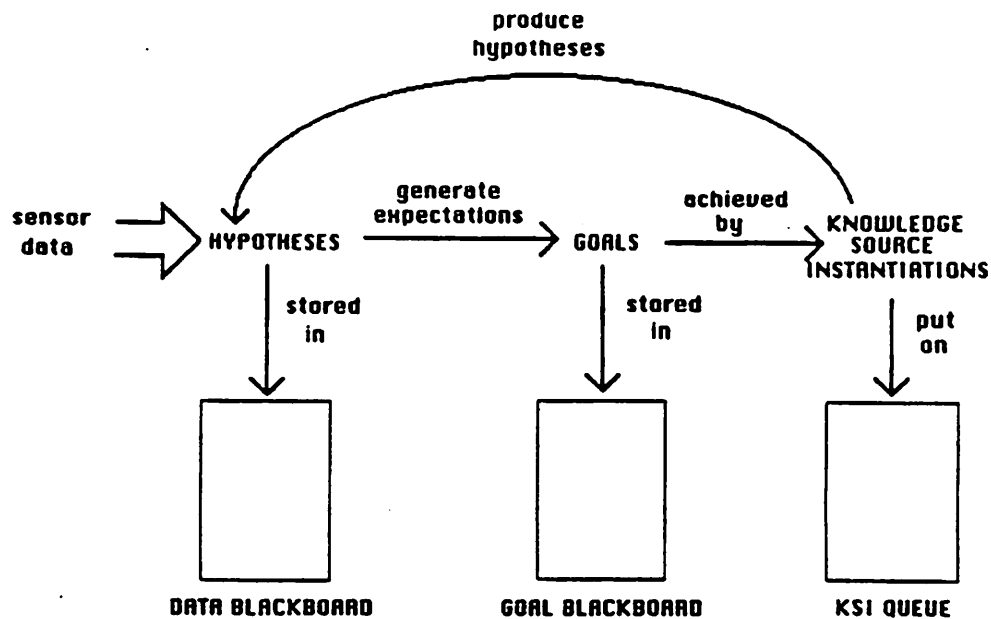


Figure 12: The Structure of Processing at Each Node in the DVMT System

The DVMT begins the interpretation task with the arrival of the sensed data. All data is represented by hypotheses objects and stored on the data blackboard. The arrival of a hypothesis stimulates the creation of a goal. The goal represents a prediction of how the hypothesis might be extended in the future. A hypothesis together with a goal stimulate the instantiation of a knowledge source (ksi). Each such instantiation is rated and if this rating is high enough the ksi is inserted onto the scheduling queue. At the beginning of each system cycle the highest rated ksi executes and produces additional hypotheses. This cycle repeats until the final answer is derived or until there is no more data to work on.

goal satisfaction. Since these aspects of the planner are not represented in our model of the DVMT system, they will not be discussed here. Corkill [5] discusses the details of the planner and Lesser and Corkill [23] discuss the details of the DVMT system.

§1.3 DVMT System Organization

The DVMT system is a distributed problem-solving system. It consists of a number of spatially distributed processors, called nodes. Each node is a sophisticated problem-solving system capable of solving the entire task by itself, given the necessary knowledge and data. However, it may be more efficient to distribute the task among the various nodes. The way the task is divided and allocated to the individual nodes imposes a particular organization of the nodes in the DVMT system. This organization is expressed by the parameters that control communication among the nodes (both who communicates with whom and what is communicated), data allocation (which nodes get data from which sensors), and task allocation (which nodes process what data). There is a large number of possible organizations. The *raison d'être* of the DVMT is to experiment with various organizations of the individual nodes and to determine a suitable organization for a particular type of task. These issues are discussed in more detail elsewhere [5].

The entire system is parametrized so that any of the following attributes can be varied:

- the structure of the data represented by the system signal grammar;

- number and location of nodes;
- number, location, and type of sensors of each node;
- type and quality of the knowledge available at each node;
- answer derivation pathway for each node;
- communication protocols among the nodes;
- balance between local and external control for each node.

These numerous parameters make the DVMT a rich environment for experimenting with different types of tasks and different system architectures, in order to evaluate how to match them up. These parameters can also be a source of error, when they are not set consistently. Such errors are the problem solving control failures the DM focuses on.

It is important to keep things straight at this point. Note that there are two problem-solving systems: the DM, whose task domain is the diagnosis of the DVMT system, and the DVMT system, whose task domain is acoustic signal interpretation. Figure 13 illustrates the relationship between these two systems.

§2. Unique Characteristics of the DVMT System

Two characteristics of the DVMT system played a crucial role in determining our approach to modeling and reasoning about its behavior. This section describes these characteristics and discusses their implications for the DM design.

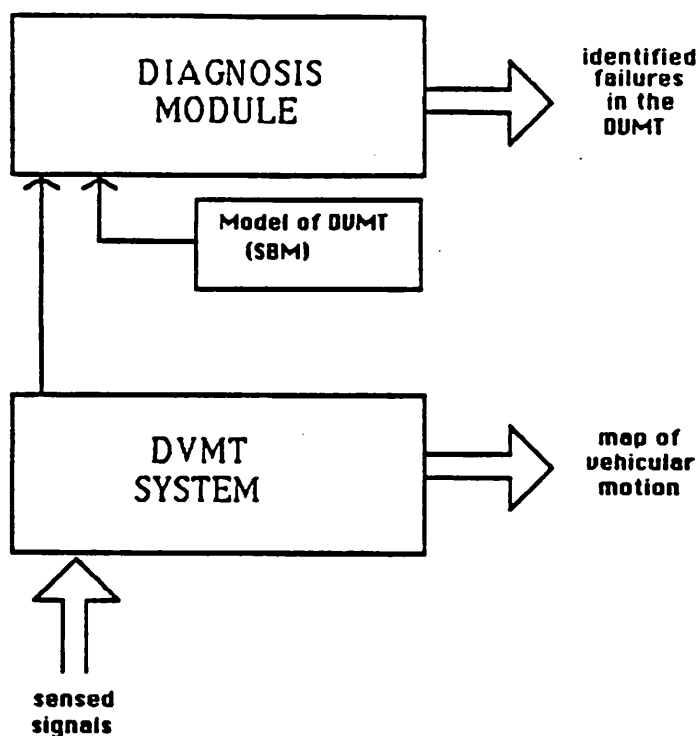


Figure 13: Relationship between DVMT and DM Systems and Their Domains

The Diagnosis Module receives as input symptoms representing inappropriate DVMT behavior. It uses a model of the DVMT as well as the DVMT data structures to diagnose what caused the observed symptom. The Diagnosis Module outputs the identified faults responsible for the symptom.

§2.1 The Complexity of the DVMT System

The characteristic that dominated our approach was the sheer complexity of the DVMT system. The use of a causal model for diagnosis is partly a consequence of this complexity. In a simpler system, the use of a fault dictionary for diagnosis is feasible. The fault-dictionary approach uses a set of pre-compiled symptom \Rightarrow fault associations (the fault dictionary) to perform both detection and diagnosis. Detection is done by matching observed states in the system with symptoms in the dictionary. Diagnosis is done by reading off the fault part of each symptom \Rightarrow fault rule. If the table size is small, this method of detection/diagnosis is quite efficient. In fact, experienced diagnosticians use this method for "every-day" cases. This technique works well as long as:

- The system fails in known ways, allowing the compilation of the fault dictionary.
- The number of failures is small, so the table size is manageable.
- A given symptom always maps to a very small number of faults, so the search during diagnosis is manageable.

In the case of the DVMT system none of the above conditions hold. Because of the system complexity, the number of symptoms is extremely large. The number of parameters required for system control is large and the number of wrong combinations of those parameter settings is much larger. It would be impractical to try and map all the failures resulting from all the possible wrong combinations of parameter settings. A causal model seems a more robust as well as a more practical approach to diagnosis in complex systems because it does not require

the compilation of possible failures. In addition, there are fringe benefits associated with the use of a causal model. The knowledge representation is much more structured than in a fault dictionary. The model represents an abstraction of the actual system and as such can be used for other applications beside diagnosis such as simulation or explanation of the system's behavior.

The complexity of the DVMT system precluded representing the entire system in the model. The problem of devising a concise representation of a large set of possible system behaviors led to various types of abstractions, both in the model construction and in the use of the instantiated model. These abstractions allow us to represent and reason about classes of objects rather than individual cases. Chapter VI discusses in detail the types of abstractions we have made in order to be able to represent and reason about a complex system using a relatively small model.

§2.2 Availability of the Intermediate DVMT States

The DVMT system maintains a number of data structures (blackboards, scheduling queues) that contain its recent history. **We exploit this availability of the system's intermediate states in constructing an instantiation of the system model to represent how a particular situation was reached.** No objects created by the system are ever deleted. Thus all the hypotheses, goals, and knowledge source instantiations are kept in the system data structures. All these objects have a time attribute, and we can therefore reconstruct the state of problem-solving at any time in the past. This means that we can trace back

through the system history records and reconstruct the derivation of any object. This has implications for the diagnosis. Diagnosis is done by comparing the actual behavior with the desired behavior. Starting with some unexpected result detected by the detection component, the diagnosis component tries to reconstruct how that result came about. During this process it must determine the intermediate steps that led up to the final result. In most diagnostic systems the results of these steps must be recomputed (either by probes into the real system or by simulation).¹ The DM, because of the availability of the intermediate results, simply looks them up in the system data structures. This saves processing but does require lot of storage space. Having the intermediate state available has allowed us to concentrate on different types of diagnostic reasoning, such as Comparative Reasoning and reasoning about classes of objects, which has not been explored in AI diagnostic systems.

§3. Examples of DVMT Failures DM Can Diagnose

This section presents two examples where either a simulated hardware failure or a faulty parameter setting caused problems in the DVMT system. In both examples the problem that initiates diagnosis is the lack of a spanning pattern track (pt) hypothesis; i.e., a hypothesis that describes a pattern track starting and ending at the boundaries of the node's sensed area. This is a good starting symptom because we know that a node should always derive such a spanning

¹Chapter VII discusses this in more detail.

hypothesis if allowed by its control parameters. Only a high level description of the types of reasoning required is given here; a detailed description of the diagnosis is in Chapter V.

§3.1 Example I: Four-Node System without Communication Knowledge Sources

In this example, the DVMT system is configured such that its four nodes are distributed over the sensed area (see Figure 14). Each node receives data from its part of the environment and the nodes must therefore communicate partial results (vt hypotheses) so that each node can derive the final answer; a pt hypothesis that spans the entire sensed area. The system never does derive this spanning pt hypothesis due to the faulty settings of the parameters controlling communication. Two communication knowledge sources (cks's) are responsible for communication, but neither can be scheduled because both are missing from the knowledge source (ks) set of the nodes. We would like for the Diagnosis Module to trace the lack of the spanning pt hypothesis to the missing cks's. Once these failures have been identified, we would like the DM to simulate their effects on future system behavior and predict that no vt hypotheses will be communicated as a result.

The diagnosis begins with the lack of a spanning pt hypothesis. In a fully implemented Fault Detection/Diagnosis system this symptom would be detected by the detection component. Since this component has not been implemented, the diagnosis is initiated by giving the symptom to the DM as data. In order to make the problem easier, we begin with the actual correct answer rather than with a

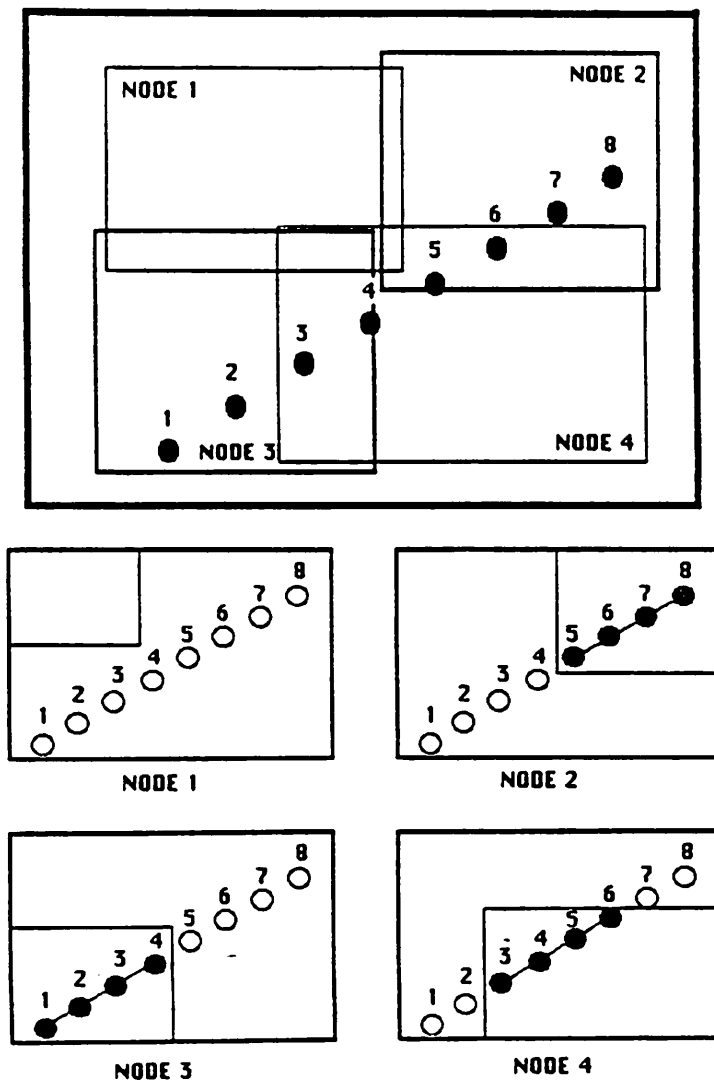


Figure 14: Scenario for Example I

The upper part of the figure shows the overall view of the four nodes and the data. The lower part shows the pattern track segments derived at each of the individual nodes.

description of a spanning pt hypothesis. Beginning with the correct answer does not in fact avoid any difficult problems. If we were to begin with a description of a spanning hypothesis, the DM would first examine the DVMT data blackboard to find any hypotheses which could be extended into a spanning hypotheses; in other words to find all pattern track segments. A subset of these would be selected for further diagnosis in order to find why they were not extended into spanning tracks. As will be seen later, this is exactly what is done when tracing backward from any track state. By starting the diagnosis with the correct answer, we are simply avoiding doing this process twice. Diagnosis is done separately for each of the four nodes in the DVMT. When one node reaches a problem that needs to be diagnosed by another node, it saves it as a pending symptom for later diagnosis. These pending symptoms are then analyzed after the diagnosis at all four nodes is completed.

In each node, the diagnosis begins with the lack of a correct pt hypothesis, in this case a pattern track hypothesis integrating locations 1 through 8. The DM has a model of how this hypothesis is derived from shorter pt segments and from hypotheses at the vt level. The DM works by tracing back through this model trying to find a place where the correct processing stopped. In this case the DM quickly discovers that either no data is available locally (as is the case in Node #1), or that only partial data exists, allowing the derivation of only a segment of the desired pt track 1-8. By examining the interest area control parameters the DM determines that in each case the missing data was to be received as partial results (in the form of vt hypotheses) from other nodes in the system. The DM

therefore tries to determine why the missing track segments were not received from the neighboring nodes.

To do this, it traces back through the communication model, which represents message transmission among nodes. There could be several reasons why a hypothesis was not communicated properly: the receiving node might not accept it, the sending node might not send it, the communication channel might be faulty, or the hypothesis might not exist in the sending node. In this case, the DM narrows the problem down to the sending nodes which never did send the expected hypotheses. A break has thus been found in the correct processing and the DM must now determine why the hypotheses were never sent by the other nodes.

These situations are examples of intermediate symptoms which are saved for later diagnosis from the perspective of the sending nodes. After diagnosing all four nodes and determining that the symptoms are all due to the lack of communication, the DM begins to analyze the pending symptoms at each node. The diagnosis of these symptoms reveals that the knowledge sources that were responsible for communicating the hypotheses among the nodes are missing in each node. This then constitutes the identified failures. Since these are faulty parameter settings, they are examples of problem-solving control failures.

Having identified a set of failures, the DM could continue as before, diagnosing each symptom in turn. This would often lead to the repeated identification of the same failures. It would therefore be more efficient to propagate the effects of the identified failures forward through the system's causal model and account for any pending symptoms as they are reached. The next step therefore is to simulate the

effect of the missing communication ks on future system behavior. In this case, the effect is that no vt hypotheses will be sent among any of the nodes. This fault simulation saves a considerable amount of time in future diagnosis both because it accounts for symptoms still pending diagnosis and because it also accounts for any such symptoms that will be detected in the future.

§3.2 Example II: Single-Node/Two-Sensor System with a Failed Sensor

This example presents a scenario in the DVMT system where the system is receiving data from two input sources; sensors A and B. Figure 15 illustrates this situation. The two sensors overlap so some data is sensed by both. In the overlapping area the sensor weight parameter for sensor A is set low because the system is less confident about the sensor's accuracy in that area. The sensor weight parameter of sensor B is high. This results in the data from sensor B being rated high and the data produced by sensor A in the same area being rated low.

In the example scenario the supposedly reliable source of data for the particular task (sensor B) does not in fact generate reliable data because it is malfunctioning. Not only does it not sense the existing data but it generates short noise segments which cannot be incorporated into a single track. Since sensor B's sensor weight parameter is high, these short noise segments produce highly rated hypotheses which stimulate the scheduling of highly rated knowledge source instantiations (ksi's). The correct data is still sensed by sensor A but because

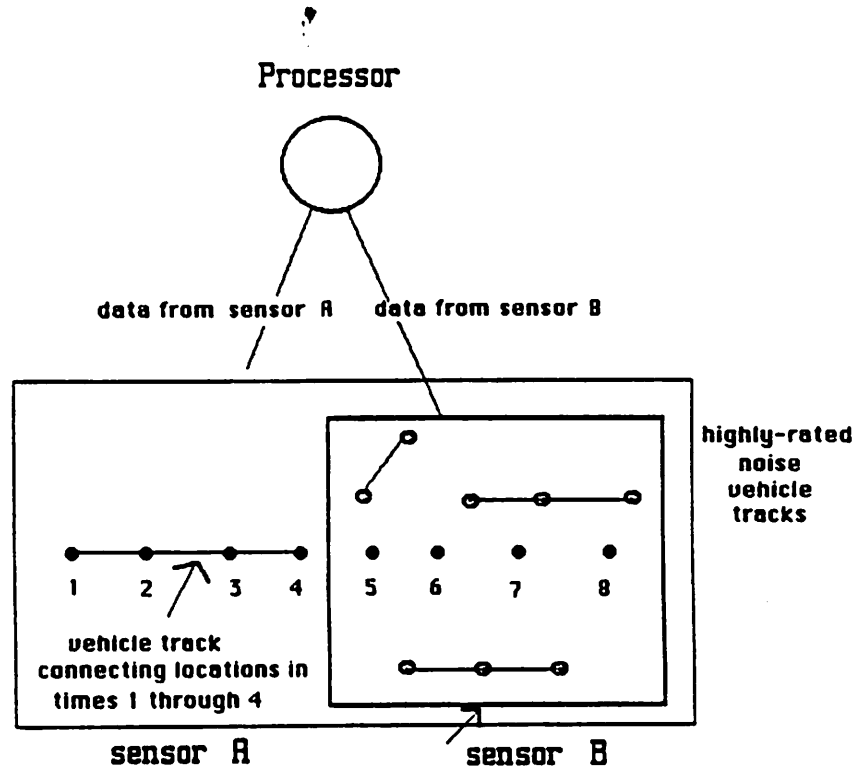


Figure 15: Scenario for Example II.

This figure illustrates the DVMT system configuration and the data for Example II. A vehicle is moving from left to right, producing signals in locations 1 through 8. Sensor A senses all signals, but due to the sensor weight parameter setting, senses locations 1-4 strongly and locations 5-8 weakly. Sensor B is malfunctioning and is not sensing the correct signals at all. Instead, it is generating highly-rated noise signals. It is the ksi's resulting from these noise signals that prevent the low rated "correct" ksi's from executing.

of its low sensor weight parameter, this data produces low rated hypotheses, which stimulate low rated ksi's. The combination of these events results in the scheduling queue being swamped by the highly rated noise ksi's which prevent the execution of the low rated knowledge source instantiations that would produce the correct hypotheses. The aim of the diagnosis is to:

- Determine that it is the low sensor weight for sensor A which is responsible for the low rated ksi's that would extend the correct track.
- Detect the fact that the data produced by the two sensors in the overlapping area does not agree. This will initiate further diagnosis, which will eventually determine which of the two sensors is faulty, by finer analysis of the data produced by each of the sensors.

The exact scenario is as follows. A vehicle is moving through the monitored area from left to right, generating signals at locations 1 through 8. Sensor A senses all 8 locations but, due to the low sensor weight parameter, locations 5 through 8 are weighted lower. Sensor B is not sensing the signals and is generating highly rated noise segments. The DVMT system generates a pt hypothesis connecting locations 1 through 4 based on the strong data from sensor A. It cannot however extend this partial track because the ksi's that would produce the necessary hypotheses are rated low, due to the low sensor weight parameter. The queue is swamped by the highly rated ksi's stimulated by the noise segments produced by sensor B. As a result, the low rated correct ksi's do not execute. The DVMT system is thus diverted from processing the correct data by the highly rated noise whose processing is delaying the execution of the low rated ksi's which would extend the correct pattern track hypothesis.

Diagnosis begins, as before, with the lack of a spanning pt hypothesis. The DM discovers that part of the hypothesis does exist (pt 1-4), and part is missing (pt 5-8). The DM first tries to determine why the missing segment was not created. It traces back through the steps the DVMT system should have taken to derive the pt 5-8 and discovers that the necessary locations (5 through 8) do exist at the group location (gl) level but are never driven up to the next level, vehicle location (vl). Further analysis reveals that the knowledge sources that would drive the gl hypotheses up to the vehicle level have been scheduled but have not yet executed because their rating is too low.

The next task for the DM is to determine why the rating was so low. To do this, the DM uses a model of how the knowledge source instantiation rating is derived from the rating of the knowledge source, the knowledge source's data, and the goal that stimulated the ksi's scheduling. Determining the reason for an abnormal rating presents some difficulty however, because we are dealing with relative quantities here. There is no such thing as an absolute low rating since it is the highest rated knowledge source instantiation on the queue that executes. In order to determine whether some rating is high or low, we must compare it to the other ratings in the system. This is an example requiring Comparative Reasoning, which is discussed in detail in Chapter IV. Comparative Reasoning is a heuristic used to find the causes of a problem by comparing the problem situation with a correct situation. Since such a correct situation (for example, an absolute standard for ksi rating) cannot be set a priori, the DM must find it automatically during diagnosis. In the DVMT, comparing a problematic situation

with a correct situation translates to a comparison of a low rated object with a highly rated object and Comparative Reasoning (CR) is invoked anytime some problem is traced to an object that is rated too low. In tracing the derivation of the rating of each object, CR often uncovers an error.

Since CR uses normally rated object to understand why the problem object was rated low, the first step is the selection of such a model object. In this example, it is the selection of a highly rated ksi. In comparing how the ratings of the two ksi's differ, the DM discovers that the group location hypothesis of the problem ksi is rated lower than the corresponding hypothesis of the model ksi. The DM continues Comparative Reasoning and traces back through the derivation path of the gl hypothesis, eventually arriving at the sensor weight parameter and the data signal. The cause of the low rated gl hypothesis is identified as the low sensor weight parameter.

One cause of the initial symptom has thus been found. What remains to be done is to determine that sensor B is malfunctioning. The DM first notices a problem when it tries to determine the value (signal strength) for the data signal in location 5. Since this value cannot be determined directly from the environment it must be derived from the surrounding states, by finding a signal strength consistent with the sensor weight parameter and the sensed value. Since location 5 should be sensed by both sensors A and B but is only sensed by A, a consistent value cannot be determined. This is an example of the use of internal consistency criteria to detect a problem. Anytime an internal inconsistency is detected, all of the inconsistent values are suspect and another method must be

used to determine what the problem is. In this case, both sensors are suspect and a model of an ideally behaving sensor must be used to decide which of the two candidate sensors is in fact faulty. This is done by analyzing the data produced by each of the sensors and seeing which one is closer to data produced by an ideally behaving sensor. Such analysis would reveal that it is in fact sensor B that is the more likely candidate, since it produces uncorrelated data. (This type of reasoning has not yet been implemented and currently the DM stops by reporting the uncovered inconsistency.)

§4. Summary

This chapter described the Distributed Vehicle Monitoring Testbed (DVMT). The DVMT is the problem-solving system being diagnosed by the Diagnosis Module (DM). The DVMT is a distributed problem-solving system whose task is the interpretation of acoustic signals generated by vehicles moving through the environment. The goal of the DVMT system is to produce a map of the sensed environment, identifying each vehicle and describing its path. The DVMT system consists of a number sensors and processing nodes. Since each node typically receives sensed data from only a portion of the environment, inter-node communication is necessary in order to produce an overall map of the environment. The architecture of each node is a modified Hearsay-II architecture, extended to integrate both data-directed and goal-directed processing.

Two characteristics of the DVMT system played a major role in determining

our approach to modeling and reasoning about its behavior. One was the **complexity of the system** and the other was the fact that the **DVMT maintains a detailed history of its processing**. The complexity of the system required the use of a causal model rather than a fault dictionary as the approach to diagnosing the system failures. The system complexity also necessitated abstraction techniques, both in the model construction and in the model use. These are discussed in detail in Chapter VI. The availability of the intermediate system state allowed us to concentrate on types of reasoning normally not seen in diagnostic systems, specifically Comparative Reasoning and reasoning about classes of objects. Most diagnostic systems view their task domains as black-boxes, where only the inputs and the outputs are observable. The intermediate states must be derived by simulating the system behavior and by applying complex testing procedures. We were able to avoid this by having the intermediate state available in the DVMT data structures in the form of partial results.

Finally, we have motivated the need to the DM and outlined the types of reasoning it can do by discussing two examples of problems in the DVMT system. In the first example a four-node system does not derive the correct answer due to missing communication knowledge sources. The DM identifies this problem and then propagates the effects of this problem forward, in order to determine that the system will not be able to communicate any hypotheses. This involves reasoning about a class of objects rather than about the individual cases. In the second example a two-sensor/one-node system does not derive the correct answer due to the combination of a malfunctioning sensor and a low sensor weight parameter.

The DM first identifies the low sensor weight as one reason for the problem and then determines that one of the two sensors must be malfunctioning.

Chapter III

MODEL STRUCTURE

Oh knowledge ill-inhabited, worse than Jove in a thatched house.

- William Shakespeare, in *As you like it*

This chapter describes in detail the formalism for representing the DVMT system behavior in the form of a causal model. The first five sections focus on the components of the model: state transition diagrams, abstracted objects, constraint expressions among the objects, and state-object links. The last section describes how the model is instantiated to represent a particular situation in the DVMT system.

§1. Introduction

In the first chapter we have motivated the use of a causal model in diagnosing the DVMT system behavior. The main reasons were:

- The complexity of the system precluded the construction and use of a fault dictionary approach, traditionally implemented using sets of unstructured rules representing the empirical associations among symptoms and faults.

- A desire for a representation that would support diverse reasoning mechanisms.
- An interest in what types of knowledge and reasoning are necessary to analyze problem-solving system behavior.

This chapter presents a high level overview of the modeling formalism used to represent the possible behaviors of the DVMT system. We have encoded a subset of the DVMT system behavior within this formalism. The resulting System Behavior Model (SBM) contains knowledge in a form that can be used in many different ways. Unlike the traditional fault-dictionary approaches to diagnosis, the SBM represents causal relationships among events in the DVMT system. This makes it possible to use it not only for diagnosis but also for simulation of the system behavior. Unlike some other causal models, such as CASNET [39], which represent causal relationships among pathological states, our model represents the normal system behavior. The errors are represented as deviations from the expected situations that were not achieved by the system. The model can thus reason both about the causal sequences of expected events, and thereby simulate the correct system behavior, and about the sequences of abnormal events, and thereby diagnose faulty system behavior. The mechanisms we use to reason about the DVMT behavior make extensive use of the ability of the model to support the simulation of the expected system behavior. These mechanisms are discussed in Chapter IV. Although we have not implemented the Detection Module, we believe that the SBM is well suited for this purpose. Some thoughts about the detection are in Appendix C.

The SBM consists of four major components:

HIERARCHICAL STATE TRANSITION DIAGRAMS which

represent the possible system behaviors at various levels of detail. The causal relationships among events in the system are represented as sequences of states in the model.

ABSTRACTED OBJECTS which represent individual objects

(i.e., data structures) or classes of objects in the DVMT system.

CONSTRAINT EXPRESSIONS among the different attributes of the abstracted objects which represent the relationships among the objects.

OBJECT-STATE LINKS which connect the state transition diagrams to the abstracted objects.

Figure 16 illustrates a high level view of the modeling formalism and its relationship to the DVMT system. We can view the model as two parallel networks. At the higher level is the state transition diagram, consisting of states and directed state transition arcs. At the lower level is the network consisting of the abstracted objects whose attributes are linked by the constraint expressions that capture the relationships among the object attributes. The two networks are connected by state-object links. The rest of this chapter describes the SBM structure in detail. Section 6 briefly describes how the SBM is instantiated and serves as an introduction to the description of the model use in the next chapter. The last section of this chapter is a summary that highlights the points necessary to understand the subsequent chapters, which describe how the model is used to diagnose problems in the DVMT system. Figure 17 contains the legend for the figures in this dissertation.

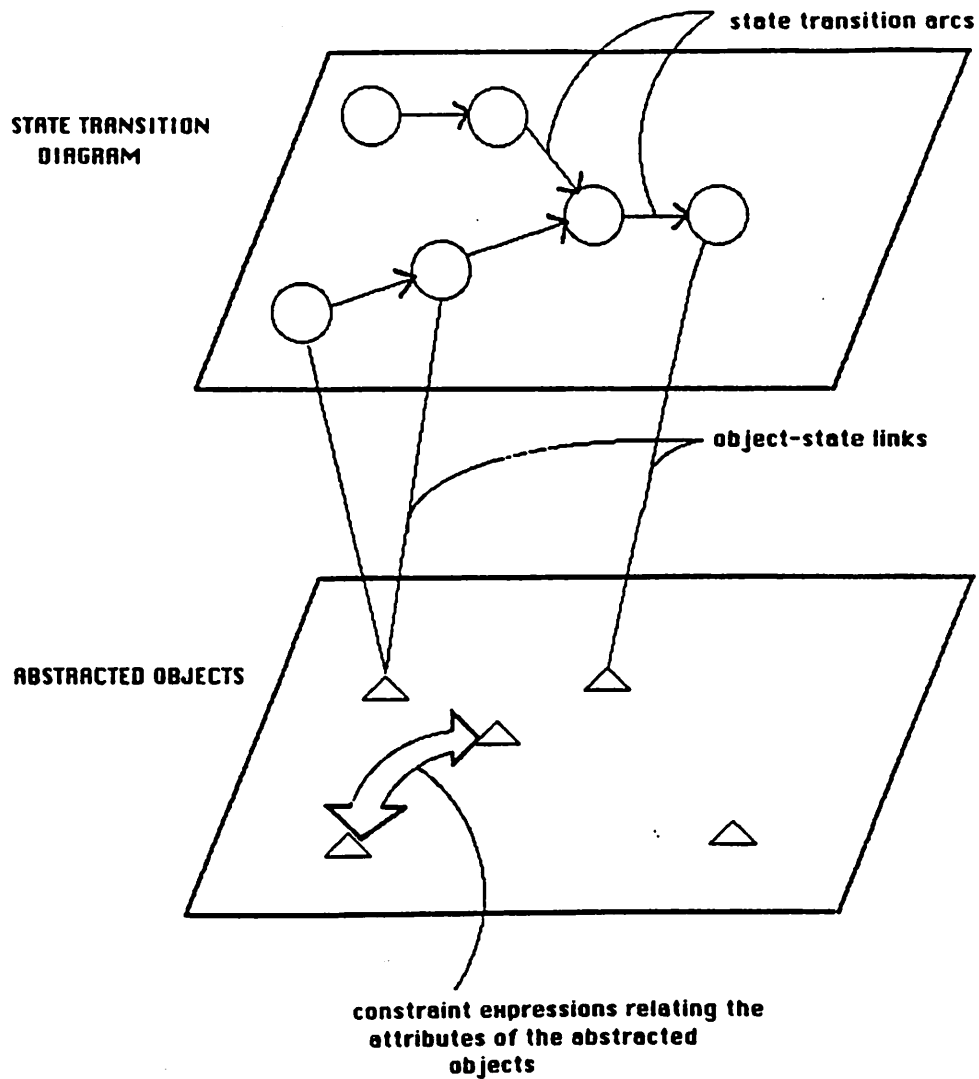














Figure 16: A High Level View of the Modeling Formalism
The figure represents the three parts of the modeling formalism: the state transition diagram, the abstracted objects, and the constraint expressions relating the objects to one another.

LEGEND

<p> STATE (uninstantiated)</p> <p> UNDERCONSTRAINED STATE</p> <p> UNDERCONSTRAINED ABSTRACTED OBJECT</p> <p> ABSTRACTED OBJECT</p> <p> STATE TRANSITION ARC</p> <p> NON EXPANDABLE STATE</p>	<p> TRUE STATE</p> <p> PRIMITIVE STATE</p> <p> COMPARATIVE STATE</p> <p> MERGED STATE</p> <p> FALSE STATE</p> <p> SYMPTOM STATE</p>
---	--

RELATIONSHIP STATES




-  GREATER-THAN its parallel state
 -  LESS-THAN its parallel state
 -  EQUAL to its parallel state
- (in cases where there are multiple parallel states these signs may be associated with the arcs linking the parallel states)

Figure 17: Legend for Figures in the Dissertation

§2. Hierarchical State Transition Diagrams

The state transition diagram component of the SBM represents the sequence of states the DVMT system is expected to follow when it is behaving correctly. Correct behavior here means going through the expected series of steps in deriving the final pattern track (pt) answer from the input sensor data. The DVMT system behavior consists of a series of events. Each event results in the creation of an object (e.g., hypothesis, goal, or knowledge source instantiation) or in the modification of the attributes of some existing object. **The states in the SBM represent the results of such events in the DVMT system.** In addition to representing the outcome of a specific event, **the states in the SBM may represent some arbitrary situation in the DVMT system** which needs to be made explicit so that the Diagnosis Module (DM) can reason about it. These types of states can be either true or false, depending on whether the event they represent occurred or the situation exists. They are therefore called **predicate states**. This approach is very similar to that taken in the CASNET glaucoma diagnosis system [39], in terms of what the states represent.

Many events and many complex relationships may be summarized by a single state. A state network can be thought of as a streamlined model of disease that unifies several important concepts and guides us in our goal of diagnosis.

The difference is that CASNET models pathological conditions, i.e., abnormal behavior, whereas we model the expected behavior. This allows us simulate the correct behavior and also to simulate abnormalities by representing them as the expected states that were never reached by the DVMT system.

In some cases we need to represent not only whether a situation exists or not, but rather the **relationship among several situations in the system**. These cases can be represented using a slightly different type of state called **relationship state**. There are thus two types of states in the system model. The rest of this section describes these states in detail, explaining how they are used to represent different aspects of the system behavior.

§2.1 Predicate States

The DVMT system behavior can be described by a sequence of predicates. When the system is behaving correctly these predicates become true as it progresses through the desired situations in deriving the results (a map of the environment) from the input data. The predicate states represent these desired situations. The names of the states correspond to the events or situations they represent. Examples of predicate states are: **HYP-CREATED**, **GOAL-CREATED**, **GOAL-SATISFIED**, **KSI-RATED**, **KSI-SCHEDULED**, **KSI-EXECUTED**, **MESSAGE-SENT**, and **MESSAGE-RECEIVED**.

In addition to representing the result of some event, a predicate state can represent some arbitrary situation in the system, which can be the result of many independently acting events. It may therefore involve more than one object, an object and some parameters, or a relationship among several objects and parameters.¹ The names of these predicate states describe the situation they

¹Even though a predicate state may represent a relationship among several parameters, this is different from a relationship state. Predicate states represents situations that are either true or false. A relationship state, on the other hand, can represent an arbitrary relationship among

represent. Examples of these states are: MESSAGE-EXISTS, KSI-RATING-OK, KSI-RATED-MAX, SENSOR-OK, and CHANNEL-OK. For example, the state MESSAGE-EXISTS represents a situation involving an object (message) and some system parameter values (the parameters controlling communication). In order for this state to be true, the object to be sent (i.e., the message) must exist in a node and that node must be aware that it should be sending this object to another node. Another situation-representing predicate state is the state KSI-RATED-MAX. This state represents a situation where a knowledge-source instantiation (ksi) exists and is the highest rated one of all the knowledge-sources instantiations on the queue. Figure 18 illustrates how the modeling formalism represents communication among two processors via message passing. Part A shows only the state transition diagram. Part B shows also the abstracted objects which are described in a later section.

In all the above cases the states represent either an event that has occurred or not, or a situation in the system that exists or not. These states can thus take on two values: true if the event it represents did occur or the situation it represents does exist, false otherwise. Evaluation of the model states is central to the reasoning mechanisms described in Chapter IV. The predicate states' values are determined by examining the DVMT system data structures (blackboards and knowledge source instantiation queues) in order to determine whether a situation exists or not.

situations.

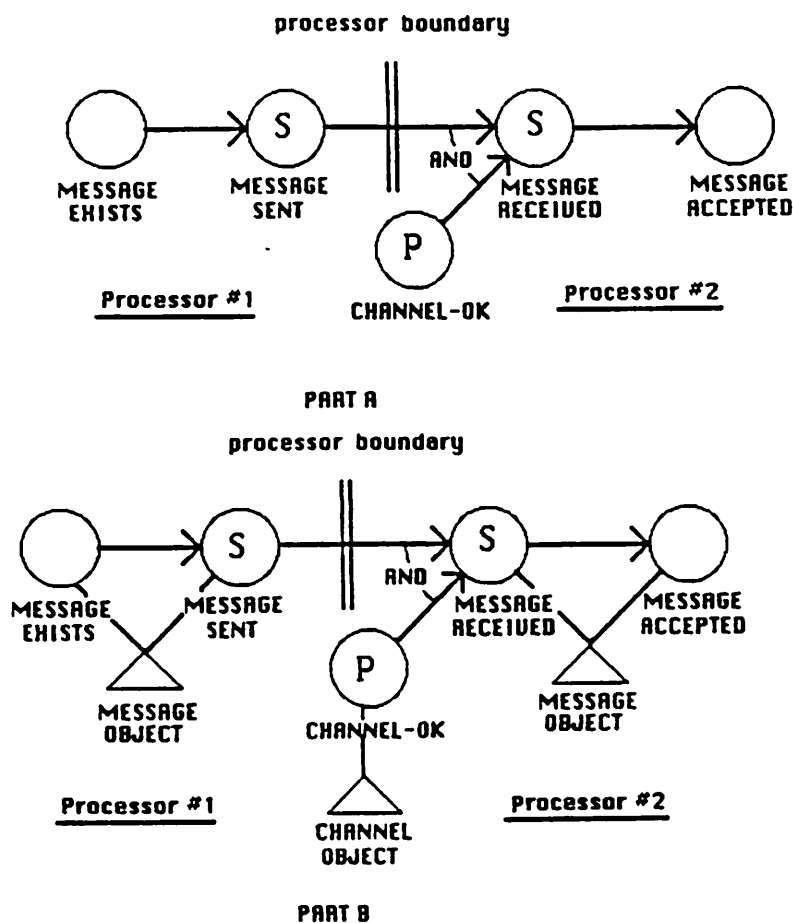


Figure 18: Illustrating the Modeling Formalism

The model represents the correct sequence of states during communication among two processors via message passing. First, the message must exist in Processor#1. Next, that message must be sent by that processor. In order for the message to be received by Processor#2, an additional state must be satisfied, the channel must be working (represented by the CHANNEL-OK state). Finally, the message must be accepted by Processor#2. The states MESSAGE-RECEIVED and MESSAGE-SENT are intermediate symptom states; whenever they are reached they are saved as pending symptoms for further diagnosis. State CHANNEL-OK is a primitive state; whenever it is found false, it is reported as an identified failure. Part A of the figure shows only the state transition diagram. Part B shows also the abstracted objects MESSAGE-OBJECT and CHANNEL-OBJECT.

§2.2 Relationship States

Relationship states can in principle represent an arbitrary relationship among events or situations in the DVMT system. Currently, they are used to represent the relationship among the ratings of DVMT objects (e.g., hypotheses, goals, or knowledge source instantiations) and can be either **less-than**, **equal**, or **greater-than**. (Chapter X discusses how these states could be used to reason about other types of relationships in the DVMT system.) As with the predicate states, a relationship state has a value: the type of relationship among the objects. The state names correspond to the objects whose ratings they represent. For example: HYP-RATING, KSI-RATING, and GOAL-RATING.

Relationship states are necessary for **comparative reasoning**. Recall that Comparative reasoning involves comparing the behavior of two objects in the DVMT system. This type of reasoning is used when the DM tries to determine why one object did not achieve some desired state by comparing it to an object that did achieve that state. Specifically, it is used to understand why one object was rated higher or lower than another object. This is useful in trying to understand why a knowledge source instantiation did not run. Comparative Reasoning utilizes qualitative reasoning [12] to determine why the rating of one object in the DVMT was higher or lower than the rating of another object. The details are discussed in Chapter IV. Figure 19 illustrates a state transition diagram consisting of relationship states, which represents how the rating of a knowledge source instantiation is derived. The details of the differences among the predicate states

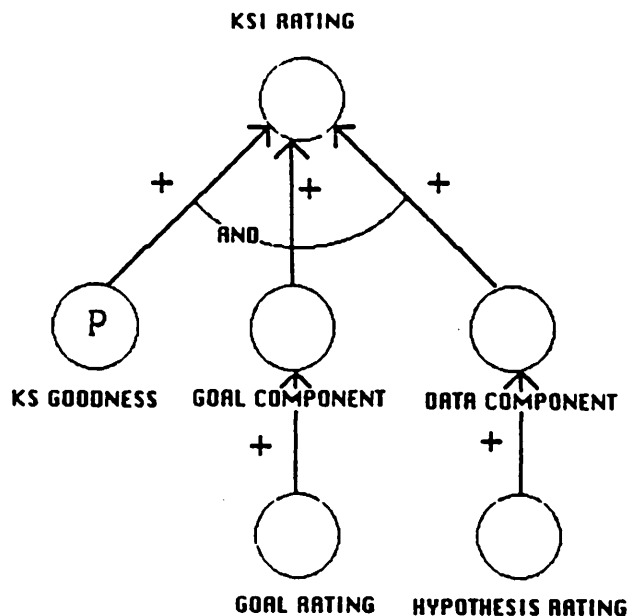


Figure 19: State Transition Diagram Illustrating Relationship States
 This figure shows the qualitative relationships among the value of the the knowledge source instantiation (*ksi*) rating and the three components it is derived from: *KS-GOODNESS*, *GOAL-COMPONENT*, and *DATA-COMPONENT*. The + on the state transition arcs indicates that each of the components is directly proportional to the final *KSI-RATING*. The qualitative relationships among the *GOAL-COMPONENT* and the *GOAL-RATING* and the *DATA-COMPONENT* and the *HYPOTHESIS-RATING* are also shown.

and the relationship states are discussed in Chapter VIII which describes the implementation details.

§2.3 Primitive States

A subset of the states in the System Behavior Model is designated by the model builder as primitive. These states represent the “primitive causes” that constitute the reportable failures. Typically, primitive states are the leaf states in the model; that is, states that have no predecessor states (back neighbors). However, this is not necessary; any state can be marked as primitive, depending

on how far the DM is to go in analyzing the causes of the observed symptoms. In the diagrams, primitive states are marked with a "P" inside the circle. Two other specially marked states are the symptom states, marked with an "S", and states indicating a transition to a different type of reasoning, marked with a "C". Symptom states are states that are considered primitive by one node but are considered as symptoms by another node, which will continue their diagnosis. Examples of such states are the MESSAGE-SENT states in the Communication Cluster. The reasoning transition states are points in the model where a transition to a different type of reasoning is to take place. Currently, this only occurs when a rating of a knowledge source instantiation is to be investigated by Comparative Reasoning. The states where this occurs are those that refer to the rating of these objects.

§2.4 Transition Arcs Linking the States

The states, both predicate and relationship, are linked together by directed state transition arcs (STA). The resulting state transition diagram (STD) (refer back to Figure 16) represents the desired sequence of events or situations in the DVMT system (in the case of predicate states) or the dependencies among the various object ratings (in the case of relationship states). To the extent that the existence of one situation enables or influences the existence of another situation the states in the STD can be considered causally related. The state transition arcs are directed and therefore implicitly capture the flow of time.

Because the state transition arcs are directed, we can speak of predecessor and

successor states of a particular state. Predecessor states can be thought of as the causes of a situation and successor states can be thought of as the consequences of the situation. In the case of predicate states, the predecessor states represent situations or events which had to have occurred before the state can occur and the successor states represent situations or events which should occur after the one represented by the state. In the case of relationship states, the predecessor states represent components that influence the rating of the component represented by the state and the successor states represent components whose rating is influenced by the current state.

If an situation or an object rating is influenced independently by a number of preceding situations or object ratings, then the states representing these situations or ratings are ORed. This means that any one of the preceding events determines the outcome of the event independently and in the same manner. Any one is sufficient to enable the event or to determine its outcome. Figure 20, part A illustrates such a relationship among the states preceding the state PT, which represents a pattern track hypothesis. The graph represents the different ways of deriving a pattern track: namely, from shorter track segments (PT), from the necessary locations (PL), from a track of the same length at a lower level (VT), or from a pattern track received from another node (MESSAGE-ACCEPTED). Since the PT state can be achieved via any of these predecessor states, these states are ORed. If the outcome of an event is influenced by a number of preceding events acting together, then the states representing these events are ANDed. This means that all of the preceding events must occur before the event can take place.

Figure 20, part B illustrates the AND relationship among the states preceding the state KSI-RATED, which represents the situation where a knowledge source instantiation (KSI) has been created and assigned a rating. These AND and OR relationships link the state transition arcs and the state transition diagram is thus an AND/OR graph. Note that the relationships can have an arbitrarily complex substructure, representing AND and OR relationships among *groups of states* in addition to the individual states.

The state transition arcs linking together relationship states contain additional information necessary for qualitative reasoning. In order to be able to reason about how the values of various object ratings influence one another, the model must represent these influences explicitly. We have found that we needed to represent two types of influences: "causes the increase", represented by a +, and "causes the decrease", represented by a -. These signs, associated with the state transition arcs linking two value states, indicate how the increasing value of the preceding state will influence the value of a state. A positive influence indicates that as the value of the attribute represented by the predecessor state grows, so does the value of the attribute represented by the current state. A negative influence means the opposite. The model is thus able to represent whether the value of a component of a rating is directly or inversely proportional to the rating itself. Cross has constructed a similar model for a plan justification system in the domain of air traffic control [7]. Figure 19 shows a relationship state model representing the derivation of a knowledge source instantiation rating with the arcs labeled by the type of influence they exert on the KSI-RATING state.

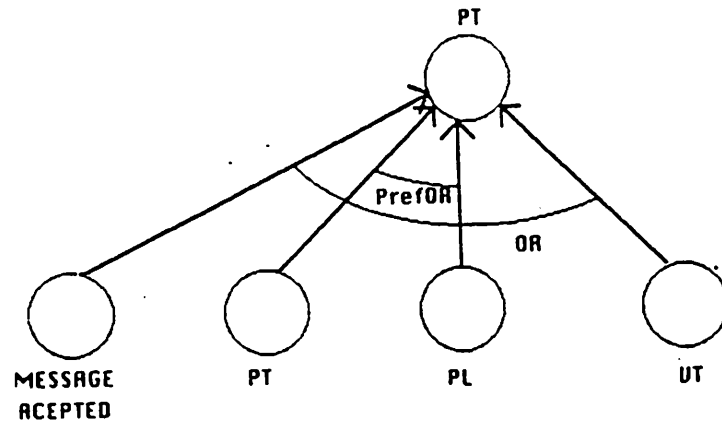
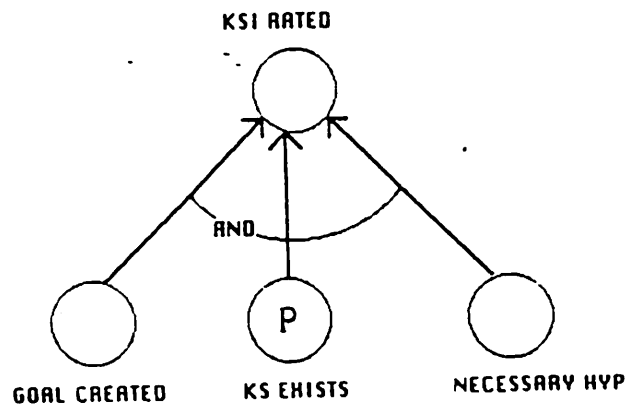
PART APART B

Figure 20: Relationships Among the State Neighbors

Part A shows how the logical relationships among the predecessor states of PT. A pattern track can be derived in four ways: via a received message (MESSAGE-ACCEPTED state), via shorter pattern track segments (PT state), via pattern locations (PL state), or via a vehicle track of the same length (VT state). These four states are therefore related by an OR, indicating that either pathway is sufficient to achieve the PT state. The PrefOR operator indicates the DM's preference to analyze the PT path before it analyzes the PL path. Part B of the figure illustrates a case where all three predecessor states must be achieved before the state KSI-RATED can be achieved.

The arcs connecting the relationship states can, in principle, also be related by the AND and OR logical operators. This would mean that there are choices of components influencing the calculation of an object rating. We have not found any cases requiring this type of representation in the DVMT system.

§2.5 Data-Dependent Choice of Neighboring States

In using the model for diagnosis, there are times when it is better to explore one pathway before another. In order for the model to capture this situation, we have introduced a third type of logical relationship among successor or predecessor states, the preferred OR (PrefOR) relationship. This relationship works similarly to the AND and OR, in that it links together the state transition arcs. The states it links are ordered and only one of them will be explored at any one time, depending on the specific situation in the DVMT system. For example, a track can be constructed from shorter track segments or from the locations. Assuming the existence of the necessary data and the appropriate settings of the system parameters, the DVMT can construct a track in either way. However, when reasoning about the system behavior, we have found that dealing with both situations led to a lot of unnecessary analysis. This could be avoided if the model designer could encode a preference among pathways. In the case of the track, first analyze why it was not constructed from shorter track segments and only if no such segments are found, begin investigating whether the component locations exist. Figure 20, part A shows a model representing this situation. The PrefOR relationship among states thus allows the DM to reason about different

pathways in the DVMT system according to an order given by the model builder and dependent on what data exists in the system at the time.

§2.6 Organizing the States into a Hierarchy

The state transition diagram representing the system behavior divided into small clusters for manageability. These clusters are then organized into a hierarchy corresponding to an increasingly detailed view of the system. A high level cluster represents selected situations as contiguous states. A more detailed cluster, at a lower level of the model hierarchy, represents other events or situations which occur in between these states. There are two reasons for this hierarchical organization.

1. It allows efficient representation of the system behavior in the model by representing repeated sequences of actions as a separate cluster. This cluster is parametrized with respect to the varying data. This cluster is placed at a lower level of the hierarchy and linking this cluster into a high-level model where ever necessary.
2. It makes diagnosis more efficient by allowing reasoning at different levels of abstraction. This allows the DM to quickly focus on the problem by delaying detailed investigation until necessary.

Efficient Representation. Many actions occur repeatedly in the system. For example, the sequence of steps necessary for knowledge source instantiation, scheduling, and execution. This process is identical for all knowledge sources as far as the events which must occur. What differs from one instance to the next is the data, the type of goals, and the type of knowledge source used. Each

hypothesis stimulates the creation of a goal and, if possible, the scheduling of a knowledge source instantiation. The sequence of steps described above thus occurs in between each pair of hypotheses at adjacent levels of abstraction. It represents the process of driving a lower level hypothesis to a higher level of abstraction, e.g., sl to gl, or combining locations or shorter track segments into longer tracks (sl and st to st).

Rather than representing this process repeatedly in the model every time it occurs during the data transformation, it is more efficient to represent it in a separate cluster. This cluster parametrically represents the specific data and knowledge source types and thus represents an arbitrary data transformation. This cluster is then placed at a lower level of the hierarchy. In the higher level Answer Derivation Cluster, all states point to the KSI-RATED state in the Ksi Scheduling Cluster as their lower successor state and all the KSI-EXECUTED states in the Ksi Scheduling Cluster point to the states in the Answer Derivation Cluster as their higher successor state. The KSI Scheduling Cluster and its relationship to the Answer Derivation Cluster is shown in Figure 21. Because of this hierarchical organization of the model, we now need additional types of predecessor and successor states: higher and lower successor and predecessor states. Figure 22 illustrates the different types of neighbors a state may have.

Efficient Diagnosis. The hierarchical organization is more efficient not only with respect to representing the system behavior but also when it comes to reasoning about it and diagnosing its errors. By allowing the DM to view the system at varying degrees of detail, a hierarchical model enables it to reason

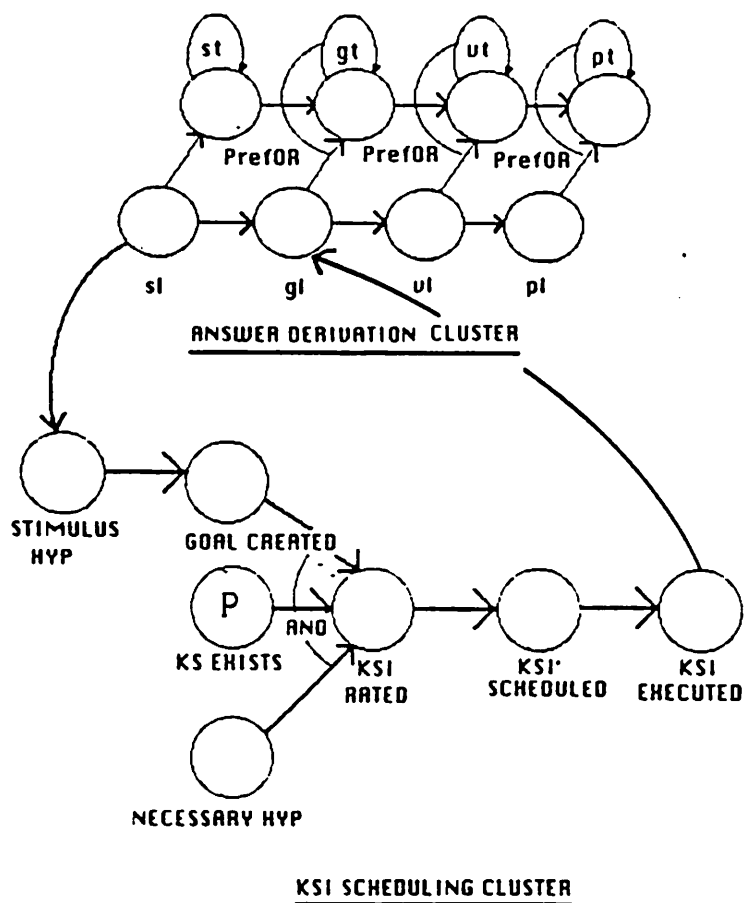


Figure 21: Linking Clusters at Different Levels of the Hierarchy
 The lower Ksi Scheduling Cluster represents events occurring in between each pair of states in the Answer Derivation Cluster. The state **STIMULUS-HYP** is thus the lower front neighbor of all states in the Answer Derivation Cluster, representing that each hypothesis is expected to stimulate the production of a goal the scheduling and execution of a knowledge source. The state **KSI-EXECUTED** is the lower back neighbor of each of the states in the Answer Derivation Cluster, indicating that each hypothesis is produced as a result of knowledge source execution. Only two of these links are shown: the lower front neighbor of *SL* is **STIMULUS-HYP** (conversely, the upper back neighbor of **STIMULUS-HYP** is *SL*), and the lower back neighbor of *GL* is **KSI-EXECUTED** (conversely, the upper front neighbor of **KSI-EXECUTED** is *GL*).

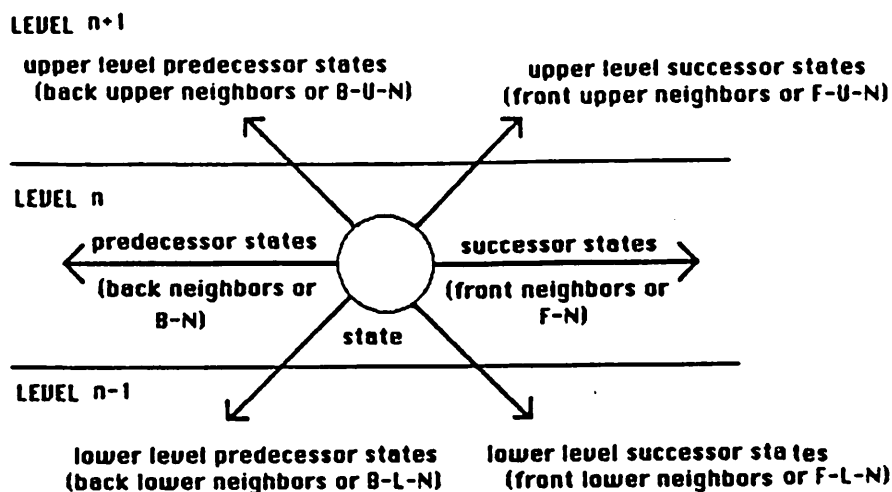


Figure 22: The Different Types of Neighbors a State May Have
A state may have up to 6 types of neighbors; front and back neighbors at the same level of the model hierarchy, lower front and lower back neighbors linking in a lower level model, and upper front and upper back neighbors linking in an upper level model.

about the system behavior at different levels of abstraction. The DM can thus omit details if they are not needed, but have them available, at a lower level of the model hierarchy, when necessary.

The difference between a flat and an hierarchical model is analogous to the difference between linear and tree search. The idea is that we choose certain states to look at first. These are the states we place at a higher level of the model hierarchy. (Chapter VI discusses some of the issues involved in choosing the appropriate hierarchy.) Depending on the information we find at these states, (i.e., the state value), we may either continue to do the diagnosis at the same level or drop down to a level below. This is more efficient than a flat model as long as the conditions which are most likely to cause problems are at the higher levels of the hierarchy and are therefore checked first. The following example illustrates

this point on a common sense problem.

Suppose you are typing at your terminal, which is linked to the computer via a phone line, when suddenly the screen goes blank and there is no response. How you will go about determining what the problem is depends on what you know about your terminal, the system you are working on, and on other environmental factors. If you know that your terminal has been acting strange recently, the first thing you are likely to check is the terminal. If you know that the system has been unreliable, you are likely to check whether the system is still up. If there is a thunderstorm, you are likely to assume that you lost the line and will redial, etc. Clearly, you are not going to start by taking apart the terminal and checking some resistor unless you know that this has happened many times in the past and that this resistor was the problem.

An analogous situation in the DVMT occurs when the DM tries to determine why some hypothesis was not constructed. Rather than looking for the knowledge source instantiation that could have produced that kind of hypothesis, the DM first looks for the necessary supporting data and only if this data exists does it investigate the knowledge source instantiations to see whether they were scheduled and if so, why they did not run. This means that the diagnosis is first done using the Answer Derivation Cluster and only later using the Ksi Scheduling Cluster (see Figure 21).

Our knowledge of the system behavior thus determines the hierarchy of the model we will use to reason about the system. The structure of the hierarchy determines the order in which the different states of the model are examined

during diagnosis and thus imposes a search strategy on the diagnosis. We could choose a simple, blind search strategy and simply search a flat model sequentially every time, doing either a backward or forward exhaustive search. This could take a long time and we will therefore attempt to speed up the search by using the principle of binary search and splitting the area to be searched by determining how far the system got in its expected behavior. We do this by looking at a state and determining whether the system has reached that state. If the state has not been reached, we continue looking at the states' predecessor states, going backwards in time. If the system did reach that state, then we know that the problem must lie in between that state and the following one, and we examine the states that occur in between these two states, at a lower level of the hierarchy.

§3. Abstracted Objects

The states are linked to the abstracted objects whose behavior they represent. Abstracted object in the SBM represent objects in the DVMT system; e.g. hypotheses, goals, knowledge source instantiations, messages, sensors, and channels. An abstracted object represents an object that may or may not actually exist in the DVMT system. In other words, it can represent a hypothesis or a goal which has not been created by the DVMT system. This ability to represent objects that have not been created by the DVMT system is the key to reasoning about the expected behavior of the system. Diagnosis most often begins with the description of some desired object and continues by tracing back through the System

Behavior Model to see why this object was not created by the DVMT. These objects are also useful when we want to predict how some DVMT situation will develop in the future. Figure 23 illustrates the relationship among the processing structure of a node in the DVMT system, the abstracted objects, and the state transition diagram.

Why do we need abstracted objects as separate entities in the system model? If the system model was represented only by the state transition diagrams, much information would need to be represented redundantly. Because each state refers to some aspect of some object it is necessary to represent that object's attributes within the state's attributes. Since several states may refer to the same object, it is inefficient to repeat that object's definition in each of the states. The model therefore separates the information about the object from the information about that object's state. Figure 24 illustrates two situations where the same objects are referred to by different states. Not only can separate states in the same cluster refer to the same object (see Figure 24, part A) but states in different clusters can refer to the same set of objects (see Figure 24, part B). In the latter case each cluster models a different aspect of the object's role in the system behavior. This makes the model into a much more concise description.

Representing Classes of Objects by Underconstrained Objects. An abstracted object may represent an individual object or it may represent a whole class of objects. An abstracted object representing a class of objects is called an **underconstrained object** because some of its attribute values are under-specified and may therefore define a collection of objects. This is in contrast to

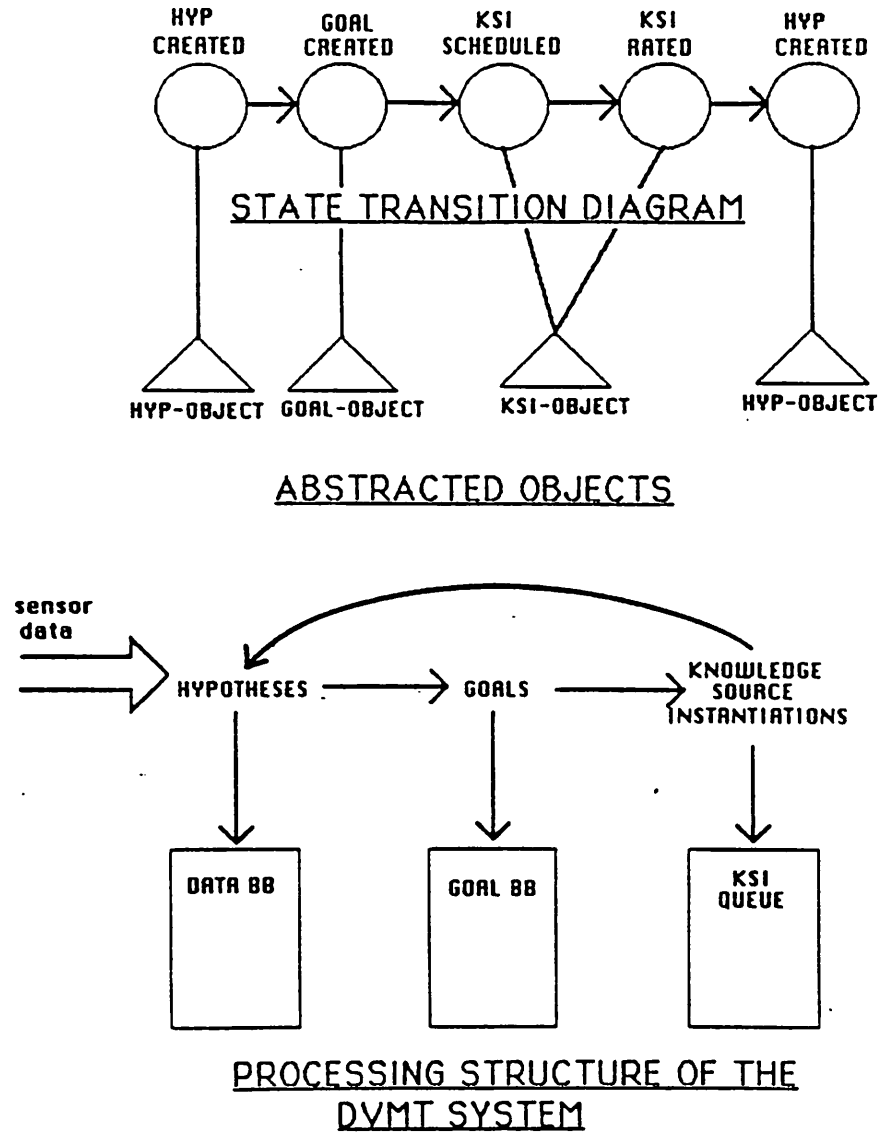


Figure 23: Relationship Among the DVMT system, the Abstracted Objects, and the State Transition Diagrams

This figure illustrates how the expected behavior of a DVMT node is represented by System Behavior Model. The three distinct types of objects in the DVMT, the hypotheses, the goals, and the knowledge source instantiations, are each represented by an abstracted object in the model. The different states these objects are expected to undergo are represented in the state transition diagram.

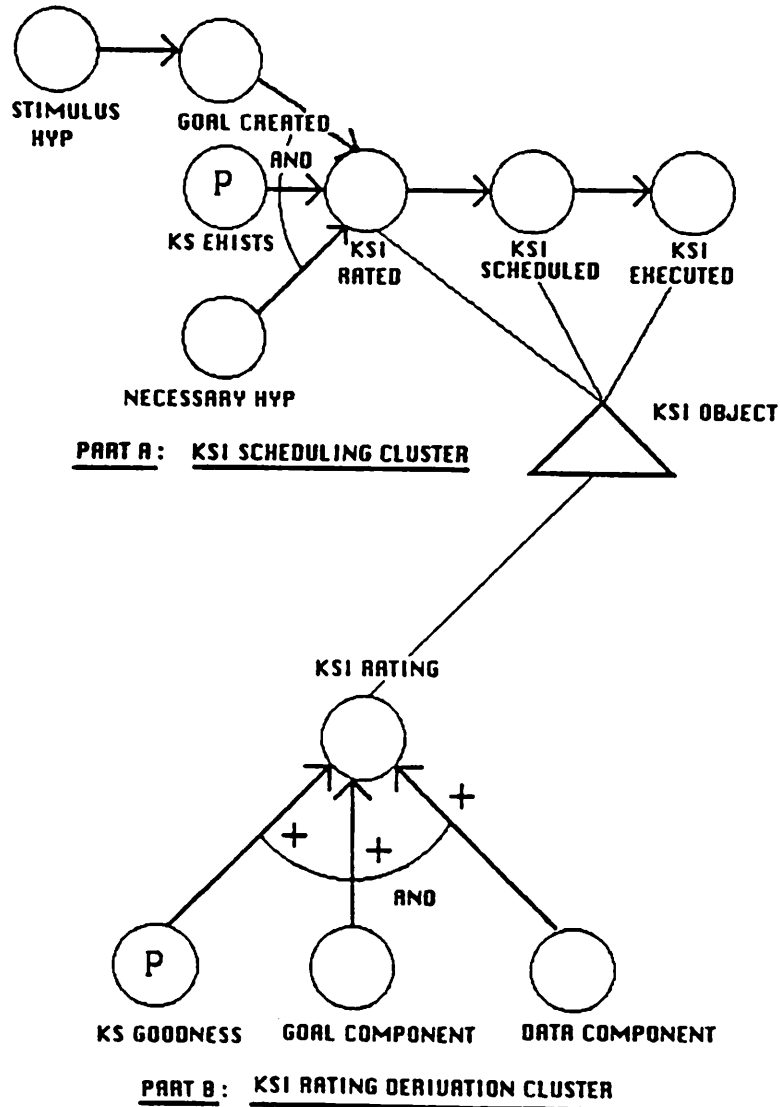


Figure 24: Different States Can Refer to the Same Object
 Part A shows a situation where several states in one model cluster can refer to the same object. Part B shows how two different model clusters (Ksi Scheduling Cluster in Part A and Ksi Rating Derivation Cluster in Part B) can refer to the same object.

a **fully-constrained object** where the attribute values together can represent just one object in the DVMT. For example, a fully constrained abstracted object KSI-object looks like this:

Fully-constrained KSI-OBJECT

ks-type	mf:vt:vt
stimulus-hyp	vt 1-5
output-hyps	vt 1-8

In an underconstrained object one or more attributes are underconstrained. They may specify a class of acceptable values or may be empty, meaning any value is acceptable. Such objects define a class of objects rather than just a single instance. For example, an underconstrained hypothesis object might have a list of levels in its level attribute and thus represent the entire class of hypotheses at any of those levels. Below are examples of two underconstrained objects; each representing a different class of the knowledge source instantiation object.

Underconstrained KSI-OBJECT #1

ks-type	don't care
stimulus-hyp	don't care
output-hyps	vt 1-8

Underconstrained KSI-OBJECT #2

ks-type	mf:vt:vt or mb:vt:vt
stimulus-hyp	don't care
output-hyps	don't care

Why do we need underconstrained abstracted objects? They are useful in two cases. First, we can save time by reasoning about a class of objects rather than about each individual instance (of course, we must know that each of the objects in the entire class will behave identically with respect to whatever it is we are trying to reason about.) An example of this type of use is the simulation of the effects of an identified fault. Chapter IV discusses this in detail. Second, in cases where we do not know all the values of an object's attribute or we do not know the exact value. This is used when we want to ask an underconstrained question about the system behavior. Examples of underconstrained questions are:

- Were any hypotheses at levels sl, gl, or vt created?
- Did any goals at levels vt and pt stimulate the instantiation of a synthesis knowledge?
- Were any hypotheses, received from nodes 1 and 2, accepted and used to stimulate local knowledge source instantiations?

Any kind of reasoning that can be done with a fully constrained object, can also be performed using an underconstrained object. Some interesting issues arise when we try to integrate reasoning with both fully constrained and underconstrained objects. For example, we must define criteria for object overlap so that the DM detects cases where a class of objects has already been analyzed and there is therefore no need to repeat the analysis with the individual members of that class. Other issues include the initiation of reasoning with underconstrained objects and control of the transitions among reasoning with the different types of objects. These issues are discussed in Chapters VIII and IV.

In conclusion, the abstracted objects are used to represent the DVMT system objects in the system behavior model. There is a natural correspondence between the two; i.e., every object type in the DVMT has its corresponding abstracted object type in system behavior model. This one-to-one mapping between the object types is not necessary however. It is interesting to speculate how the capabilities of the DM would be increased if we allowed the representation of and reasoning about larger, composite objects such as groups of goals, hypotheses, and knowledge source instantiations, as well as smaller objects representing the current object attributes such as time location lists or event classes. Some of these implications are discussed in Chapter X. Another interesting application of the underconstrained abstracted objects would be to combine the salient features of several symptoms into one and reason about the resulting abstracted object instead of reasoning about the individual symptom objects. This is also discussed in Chapter X.

§4. Constraint Expressions among the Abstracted Objects

Relationships among the abstracted objects are expressed in the constraint expressions linking the attributes of neighboring abstracted objects. The set of abstracted objects can be viewed as a network of such constraints. This network of local constraints allows the evaluation of the entire model, starting with one or more objects in the model whose attribute values have already been determined.

The constraint expressions capture relationships among the object in the

DVMT system. Evaluating these expressions, starting with one or more evaluated abstracted objects, is tantamount to simulating the correct DVMT system behavior; assuming of course that the model correctly represents the DVMT system. The DVMT system functionality is thus captured in these constraint expressions and can be simulated, in either direction, by evaluating them with respect to some known data. The evaluation can begin at any point in the model. If it begins with the sensors and the data, then the model allows the simulation of how the system will process the data. If it begins with a desired final answer, then the model allows the analysis of how that answer could have been derived and what data is necessary for its derivation. The evaluation can also begin at any intermediate state and proceed in either direction through the model.

The details of the constraint expressions are discussed in Chapter VIII. Briefly, they functional expressions where the functions simulate the DVMT behavior and the arguments are the attributes of the already evaluated abstracted objects. To determine the characteristics of a goal created as a result of a vehicle track hypothesis' creation, we would evaluate the constraint expression contained in each of the variable attributes of the GOAL-OBJECT. For example, the attribute *goal-type* is a function of the level of the stimulus hypothesis. The constraint expression would therefore contain the name of the function, *get-goal-type*, as well as a description where the arguments to the function are. In this case, the argument needed is the *level* attribute of the object VT-HYP-OB, which represents the stimulus hypothesis.

§5. Object-State Links

The final component of the SBM is the set of links connecting the state transition diagram with the abstracted object constraint net. (Refer to Figure 16.) The object-state links are bi-directional links connecting each state to the abstracted object whose behavior that state represents.

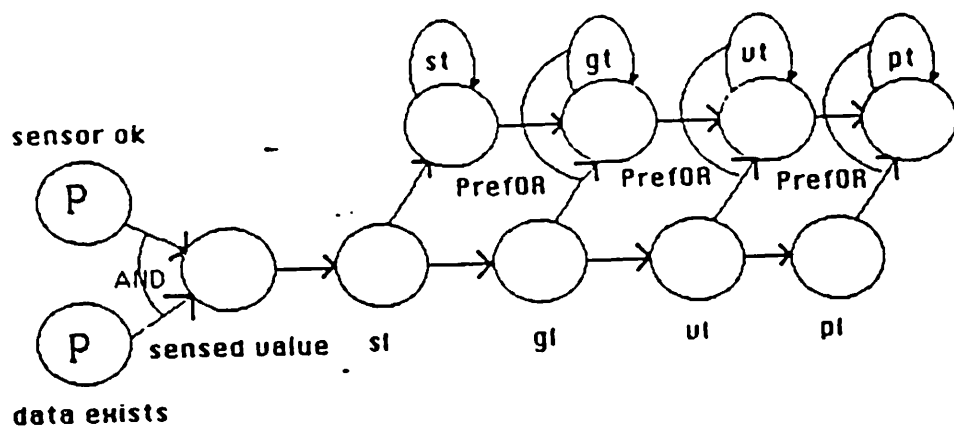
§6. Model Instantiation

Instantiation is the process of taking a general description of an object, process, or a situation, and assigning values to the attributes of the generalized description in order to represent its specific instance. For example, a generalized description (i.e., a prototype) of a chair might contain attributes such as size, color, type of seat, type of back, etc. Instantiating this generalized description consists of filling in the specific attribute values; i.e., size is large, color is white, seat is cloth, and back is straight.

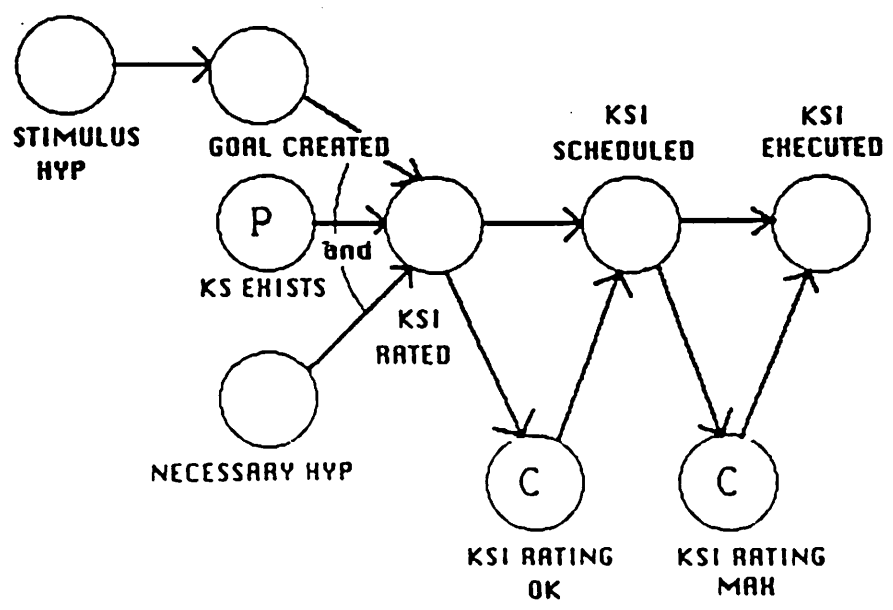
The SBM represents a generalized description of selected aspects of the DVMT system behavior. Figure 25 shows the set of model clusters we have constructed.

Before the model can be used for diagnosis, it must be instantiated to represent a particular situation in the DVMT system, that is a particular instance of the DVMT system behavior. (An instantiated SBM is analogous to the Patient Specific Model in ABEL [30] and to the set of confirmed or denied nodes in CASNET [39].)

The instantiation consists of evaluating the state and abstracted object at-



ANSWER DERIVATION CLUSTER



KSI SCHEDULING CLUSTER

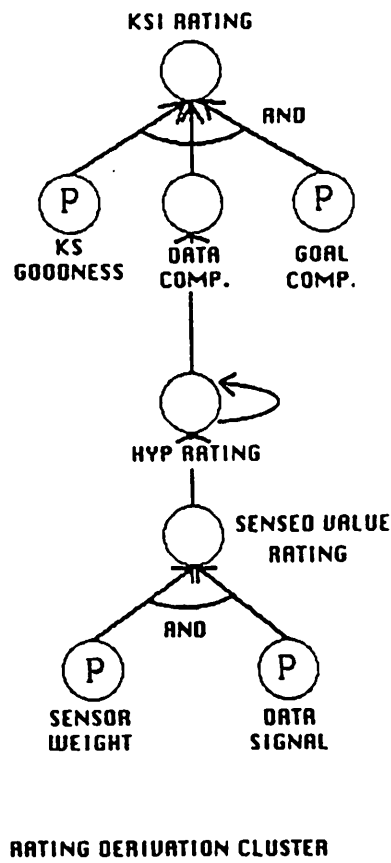
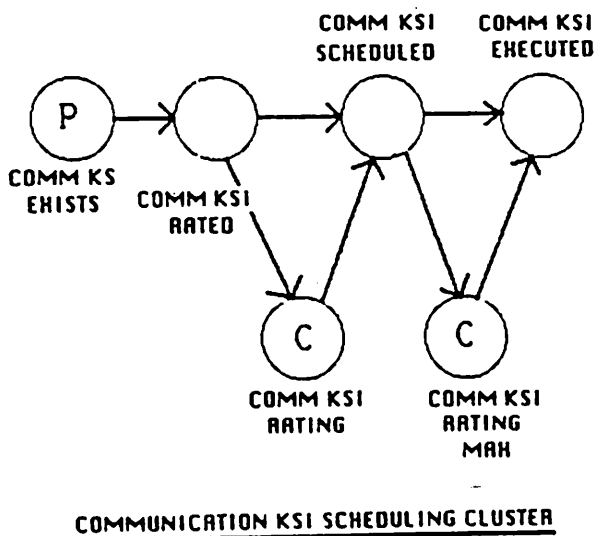
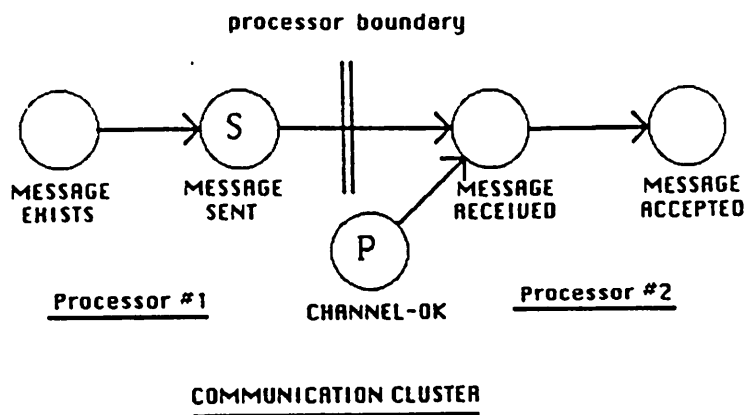


Figure 25: The Model Clusters Representing the DVMT Behavior
 This figure shows the diagrams of all five model clusters used in diagnosis. At the highest level are the Answer Derivation Cluster and the Communication Cluster. The Ksi Scheduling and Comm Ksi Scheduling Clusters represent the events that take place in between the states at the higher level models. The Ksi Rating Derivation Cluster represents the additional knowledge about value relationships among the rating components of the various objects.

tributes with respect to a specific situation in the DVMT system. The instantiated model then represents this situation. The uninstantiated SBM represents all the possible pathways of deriving a spanning hypothesis in the DVMT system, at all the levels of detail. An instantiation of the SBM typically represents a only small subset of those pathways, consistent with the current system parameters and the input data. Typically, only a subset of the events is explored and since only the explored parts of the model need to be instantiated, this further reduces the number of states and abstracted objects that need to be instantiated.

The model instantiation begins by assigning values to the attributes of one or more abstracted objects. These values can come either from the detection component, from previous diagnosis, or can be set by the user. This initially-evaluated state and object is called the **symptom**. These known values are then propagated through the model via the constraint expressions linking the neighboring objects. This in effect duplicates the correct behavior of the DVMT system. The instantiation occurs one state-object pair at a time and can be done in either direction through the model; forward or backward. If the propagation of values is done backward, then we are running the DVMT system backward, determining the situations that could have led to the desired situation. If the propagation is done forward, then we are simulating how the system will behave as a result of the initial symptom. The SBM can therefore either determine how some desired situation might come about or it can simulate how the DVMT system would process some input data. The former is used to perform diagnosis when the DVMT system does not achieve some desired state, the latter is used

to simulate the effects of an identified fault on future system behavior.

Exactly which state-object pair is instantiated depends on both the type of reasoning currently active and what the state value is; i.e., whether the situation represented by the state occurred in the DVMT system. Only the parts of the model relevant to the situation being analyzed are instantiated. Instantiation ends when the model ends and cannot be expanded further. This can occur under the following conditions: a primitive state has been reached, a non-expandable state has been reached (i.e., a state which could not occur in the DVMT system because the current parameter settings would not allow it), the model ends (i.e., the state has no neighbors to expand), or a state has been reached which is labeled as a symptom state, for example a node transition state, which means that it will be saved for later diagnosis.

The algorithm for the model instantiation is as follows:

1. Assign values to the attributes of the initial state-object pair, the symptom.
2. Choose a neighbor of this state to instantiate next. (Since most symptoms are false predicate states, diagnosis usually begins by expanding the back neighbors of the symptom, in order to determine what predecessor states could have led up to it.) After that, the neighbor to expand is a function of the reasoning type active and the state's value.
3. Instantiate (interchangably used with expand) the selected object and state by evaluating their attributes.
4. Determine whether that state has been reached by the DVMT system or what the relationship of that state is with the what actually occurred in the DVMT system. In other words, determine the value of the state. Ei-

ther by looking at the DVMT system data structures or by looking at the surrounding states.

5. If state is a non-primitive state, go to step 2. If state is a primitive state, end instantiation for this part of the model.

Instantiation thus consists of evaluating a portion of the SBM in order to explain the effects of some situation on the system behavior or to analyze how some situation came about.

Both the states and the abstracted objects in the DVMT system model contain information necessary to instantiate the model for a specific situation in the DVMT system. This information includes information about how to group together similar objects so that they can be analyzed together, when to switch between different types of reasoning, how to access the DVMT system data structures to determine whether the DVMT system behaved as predicted by the model, and how to choose a subset of the the DVMT system objects for analysis. Chapter VIII discusses model instantiation in more detail.

§7. Summary

This chapter described the structure of the System Behavior Model and showed how the model is instantiated to represent a particular situation in the DVMT system. The model consists of a state transition diagram representing the possible behaviors of the DVMT system. There are two types of states in the model: predicate states and relationship states. Predicate states represent

whether some situation in the system exists or not and can therefore be true or false. Relationship states represent the relationship among groups of events in the system and can, in principle, have any value. Currently, these states are used to represent the relationship among the ratings of two different objects and can be less-than, equal, or greater-than. The states are linked so as to capture causal relationship among the events in the DVMT. The resulting graph is an AND/OR graph. In order to capture preference among several states the graph also includes a PrefOR relationship where several states are specified in a preferred order but only one will ever be instantiated.

Each state refers to an object in the DVMT system. These objects are represented by their corresponding abstracted objects in the system model. The relationship among the DVMT objects is captured in the constraint expressions relating the attributes of neighboring abstracted objects. It is these constraint expressions that effectively simulate the correct DVMT system behavior by allowing the evaluation of the model during diagnosis. This process is called the model instantiation and is the basis for all the different types of reasoning, which will be discussed in Chapter IV. The model is organized into a hierarchy in order to make both the representation of the DVMT more concise and to make reasoning about the DVMT more efficient by delaying details until necessary.

Chapter IV

MODEL USE

Systems for complex tasks
are generally more complicated
than systems for simple tasks.
- Stefik et al., in *Building Expert Systems*

This chapter discusses how the System Behavior Model (SBM) is used by the Diagnosis Module (DM) to analyze DVMT behavior. The SBM represents its knowledge about the system behavior in a form suitable for many different types of reasoning. These are used to diagnose why some desired situation in the DVMT did not occur, to predict what effects a situation will have on future system behavior, or to compare two situations in the DVMT and explain why they are different. The different reasoning types will be described and illustrated by examples. Use of underconstrained objects in representing and reasoning about a class of objects in the DVMT will be discussed. Finally, the DM's method of checking global consistency of the diagnosis by using path values is described.

§1. Introduction

Reasoning about DVMT behavior consists of instantiating the part of the SBM that represents the system behavior relevant to the situation being analyzed. Reasoning in this context thus means the different strategies for model instantiation and the different uses of the information contained in the instantiated model. If we view the uninstantiated model as a representation of all the expected system behaviors, then instantiating a part of the model represents a search through this space. The instantiated model thus represents a small subset of the expected DVMT behaviors. Each reasoning type imposes a different search strategy on the model instantiation. The search stops when all possible pathways relevant to the situation being analyzed have been explored. Within the instantiated model, the analysis is done exhaustively by a depth-first search. Exhaustive search is feasible here because the search space has already been sufficiently pruned by the strategies associated with each reasoning type, by the data in the DVMT system, and by other search-reducing methods¹ discussed in Chapter VI.

The aim of all the different reasoning types is to explore the causes, or the effects, of an initial situation, provided as input to the DM.² This situation is represented by an instantiated state and its abstracted object; that is, a state and object whose attributes have been evaluated. The DM propagates these known

¹For example, grouping together objects that behave similarly and using underconstrained objects to reason about a class of objects rather than the individual cases.

²Currently this is done by hand. In a fully fault tolerant system the input would come from a detection component.

values through the SBM, using the constraint expressions among the abstracted objects, and thereby instantiates a sequence of states causally related to the initial situation. We call such a sequence a causal pathway.

When the initial situation represents some desirable event in the DVMT that never occurred, we call it a **symptom**. A symptom represents an object that was never created by the DVMT. Usually this object is a hypothesis. Upon receiving a symptom, the DM traces back through the SBM in order to find out at which point the DVMT stopped working "correctly"; i.e., in such a way as to reach the desired situation. This type of diagnostic reasoning thus consists of backward chaining through the SBM. Since it constructs a causal pathway linking the initial symptom to the faults that caused it we call this type of reasoning **Backward Causal Tracing**.

In addition to a symptom, an initial situation may represent an arbitrary event in the DVMT whose effect on the DVMT behavior needs to be simulated. This is done, for example, when the DM needs to see what effects some identified fault, such as a faulty parameter setting or a failed hardware component, has on the DVMT system. This type of simulation thus consists of forward chaining through the SBM. Here the DM constructs a causal pathway which links the initial situation to all situations caused by it. We therefore call this type of reasoning **Forward Causal Tracing**.

We will be following the convention of representing both symptoms and faults (i.e., any undesirable situations) by false states, in the case of predicate states, and by lower-than or higher-than, in the case of relationship states. For example,

a lack of some hypothesis at the vehicle track (vt) level will thus be represented by a false state VT which will have an associated abstracted object with attribute values representing the vt hypothesis.

§2. A Dammed Metaphor

Before we begin discussion of the different reasoning types, let us look briefly at diagnosis in a more familiar domain. Suppose we have a system of interconnected canals like that shown in Figure 26. There are several springs which feed water into the system and there are several outlets where the water is expected to arrive. At various points in between the sources and the outlets there are dams, which can be either closed or open. We can observe whether there is water at the springs and at the outlets. We can also check each dam individually to see whether it is open or closed.

This diagram is analogous to the SBM. The springs (labeled A,B, and C) are analogous to the sources of data in the DVMT (the sensors and the communication channels). The outlet points (W,T,S, and R) are analogous to the output hypothesis in the DVMT. The flow of water through the canals is analogous the flow of data (signals) through the blackboard levels in the DVMT. Not all canals may be used, since some of the dams might be closed. Similarly, not all the pathways through the blackboard levels may be used, since they may not be within a node's interest area. We will adopt the convention that absence of water at some point makes that point false; presence of water makes the point

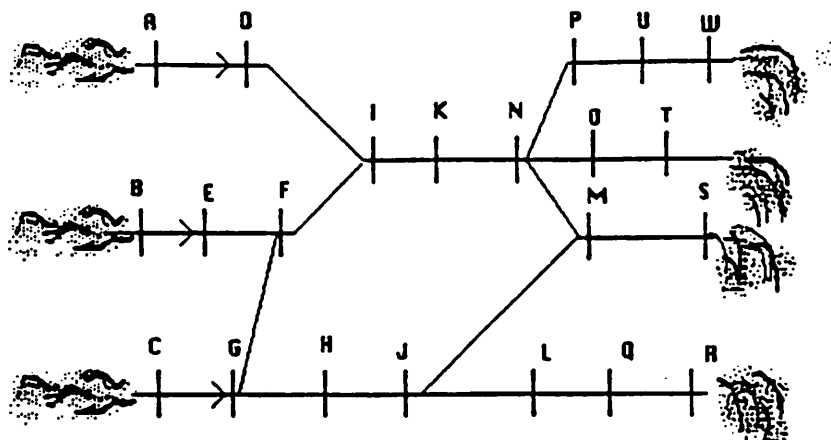


Figure 26: A Canal System Analogy for the DVMT Diagnosis
Reasoning about the flow of water through this network of canals and dams is similar to reasoning about the "flow of data" through the various intermediate stages in the DVMT system. The points A, B, and C represent the sources of water. The points W, T, S, and R represent the outlets where the water is expected to arrive. The vertical bars represent dams which can block the flow of water.

true. Analogously, in the DVMT, a false state represents the lack of expected data at that point and a true state means that the expected data does exist. The DVMT model is more complex; the data is represented by abstracted objects with many attributes and the model is hierarchical. The canal analogy is appropriate however for getting an intuitive feeling for the problems involved in diagnosing the DVMT system behavior.

Let us call the initially observed symptom data and the identified fault (empty spring or closed dam) a goal. Suppose we expect water at outlet W but there is none. The state representing water at W is therefore false. The problem might be that there is no water at one of the sources, or that one or more dams lying in between W and the sources is closed. How will we go about determining where the problem lies? Given our model of the canal system, there are two obvious

approaches. We can start with the state W and, using the model to determine where to look next, trace back through the connecting canals to see at what point the water stopped flowing. This may take us all the way to the sources, if in fact no water ever came in to the system, or we may find that one or more of the dams are closed. The other possibility is to start with an empty source or closed dam and simulate the effect of the resulting lack of water on the rest of the system. The correct combination of these problems will produce the same set of symptoms as those initially observed. This will then allow us to conclude that the current set of simulated faults represents the actual faults

These two strategies for diagnosis correspond to backward chaining and forward chaining respectively through the model of the system. Which is the more appropriate choice, like the choice between backward and forward chaining in any problem, depends on the structure of the problem space. We should move in the direction that requires less search to produce a solution. If there are many possible faults and few final symptoms, then backward chaining is more appropriate. In such a situation the simulation of all possible fault combination to see whether they produce the observed symptoms would be too costly. If on the other hand there are many symptoms and few faults, then forward chaining is appropriate. We also can combine the two in some way, and allow the results of one type of search constrain the other search. That is in fact what the DM does in the DVMT diagnosis.

The two primary types of reasoning used by the DM, Backward Causal Tracing and Forward Causal Tracing mentioned in the previous section, are analogous to

the backward and forward chaining above. In Backward Causal Tracing the DM looks for the causes of some symptom, using the SBM to guide it where to look, and tries to reconstruct how the data actually did flow through the system in order to find what the problems are. Perhaps the data never arrived or perhaps the interest area parameters did not allow the data to get to the desired point. DM begins diagnosis with Backward Causal Tracing since going backward requires less search than going forward from the faults. Once a fault has been identified, Forward Causal Tracing is used to do propagate its effects forward and thereby account for all generated symptoms. This is analogous to the forward chaining above. Notice that the DM does not attempt to find the faults by simulating each possible combination. This would be prohibitive due to the large number of such combinations even for a single DVMT symptom. Instead the more constrained Backward Causal Tracing is used to identify a fault, and only then does it make sense to simulate its effects through the model. At that point this is no longer a search, as it would be if simulation was used for fault identification, but rather a highly constrained simulation of a known situation, the identified fault.

So far, in both the canal example and the DM, we have assumed that we can examine any intermediate point in the system to see whether it is behaving as expected. In the canal example, this means examining the dams. In the DVMT, this corresponds to being able to determine whether some arbitrary intermediate state was achieved, by examining its data structures. In most cases this is indeed possible. There are however a few instances where the DVMT system does not keep track of its intermediate results. In these cases, as in the cases where the

dams could not be examined directly, we must use other methods to determine what the intermediate states are in the system. The method we use is to examine the points surrounding the one we are interested in and try and deduce what the intermediate state is from the surrounding states. In the canal system, if a dam is closed, we can assume that there will be no water forward of this dam, unless some open canal converges onto the empty one at some point. Similarly, if a canal has water then we can assume that all dams prior to it are open and the source has water in it, unless, again, some other pathway converges onto the one in question. These relationships make it possible to determine the value of a current state without having to examine the current state directly. We call this type of reasoning Unknown Value Derivation. This is the type of reasoning used by DART [15] to diagnose faults in digital circuits. This type of reasoning is also used in a digital circuit fault analyzer built by Davis [10]. He calls it "interaction between diagnosis and simulation", since it involves going backward and forward through the model as the surrounding states are explored.

The rest of this chapter discusses in detail the different types of reasoning used by the DM. These are summarized below.

BACKWARD CAUSAL TRACING (BCT) is used to diagnose why a desired situation did not occur by comparing the behavior necessary for its occurrence, as represented by the instantiated model, with what actually did occur in the problem-solving system, as determined from the DVMT data structures. The aim is to construct a path from the symptom state to some false primitive states which were responsible for the symptom and thus explain, in terms of the primitive causes, why the DVMT system did

not behave as expected.

FORWARD CAUSAL TRACING (FCT): is used to reason about the effects of an identified fault on future system behavior. FCT uses underconstrained objects to reason about the class of problems caused by the fault, rather than just the individual cases.

COMPARATIVE REASONING (CR): is used to compare two situations in the DVMT system and to explain why they are different.

UNKNOWN VALUE DERIVATION (UVD): is used to determine the whether some intermediate system state occurred by examining surrounding states. This type of reasoning is used when the state value cannot be determined directly from the DVMT system data structures.

RESOLVING INCONSISTENCIES: is used to decide which of several possible faults is most consistent with the overall system behavior. This is done by comparing each of the candidate faulty objects to a model of an ideal object.

Figure 27 illustrates the relationship among these types of reasoning, the simple backward and forward chaining, and the mechanism for propagating values through the constraint expressions. The following sections discuss each of the reasoning types, how they are used, what they accomplish, and what their limitations are.

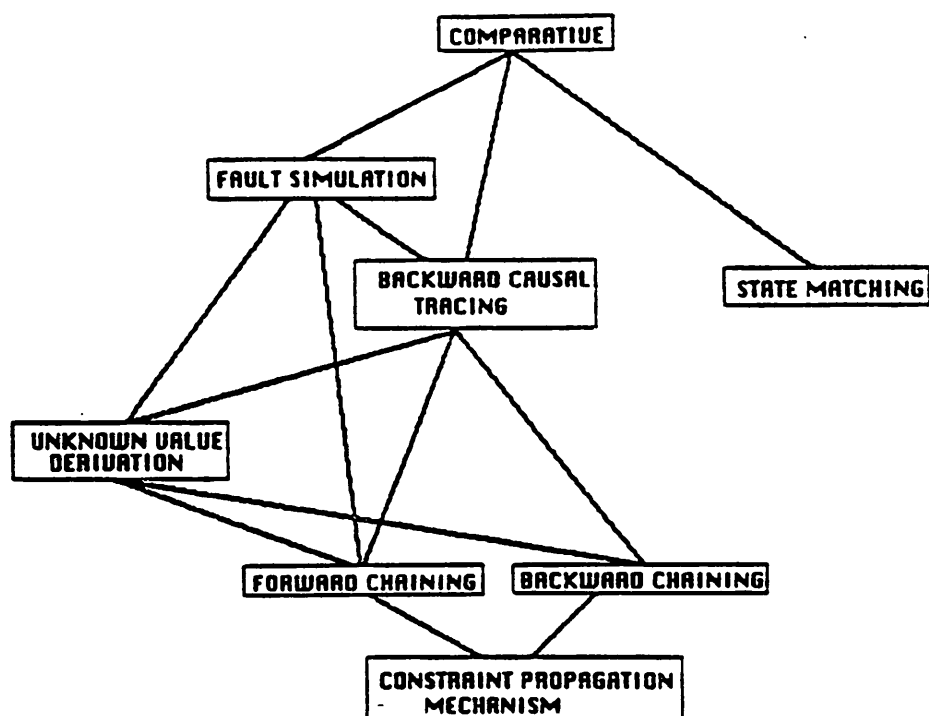


Figure 27: The Types of Reasoning Used by the DM

At the lowest level is the mechanism for propagating values through the constraint expressions linking the abstracted objects. Propagating these values forward through the SBM constitutes forward chaining. Propagating them backward constitutes backward chaining. The more complex types of reasoning consist of combinations of these simpler ones as shown in the diagram.

§3. Backward Causal Tracing

BCT is the primary diagnostic reasoning used by the DM. Recall that the aim of BCT is to construct a causal pathway connecting the observed symptom with the primitive causes. BCT begins with the arrival of a symptom, a false state representing some desired situation in the DVMT which never occurred, and uses the System Behavior Model to guide its search through the space of possible DVMT behaviors. Given some symptom, represented by a state and its abstracted object, BCT first instantiates the back neighbors of that state. This results in the creation of more states and abstracted objects, each such pair representing a situation that would have to occur in order for the situation represented by the symptom to be true.

Consider the examples in Figure 28. Part A represents the lack of some pattern track (pt) hypothesis, represented by a false PT state. (The abstracted objects are not shown in the figure.) How could such a hypothesis be derived by the DVMT? Looking at the back neighbors of this state tells us that either one of the states MESSAGE-ACCEPTED, PL, or PT could have led to the creation of the desired hypothesis. BCT therefore instantiates each of these states, along with their corresponding abstracted objects, which characterize the type of hypotheses from which the desired pt hypothesis could be derived. Having instantiated the potential precursor hypotheses, BCT examines the DVMT data blackboard to see whether such hypotheses were in fact created. If they do exist in the DVMT, then the corresponding states are marked true. If not, the states are marked false.

In the example in Figure 28 we see that the desired hypothesis was received as a message from another node and accepted. We also see that deriving the pt hypothesis via the other two possible pathways, represented by the pattern location (state PL) and by shorter pattern tracks (state PT), is not possible, since both of those states are false.

The example above illustrated use of BCT with predicate states. Part B of Figure 28 illustrates causal relationships among relationship states. Recall that relationship states are used in qualitative reasoning to represent whether some object rating is lower-than, equal-to, or higher-than some other object rating, serving as a model. The figure shows an example where the rating of a ksi is higher than the rating of the model. (The model ksi rating is not shown in the figure.) BCT expands the back neighbors of the state KSI-RATING, which represent the components affecting the rating of the ksi. Each of these components is then compared with the corresponding component of the model ksi, resulting in the relationships shown in the figure: the goal component ratings are equal, the data component is lower than the model data component, and the ks goodness component is higher. Thus the only component that explains the high ksi rating is the high ks goodness. This type of reasoning will be discussed in more detail when Comparative Reasoning is described later in this chapter. For the rest of this section, we will focus on BCT with predicate states; that is, states which are true or false.

BCT behaves differently, depending on whether a predecessor state explains the current state (i.e., is causally related to it) or not. In the example, the true

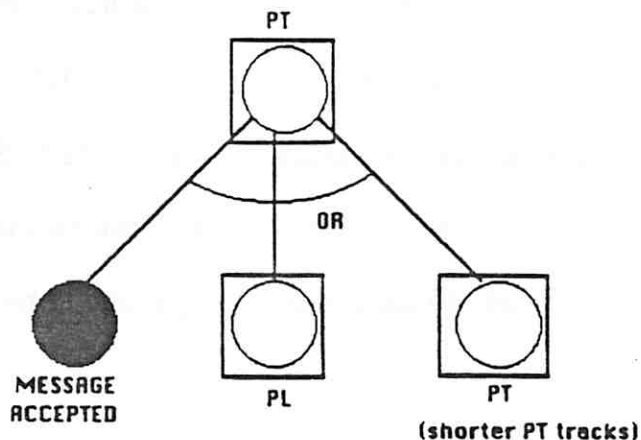
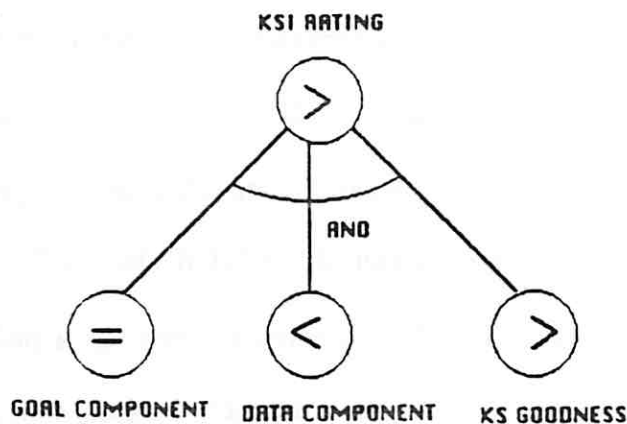
PART APART B

Figure 28: Causal Relationships Among States

Part A shows an instantiated portion of the SBM which represents how a pattern track hypothesis (state PT) can be derived from three different sources: a message from another node, pattern locations, or shorter pattern tracks. Part B shows an instantiated portion of the SBM illustrating causal relationships among relationship states. In this case, the high ksi rating could have been caused by the high ks goodness, but not by the other two states.

state MESSAGE-ACCEPTED does not explain the false state PT; if a message was accepted then the desired hypothesis should have been produced. The false states PL and PT, on the other hand, do explain the false PT state; if no pattern locations exist, then no pattern track can be created. Similarly, if no shorter pattern tracks exist, then the longer pt hypothesis cannot be produced. Figure 29 shows more examples of causal relationships among both predicate states and relationship states.

BCT treats these cases differently. When it encounters a state that is not causally related, while expanding the back neighbors, it switches to a more detailed model cluster, one that represents the events in between the two states. BCT then recursively expands the forward neighbors, again looking for causally related states. This time however, since BCT is now propagating the effects of a true situation forward, a causally related state will be a true state. Again, BCT looks for a break in the expected processing; a point where a true state is followed by a false state. At such a point it again switches to backward chaining, expanding the back neighbors of the false state, as it attempts to find the primitive states responsible for the original symptom. BCT thus always looks for a break in the expected processing. In predicate states, such a break is always represented by a true state followed by a false state; i.e., an expected situation followed by an unexpected situation. In relationship states, such a break is represented by neighboring states whose values do not agree; e.g., they could be any of the following: $\langle - \rangle$, $\langle - = \rangle$, $\langle - \langle \rangle$, etc. Figure 30 illustrates the search strategy BCT imposes on the model instantiation. Figure 31 shows the algorithm for BCT.

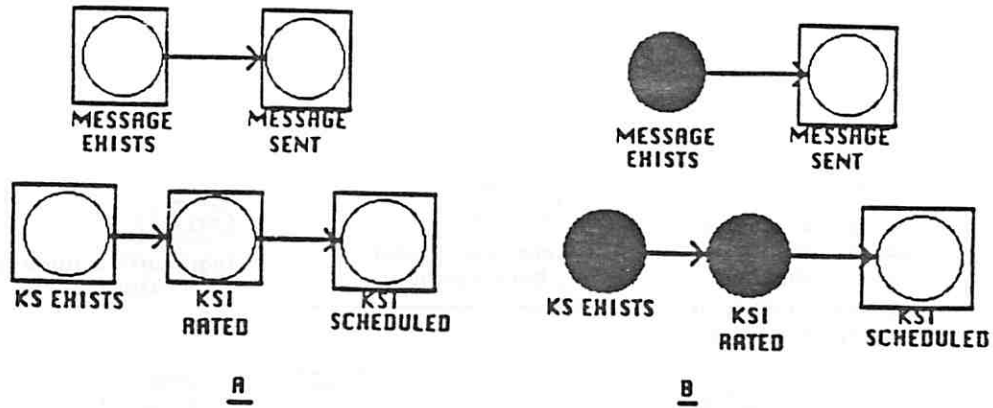
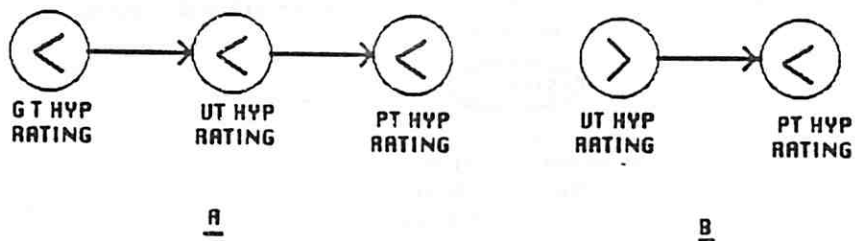
PART IPART II

Figure 29: Causal Pathways in the Instantiated SBM

Part IA illustrates causally related predicate states; part IIA illustrates causally related relationship states. In contrast, part IB and part IIB illustrate cases where states are not causally related. When these cases occur, the DM must look for the cause in between such true-false (in predicate states) or low-high (in relationship states) pairs.

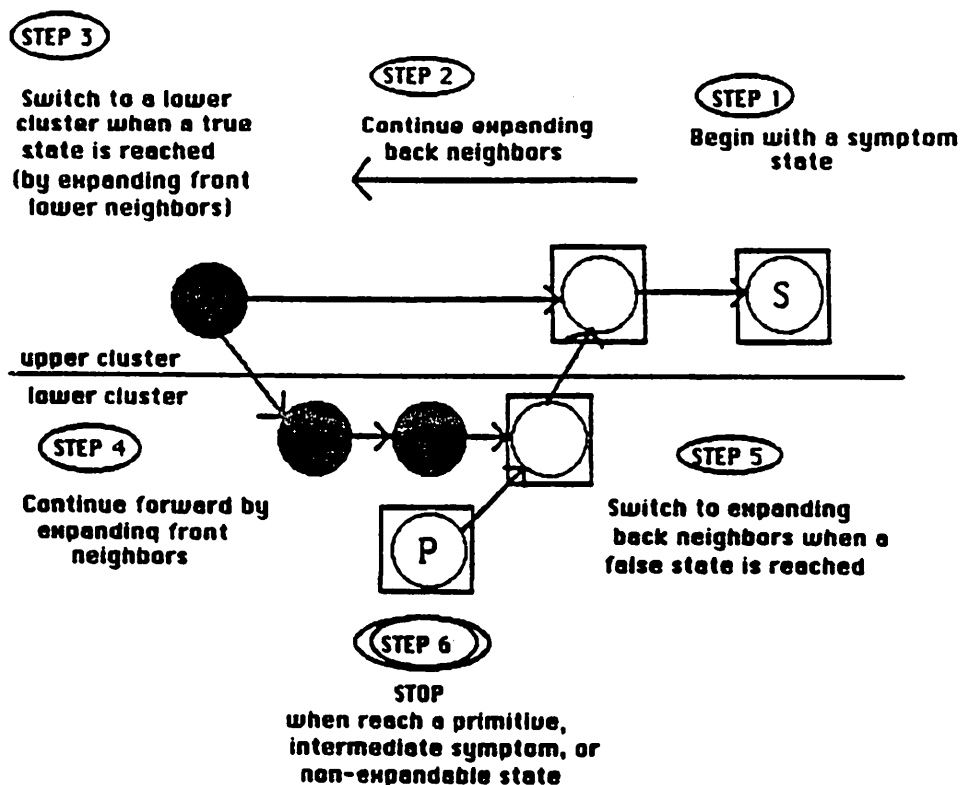


Figure 30: BCT Search Strategy

This figure illustrates a portion of the instantiated SBM which represents diagnosis of some symptom state S by BCT. BCT begins with the expansion of the back neighbors until a true state is reached. At that point BCT switches to a more detailed model at a lower level of the SBM hierarchy, and continues expanding the front neighbors of each state, until a false state is reached. This process continues until either a reportable failure is found or until the model cannot be expanded.

ALGORITHM FOR BCT

1. **BEGIN:** with an abnormal (false for predicate states; low or high for relationship states) symptom state, represented by an evaluated state and its abstracted object. Enter backward chaining mode.
2. **WHILE** (in backward chaining mode) **DO:**
 - (a) Instantiate state's back neighbors. If state has no back neighbors, instantiate upper back neighbors. Evaluate all instantiated states. For each causally related new state **DO:**
 - i. If state is a primitive state, report identified fault. **STOP.**
 - ii. If state is a non-expandable state, report identified fault. **STOP.** (Whether a non-expandable state is an actual fault cannot be determined until after the path-value has been computed. Locally however, it is treated as a fault.)
 - iii. If state is an intermediate symptom state, save as pending symptom for further diagnosis. **STOP.**
 - (b) If a state is not causally related, enter forward chaining mode by expanding lower front neighbors of state. If state does not have lower front neighbors, report unsuccessful diagnosis. Model does not provide sufficient detail to identify a fault. **STOP.**
3. **WHILE** (in forward chaining mode) **DO:**
 - (a) Instantiate state's front neighbors. If state has no front neighbors, instantiate upper front neighbors. Evaluate all instantiated states.
 - i. If a state is not causally related, enter backward chaining mode starting with the new state.

Figure 31: Algorithm for Backward Causal Tracing

When successful, BCT explains an observed symptom in terms of reportable failures. Reportable failures fall into three categories. They can be false states, which have been designated as primitive by the model builder. For example, the states KS-EXISTS, DATA-EXISTS, SENSOR-OK, and CHANNEL-OK all represent primitive states. A reportable failure can also be a non-expandable state which represents a situation that could not be reached because of the current parameter settings. An example of this type of a failure is when the lack of some pattern track hypothesis is traced to the node's interest area parameters not allowing the creation of the necessary precursor hypothesis. For example, no vehicle location hypotheses can be created at the node but pattern location hypotheses are expected. Finally, a reportable failure can be an intermediate symptom state. (These states were discussed in Chapter III.) Any state in the SBM can be labeled as an intermediate symptom state. Such states serve as break points in the diagnosis. When a false intermediate symptom state is reached, it can be reported as an identified failure. This means that this intermediate symptom successfully explains the symptom currently being diagnosed. The intermediate symptom is then saved for further diagnosis. Figure 32, part A illustrates these BCT terminating conditions.

Note that each reportable failure state (that is, a primitive false state, a non-expandable state, or a false intermediate symptom state) represents only a local failure. In other words, these states might not have any effect globally because there may be other ways of reaching the desired state. In order to determine whether a fault state is in fact responsible for the initial symptom the DM uses

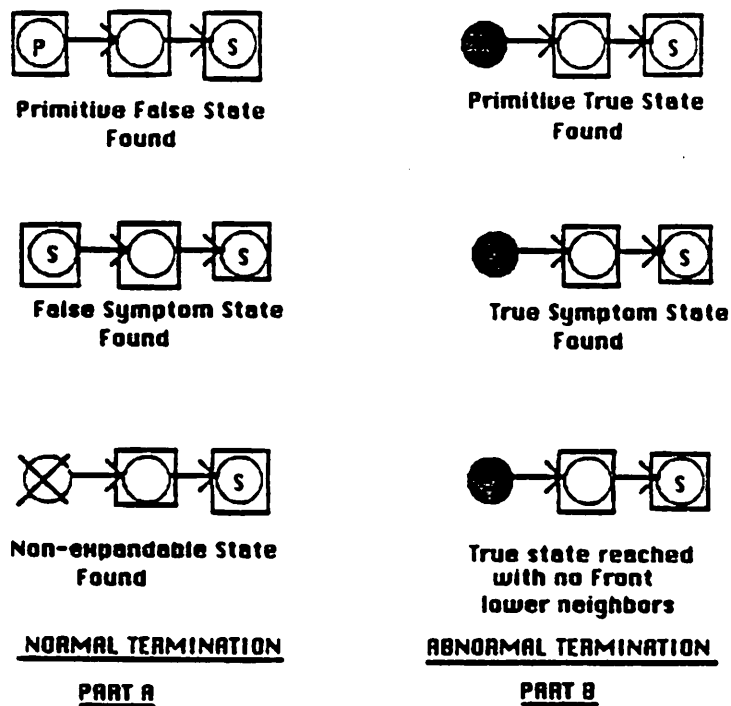


Figure 32: BCT Terminating Conditions

Part A illustrates normal terminating conditions. These are cases where BCT successfully explains the initial symptom in terms of primitive causes. Part B illustrates abnormal terminating conditions. These are cases where BCT was not successful in identifying the primitive causes but did manage to narrow down the general problem area.

the notion of a **path value**. A path value represents for each pathway in the model whether a fault lies along that pathway. The path values are combined according to the logical connections among states to determine whether a state's value is consistent with its neighbor states' values. Section 10 discusses the use of path values in more detail.

BCT may not always be successful in finding an explanation for the initial symptom. This occurs when a state is reached that is not causally related to the current state, i.e., a break in the expected processing is found, but no lower level model exists expanding the events in between the normal and the abnormal states. In predicate states, this situation is represented by a true-false state pair, in relationship states it is represented by neighboring states with values that are not consistent. Figure 32, part B illustrates the abnormal terminating conditions for BCT. Even in cases where BCT does not identify a failure, it is still helpful, since it considerably narrows down the space of possible problems into the area in between the normal state and the closest abnormal state. Assuming the SBM is correct, this in itself is great help in diagnosis, especially in situations where the diagnostic module is used during system development to aid in debugging.

We thus distinguish between normal terminating conditions, those successfully explaining the symptom, and abnormal terminating conditions, those that narrow down the problem but cannot identify it in terms of reportable states. Given the convention that abnormal means false for predicate states, and either too low or too high for relationship states, a normal terminating condition is one which results in an overall successful diagnosis, where the symptom is explained in terms

of primitive abnormal states. An abnormal terminating condition is one where no such states could be found and the observed symptom cannot be explained in terms of the primitive states.

To summarize, BCT constructs its causal pathway by recursively expanding the back neighbors of the symptom state, until a break in the causal pathway is reached. Figure 33 illustrates a causal pathway produced by the diagnosis of a missing vt hypothesis 1-3. Given our convention that undesirable situations are represented by false states and desirable situations by true states, such a break occurs when a true state is reached. Once this occurs, BCT switches to a model that represents in more detail the events in between the true and false states and recursively expands front neighbors of the true state, again looking for a break in the causal pathway, this time represented by a false state. This type of search continues until either a false state is found, that can be reported as a failure (primitive state, non-expandable state, or a symptom state), or until the model cannot be expanded any further. BCT thus duplicates aspects of the DVMT system behavior, simulating its processing at an abstract level, by propagating the values of the initial symptom state through the SBM via the constraint expressions linking the abstracted object attributes. BCT can be used with both predicate and with relationship states. When used with predicate states, BCT explains why some desired situation, represented by a false state, was never achieved by the DVMT system. When used with relationship states, BCT explains why the rating of some object was abnormally high or abnormally low as compared with the rating of another object. (Its use with relationship

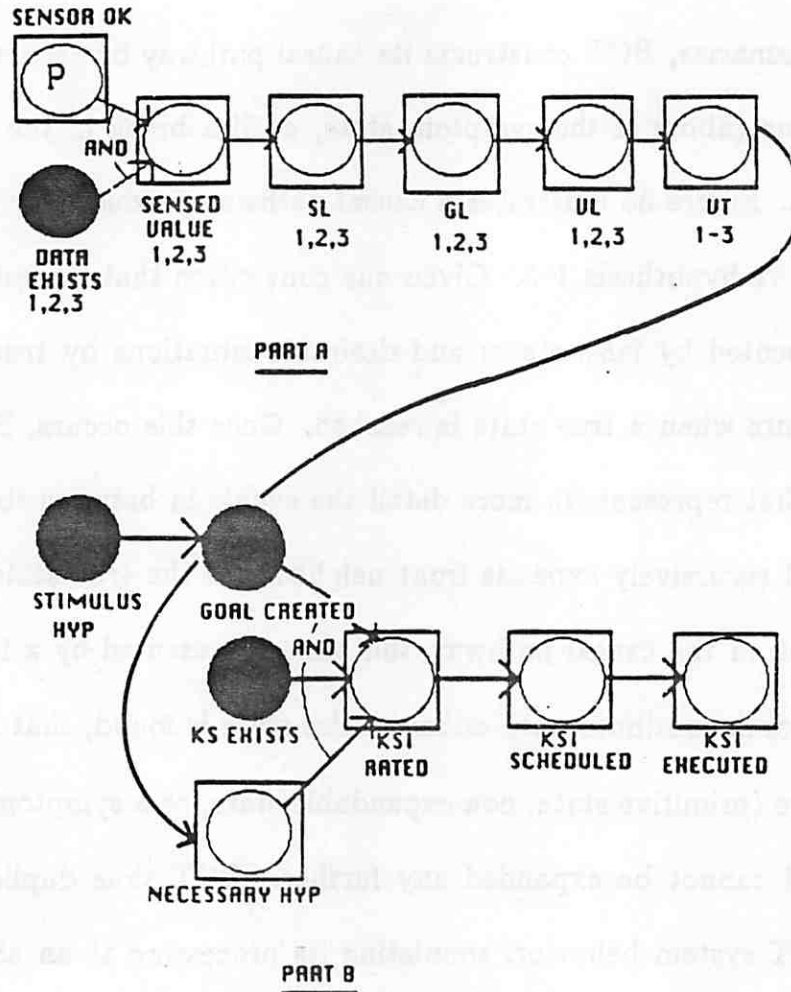


Figure 33: BCT-Constructed Causal Pathways

Part A shows an instantiated Answer Derivation Cluster, where BCT traced the lack of a vehicle track (vt) 1-3 hypothesis to a failed sensor. Part B shows an instantiated Ksi Scheduling Cluster, where BCT, using forward chaining, determined that the reason some stimulus hypothesis was not extended, was due to the missing vt 1-3 hypothesis, represented by the false NECESSARY-HYP state.

states thus requires the use of Comparative Reasoning, which is discussed later in this chapter.)

§4. Forward Causal Tracing

FCT is used to simulate the effect some situation has on the DVMT system behavior. In principle, the effects of any state could be propagated forward through the model; predicate or relationship, primitive or not. Currently however FCT is used to simulate the effects that an identified fault, represented by a false primitive state, will have on system behavior. This is useful because normally one fault will have caused a number of symptoms. For example:

- a bad sensor means that no data will be sensed in the area sensed exclusively by the failed sensor,
- a bad channel means that no messages will be received from the node linked by that channel, and
- a missing knowledge source means that no hypotheses will be derived by that knowledge source.

Once a fault has been identified, it is more efficient to simulate its effects forward and thereby account for any pending symptoms, than to individually diagnose all symptoms. Some of them may already have been reported and are awaiting diagnosis, these are the pending symptoms. Others may be reported in the future and will become symptoms then. For example, FCT will predict that no messages will be transmitted among two nodes if their communication channel has failed or if the communication knowledge sources are missing. Accounting for the effects of an identified fault thus saves time, not only because it accounts for pending

symptoms, which have not yet been diagnosed, but also because it accounts for any future symptoms that will be caused by the failure, until it is corrected. In order to increase efficiency, FCT uses underconstrained objects to represent an entire class of objects affected by the fault.

FCT begins with an identified failure, a false primitive state such as missing data or bad sensor, and uses forward chaining to propagate its effects through the SBM. This is done by recursively expanding the front neighbors of each state, until either a break in the causal relationship is found or until the end of the model is reached. When is the new state not caused by the current state? The obvious case is when the newly expanded state is true, while the current state is false. In such a case some other way must exist to achieve the true state. This is the case with all linked by the OR connective. Recall how in Figure 28, part A the pattern track hypothesis could be derived in any one of three ways: via a message from another node, via its constituent pattern locations, and via shorter track segments. It could thus happen that one of the OR states could be false, due to some failure, but its front neighbor could be true, because it was achieved via another pathway.

There is a more subtle case however, where even a false state may not be causally related to a preceding false state. Consider again the example in Figure 28, part A. Suppose we are simulating the effect of missing data on the creation of the pattern track hypothesis represented by the state PT. In order to explain this state, and any related states, in terms of the fault, we first have to guarantee that it is in fact not achievable in any other way. In the example, this is clearly not

the case, since the PT state could be achieved via the MESSAGE-ACCEPTED pathway. In fact, the state MESSAGE-ACCEPTED is true. The only time we can be sure that a false state will never be achieved due to some failure, is if we can prove that it is not achievable in any other way. This situation is called the converging OR problem. Whenever it is encountered, the DM considers the false state as a symptom and invokes BCT to determine whether this false state is achievable in the current system configuration. In the process of determining whether the state can be reached in multiple ways, BCT may uncover more faults.

Having determined that a state was caused by a fault and that it cannot be achieved in any other way, FCT checks whether it is on the list of pending symptom states. If it is, it is removed from this list and is considered explained by the identified fault. FCT ends when the end of the initial state's causal pathway has been reached. Figure 34 shows the search strategy imposed on the model instantiation by FCT. Figure 35 shows the algorithm for Forward Causal Tracing.

FCT is made more complicated because use of underconstrained objects is necessary to simulate the effect of a fault. This is because we cannot know the exact characteristics of the objects affected by a fault. Consider the following example. A fault has been found that explains why no messages (specifically, no hypotheses on the vt level) have been sent out from a node. The fault is a missing knowledge source, HYP-SEND:VT. We would now like to account for all the currently pending symptoms related to this fault. Namely, all the false MESSAGE-SENT states and their associated objects. Not only that, we also want to account for

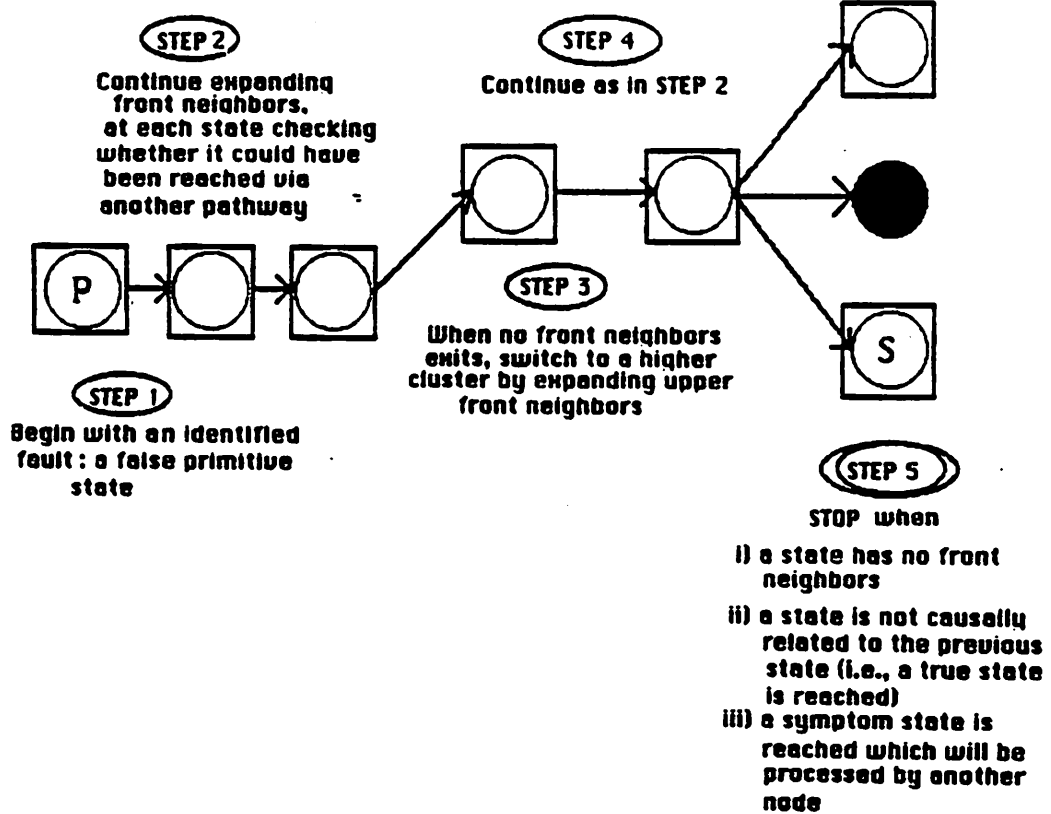


Figure 34: Search Strategy Imposed on the SBM Instantiation by FCT

FCT begins with a false primitive state, representing an identified fault, and continues expanding front neighbors of each causally related state. At each point, FCT invokes BCT to make sure that the state cannot be achieved in any other way given the current system configuration. Each such state can then be considered explained by the identified fault. FCT ends when it cannot find any more causally related states or when the model ends.

1. **BEGIN:** with a false predicate state representing an identified fault.
2. **Instantiate the state's front neighbors.** If state has no front neighbors, instantiate upper front neighbors. Instantiate underconstrained objects corresponding to the objects affected by the identified fault. Evaluate all newly created states. For each state **DO:**
 - (a) If state is true, causal pathway ends. **STOP.**
 - (b) If the state is false, invoke **BCT** to make sure it cannot be achieved via any other pathways. If it can be achieved, **STOP.** If not, explain all overlapping states (i.e., states whose objects overlap with the underconstrained objects), which represent pending symptoms as results of the fault being simulated. Go to 2.

Figure 35: Algorithm for Forward Causal Tracing

any future **MESSAGE-SENT** symptoms. We therefore initiate simulation reasoning with the false primitive state **KS-EXISTS** and its abstracted object, **KS-OB**, which represents the knowledge source **HYP-SEND:VT**. Notice what happens when we evaluate the object knowledge-source-instantiation, **KSI-OB**, associated with the successor state of the state **KS-EXISTS**, **KSI-RATED**. Not all of this object's attributes will be definable in terms of the known **KS-OB** attributes. In fact, most of them will not be. The only attributes we can determine are **KS-TYPE (HYP-SEND:VT)**, **MESSAGE-TYPE (HYP-OB)**, and **NODE**. The other attributes, such as rating, or the exact characteristics of the message, cannot be evaluated from the **KS-OB** attributes and thus remain unknown, or "don't care". This results in the **KSI-OB** being underconstrained and therefore representing not just one specific instantiation of the **HYP-SEND:VT** knowledge source but rather the entire class of all the instantiations of that knowledge source. This is exactly

the effect we want to achieve since we can now account for all the symptoms involving the knowledge source HYP-SEND:VT. The use of underconstrained objects in FCT is illustrated in the 4-node example in Chapter V.

Another example of the need for underconstrained objects is the simulation of the effects of an identified failed sensor. In this case we want to be able to explain any pending symptoms due to the missing data from the area sensed exclusively by the failed sensor. We also want to account for any such future symptoms. The sensor object characteristics are: event-classes, time-region-list, and nodes. We cannot define all the possible hypotheses which could be generated from the data in this area, since we would have to define all possible combinations of times, locations, and event-classes. We therefore propagate forward the entire sensor area, time list, and event class set. This then results not in a large number of hypotheses but rather in one underconstrained hypothesis object for each level. As we progress forward through the answer derivation model we look for any symptoms that fall within the space/time/event-class area of the underconstrained object representing the class of hypotheses derivable from the data in the failed sensor area. Provided the converging OR problem is solved satisfactorily, i.e., there are no other ways to achieve any of those states, all those symptoms are explained by the failed sensor whose effects we are simulating forward. The use of underconstrained objects is thus both practical (in reducing the number of objects we need to create), and necessary (since often we cannot know what the objects' characteristics would be).

§5. Comparative Reasoning

Comparative Reasoning (CR) is a heuristic used to focus in on a potential problem by comparing the a problem situation with a desired situation in the DVMT system. CR is used to compare the behavior of two objects in the DVMT. In principle, it could be used with predicate or relationship states. Currently however, it is used with relationship states only, to compare the derivation paths of the rating attributes of two different objects. For example, the different ratings of two knowledge source instantiations (k_{si}'s) or the different ratings of two hypotheses. CR first reconstructs how the rating of each of the objects was derived, by representing this derivation process in terms of the instantiated system behavior model. This is simply a translation from one representation (the DVMT system data structures) into another (the instantiated SBM). The models of the two object rating derivations are then compared, state by state, in order to understand why they differ; i.e., why one is lower or higher than the other. The aim of CR is to determine the cause of an abnormal rating in terms of the abnormal values of some primitive component, such as the lowest level hypothesis or some parameter. Relationship states are used here to represent the relationship among the two object ratings.

Why do we need CR? The types of reasoning described so far involved comparing the actual DVMT behavior with the correct behavior, as represented by the SBM. There are events in the DVMT system however, for which no absolute criteria exist and which cannot therefore be represented by a fixed model. For

example, we cannot say that a ksi rating is too low, because there is no absolute standard with which to compare it. Its "lowness" or "highness" is only relative to the ratings of the other ksi's on the queue. The actual value of the rating is not important since it is the highest rated knowledge source that will execute.

It is this lack of absolute standards for system behavior that necessitates CR because we cannot always understand the system behavior by looking at an isolated object in the system and comparing that object to a fixed, absolute standard. What we need to do instead is to dynamically choose a model in the form of an object that did achieve some desired situation. In investigating why the problem object differs from the model object, CR often uncovers a fault. Note that Comparative Reasoning is a heuristic and does not always lead to fault identification. What it provides is an explanation of why one object rating is lower (or higher) than another. It would require a higher level model (one with a more global view of the situation) to decide whether the explanation found is in fact a bona fide fault or not. For example, when a low sensor weight parameter causes all data from its area to be rated low, does not necessarily imply there is a problem. Perhaps that area is sensed redundantly by several sensors and it is appropriate that all but one of these should weigh their data low, so that the DVMT system does not spend all its time doing redundant processing. At this point what CR can do is provide an explanation of something in terms of such local causes; i.e. low sensor weight parameter. In order to determine whether this is in fact a problem, the DM would have to examine all sensors sensing that area and compare their weight parameters to determine whether the specific low

weight is in fact a problem.

CR is invoked whenever a state is reached that represents a situation requiring the use of a dynamic model. This information is contained in the SBM by marking certain states as CR transition states by the model builder. Currently, this occurs when the rating of a knowledge source instantiation is too low and prevents the knowledge source from being either scheduled or executed. CR begins by selecting the model object with which the problem object will be compared. (This selection process is described below when situation matching is discussed.) The initial states in both models represent the rating of some object. CR first determines the relationship among the ratings of the two parallel states; i.e., less-than ($<$), equal ($=$), or greater-than ($>$). Depending on the situation being analyzed, the problem object is either higher or lower than the model object. It is thus abnormally high or abnormally low as far as the model object is concerned. The aim of CR is to trace back through the two models, comparing the various intermediate ratings of the objects, in order to find primitive states that are $<$ or $>$, which would explain the abnormal rating of the initial object. Like the other reasoning types discussed so far, CR constructs a causal pathway linking causally related relationship states. In this case states are causally related if an abnormal value of one could have caused the abnormal value of the other. CR employs BCT mechanisms to determine which neighbors are causally related and to control which will be expanded further. Figure 36 describes the algorithm for CR.

In order to determine which states are causally related, the problem states' value attribute must be calculated. This attribute represents the relationship

1. **BEGIN** by selecting an initial object to use as a **MODEL** for **PROBLEM** object.
2. **LINK** the problem and model objects and the problem and model states.
3. **INSTANTIATE** the back neighbors of both the problem state and the model state.
4. For each resulting **PROBLEM** object, select a **PARALLEL MODEL** object from among the just instantiated **MODEL** objects, according to the state matching specifications associated with each state. If no parallel object/state can be found, **STOP**.
5. **LINK** the problem and model objects and the problem and model states.
6. **DETERMINE** the relationship among the objects by comparing their ratings. This will result in less-than (<), equal-to (=), or greater-than (>).
7. Of the newly expanded states, both **PROBLEM** and **MODEL**, select those that are **CAUSALLY** related to the existing states. States are causally related if both have the same type of relationship. (I.e., if the existing state was <, then all causally linked states will be <; similarly for >).
8. If a **PRIMITIVE** state has been reached, report identified failure. **STOP**. If state is not causally related, **STOP**.
9. **GO TO** 4.

Figure 38: Algorithm for CR

CR is invoked whenever a state is reached in the model that is marked as a CR transition state. Currently there are four such states: KSI-RATING-OK, KSI-RATING-MAX, COMM-KSI-RATING-OK, and COMM-KSI-RATING-MAX.

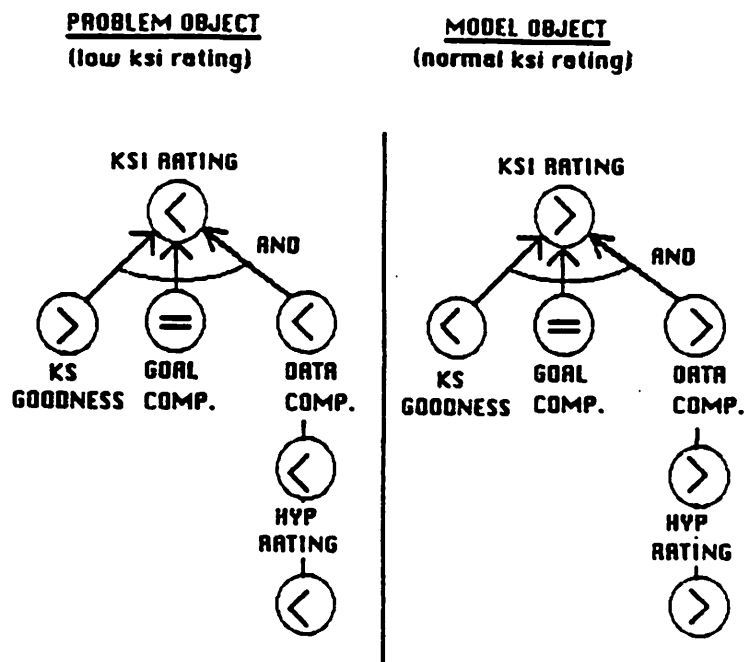


Figure 37: Example of CR

This figure illustrates the use of CR. The left half of the figure shows the instantiated model for deriving the rating of the ksi being diagnosed. The right half shows the instantiated model for deriving the rating of the ksi chosen as a model. The two instantiated models are compared, state by state, and their relative values, i.e., $<$, $=$, or $>$, are determined. The aim is to explain the low rating in terms of a low rated primitive state.

among the problem state and its parallel model state. In order to determine the relative value, we need to find a matching state for the problem state. The state matching is complex because we must make sure we are not comparing apples and oranges during CR. Next section discusses some aspects of this problem and the reasoning it uses, **Situation Matching Reasoning**. In some cases, no state can be found with which to compare the problem state. In such cases CR terminates abnormally.

Figure 37 illustrates how CR is used to analyze why the rating of one knowledge source instantiation (ksi) is lower than the rating of another ksi. The rating

of a ksi is a function of three components: the knowledge source (ks) goodness value, the goal component rating, and the data component rating. In order for the ksi rating to be abnormally low, at least one of these components must be low. Suppose that the relative values of the three components are high, equal, and low respectively. In other words, the ks goodness value is higher than that of its parallel state, the goal component rating is identical to its parallel state (i.e., to the goal component of the model ksi), and the data component rating is lower than the data component of the model ksi. Since the only state that is abnormally low is the DATA-COMPONENT state, this is the only state that can be causally linked with the original low KSI-RATING state. The construction of the causal pathway therefore continues by investigating why the data component rating is low. This may eventually lead to discovering that the input data was too low or that a some sensor parameter was set too low and caused the sensor to weigh its data lower. The terminating conditions for CR are the same as the ones for BCT; i.e., end of model (either due to lack of a more detailed model, a primitive state, or a symptom state) or end of the causal pathway.

To summarize, CR involves reasoning not just about the behavior of an individual object but rather about the relationship among those objects; that is lower-than, equal-to, and greater-than. CR first chooses an object to use as a model for the problem object. It then reconstructs the derivation pathways for the two object ratings, by translating the internal DVMT representations of these pathways into the SBM formalism. The resulting instantiated models are then compared by matching up parallel states representing the intermediate states in

the rating derivation and determining the relationship between the problem state and its parallel model state. This relationship is used to link causally related states. CR stops when no causal connections can be made or when a primitive state is reached.

Currently there are three possible relationships among the objects, lower-than, equal-to, and higher-than, and they are restricted to relating the values of the object ratings. Chapter X discusses some ideas about how both the types of relationships and the types of attributes to compare could be extended. For example, we could compare the length of tracks or the size of sensor regions, which would lead to other relationships, such as longer and shorter or larger and smaller.

§6. Situation Matching Reasoning

State matching reasoning (SMR) is necessary for CR to find the initial model object with which to compare the problem object, and then to match up the parallel states during the comparison of the two models. These two processes are similar; the only difference is that the initial model is chosen from among all the DVMT objects whereas the subsequent parallel states are chosen from the states representing the intermediate derivation of the model state.

Why is it difficult to find a model object to compare with the problem object? Consider the following problem. Suppose one of the other reasoning types has reached a problem state where a ksi is rated too low and therefore has not yet

executed. In order to understand what caused the ksi rating to be lower than most of the other ksi's on the scheduling queue, the DM must look for a ksi that is rated high enough to execute. The DM cannot arbitrarily choose the highest rated ksi however, because it might be of a different type. Suppose that the problem ksi is a knowledge source that takes a vehicle location hypothesis and derives the next higher level, pattern location, hypothesis from it (a synthesis knowledge source). However, the highest rated ksi on the queue is a knowledge source that takes two pattern track segments and merges them together (a merge knowledge source). There are good reasons why the merge knowledge source should be rated higher; it is a higher-level knowledge source and it works with tracks, which are rated higher than locations. Both of these factors by themselves increase the rating of this ksi, regardless of the specific data, goal, or ks rating. Comparing such dissimilar objects is not very revealing. It is akin to trying to discover what the problem is with a recreational tennis player's game and a pro's. We can certainly point out many differences in their game, but most of them are not useful because they are a function of the different classes of players rather than of anything specifically wrong with the amateur player. In other words, comparing the two will uncover differences, but these can be simply the general differences between the two classes. In order to increase the chances that the comparison will reveal some real problems, objects within a class of similar objects must be compared. In the above case therefore, we would prefer to find another synthesis ksi which is rated higher than the problem ksi and could therefore be used as a model object.

In order to match two situations in the DVMT system, we need two things:

- A way of describing similar situations (objects and states).
- A way of using these descriptions to find a similar object.

The word "similar" means different things for different objects. It is up to the model designer to determine what the similarity criteria should be and to express this knowledge in a form that could be used by the SMR. We have chosen to express this knowledge declaratively in the system model so that it can be changed easily should the similarity criteria change. Chapter VIII shows in detail how this knowledge is represented and used.

How does the DM go about finding a similar situation given the criteria for similarity? The easiest way is to simply look for an already existing object in the DVMT system data structures. This is a relatively simple process involving the matching of the description characterizing the desired similar object to the various DVMT objects. Unfortunately, this will not always work because there will be cases where no such similar object exists. There are two ways of dealing with this problem:

- relax the matching criteria, or
- try to find an object, which could allow the derivation of the desired object, and calculate that object's attribute values by using the model of the DVMT system.

The first method would involve giving an ordered list of the matching criteria and a specification of how they should be relaxed in case of an unsuccessful match. The second method would involve finding some object and then attempting to find either a precursor object or some resulting object which would satisfy the

matching criteria better. This would involve either BCT or FCT to propagate the known attribute values backward or forward in order to determine the attribute values for an object which could be used as a model object. For example, suppose the rating of some ksi that would produce a desired pt hypothesis is too low to execute. The DM would like to use CR, but there is no comparable ksi in the system which could serve as a model for the low rated ksi. DM could therefore look for some other ksi's, working with data at lower levels, and project their behavior forward, thereby creating a ksi object in the instantiated SBM that could serve as a model for the problematic low rated ksi. Currently, the SMR consists of looking for a similar object, as specified by the criteria in the SBM, which already exists in the DVMT system.

Once CR has chosen a model object with which to compare the problem object it must instantiate the model of the object rating derivation and match up the states in the two model instantiations. This matching up of the parallel states poses the same problems as the choice of the initial model. There may be several states to choose from and the SMR must decide which one to use. The only difference here is that the initial object is chosen from among all the DVMT objects whereas the intermediate parallel objects are chosen from among the intermediate objects of the model. As with the choice of the initial model object, the criteria for the selection of the matching state are expressed declaratively and associated with each state that can be used for CR.

§7. Unknown Value Derivation

All the reasoning types discussed so far relied on being able to evaluate the instantiated states directly from the DVMT system's data structures. Recall that instantiation consists of first deriving the attributes of some state and object, which together describe a situation in the DVMT system, and then determining whether that situation exists in the DVMT system by examining its data structures. Unknown Value derivation is substituted for the second step of the instantiation process, when the information necessary to determine whether a situation exists in the DVMT, is not available in its data structures. In other words, we cannot determine whether some object was created or whether some situation exists by looking at the data and goal blackboards or the knowledge source queues.

Unknown Value Derivation (UVD) is used to determine the value of a state from the surrounding state values, by exploiting the dependencies among these states. UVD is the exception rather than the rule, since most of the information necessary for reasoning about the DVMT is available directly from that system's data structures. A small subset of the states in the SBM however cannot be evaluated by looking at the DVMT system data structures. These states must instead use UVD and are therefore called UVD states. UVD states occur either when the information was never stored in the DVMT system or when that information was never known in the first place. Example of information which was not kept around is whether a message was received or not. In other words, we

know whether a message was accepted, by looking at the blackboard and seeing the message there, but we do not know whether it got across the channel because that information is not maintained by the DVMT system. An example of information that cannot be known by the system is whether some data actually exists in the environment. This can only be inferred from the data existing in the system. There are also cases where we assume we cannot determine some information. For example, we assume we cannot probe a sensor or a channel in order to determine whether it is functioning correctly.

UVD works because of the dependencies among the states captured in the SBM. Because the SBM represents the correct sequence of causally related states, we can say something about a state by looking at what occurred before it or what occurred after it. For example, if a state is false, then we know that all of its successor states will also be false, unless they could have been achieved via another path in the model. Similarly, if a state is true, then we know that all of its predecessor states must be true. (If a state can be achieved via a number of pathways, then at least the states along one of those pathways must all be true.) It is this type of knowledge that is exploited by UVD to determine the state value.

UVD works by first looking at the already instantiated surrounding states and using their values to derive the unknown state value. In many cases surrounding states may not yet be instantiated. UVD then utilizes both backward chaining and forward chaining to first instantiate the neighboring states and then to use their values to determine the unknown state value.

Example Illustrating Unknown Value Derivation. This example will illustrate how the unknown value of the state KSI-RATED is derived from its surrounding states (refer to Figure 38). This is a simple case for UVD because none of the unknown state's neighbors is an unknown state. The strategy for UVD is to look at the values of the various sets of neighbors in order to determine the unknown value. The order of neighbors to look at is fixed by the type of search currently active. (If the search would expand the front neighbors next, UVD will expand those.) Currently, the DM is going forward, expanding the front neighbors of each state. UVD will therefore continue in this fashion and will instantiate the front neighbors of the state KSI-RATED. This state, KSI-SCHEDULED, evaluates to false, which is unfortunately of no use to UVD. The only time UVD could make a statement about the value of KSI-RATED is if the state KSI-SCHEDULED was true. In that case, because there is no other way to achieve this state, KSI-RATED would have to be true. However, if KSI-SCHEDULED is false, nothing can be said about KSI-RATED, because the error could be either before KSI-RATED or in between it and KSI-SCHEDULED.

UVD therefore tries to determine which of these is the case by instantiating the back neighbors of KSI-RATED: the states KS-EXISTS and NECESSARY-HYP. (UVD will look at GOAL-CREATED state but does not need to instantiate it since it is already instantiated.) If either of these states is false, then KSI-RATED is false, since it cannot have been reached by any other pathway. If all of them are true, then KSI-RATED can be assumed to be true because no other states occur in between these and KSI-RATED. In this case, the conditions work out

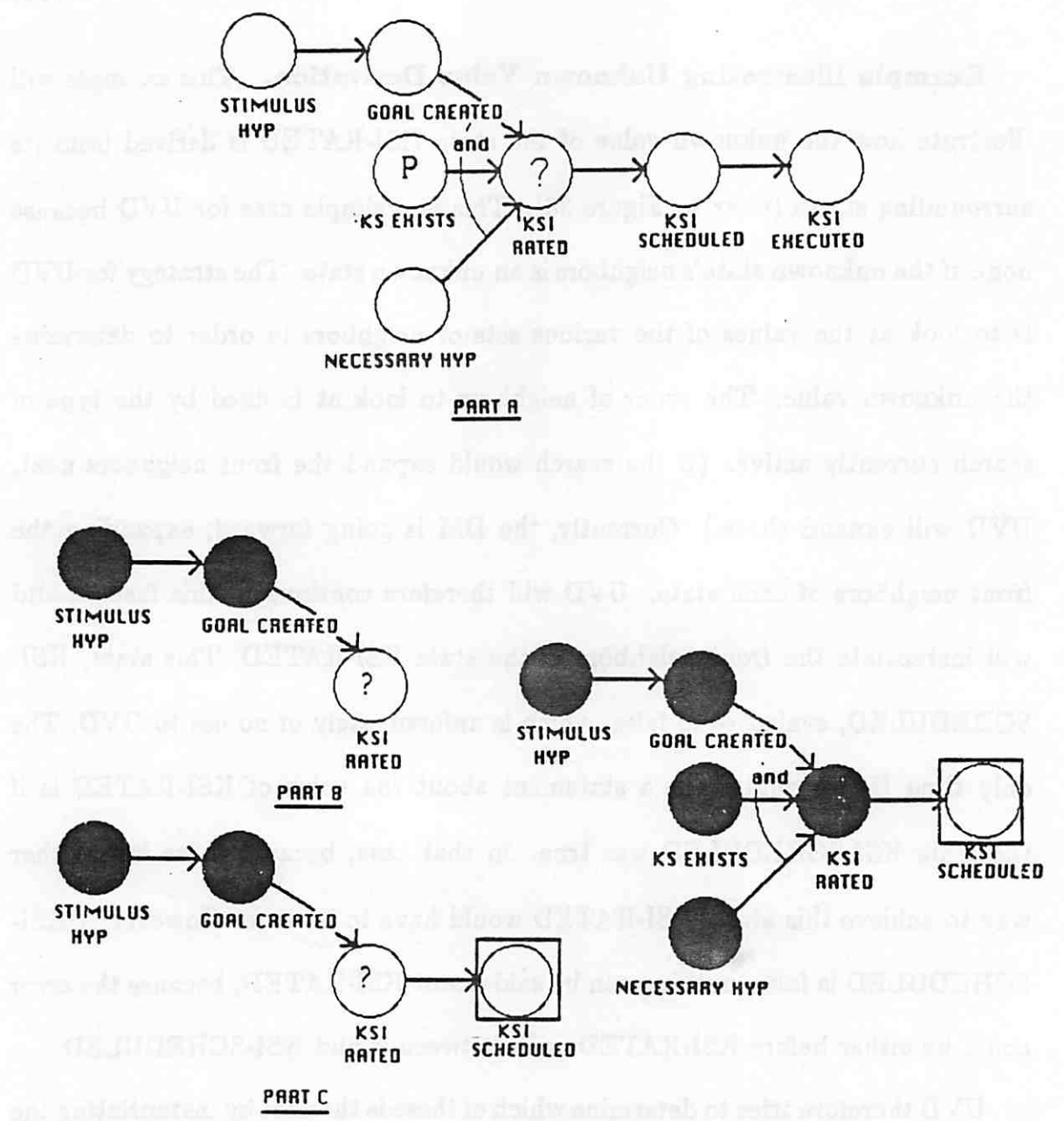


Figure 38: Example of Unknown Value Derivation

This figure illustrates how the neighboring states of the state KSI-RATED must be expanded in order to determine that state's value.

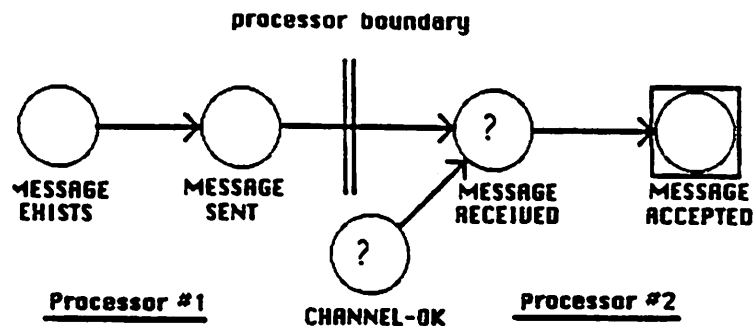


Figure 39: A Situation Where UVD Fails

This figure shows a case where UVD will not work because two unknown states are contiguous. Since there is not way to determine one without knowing the value of the other UVD will fail to evaluate the states MESSAGE-RECEIVED and CHANNEL-OK if the value of the state MESSAGE-ACCEPTED is false.

such that UVD is successful in determining the value of the state KSI-RATED.

There are cases however, where this is not possible. In those cases the unknown state value remains unknown.

It is important to realize, that UVD will not always work. In other words, there are cases where it cannot determine the necessary information from the surrounding states. Consider the situation shown in Figure 39. A UVD state (MESSAGE-RECEIVED) is preceded by two states; one of them is also a UVD state

(CHANNEL-OK). This predecessor UVD state is also a leaf state; i.e., it has no predecessors. It also has no successors which can be determined directly from the DVMT. There are therefore two adjacent UVD states and each requires the knowledge of the other's value before its value can be determined. It is important to recognize the cases where UVD will not work and not waste time trying to

determine the unknown state values.

UVD is made more complex by the fact that it has to check for multiple paths of achieving some state. (This is another example of the converging OR problem encountered in FCT.) UVD greatly relies on the correctness and completeness of the SBM. For example, UVD infers that a state must have occurred if its predecessors are all true and if nothing else occurs in between the predecessor states and the current state. Of course, if the model is incorrect and in fact some events do occur in between the predecessor and the current states, then nothing can be concluded about the current state, since the predecessors could be true and the current state false due to some other event in between those two states.

Currently, unknown value is used only with predicate states; i.e., it is used to determine whether a state is true or false, or whether the situation it represents did or did not occur in the DVMT system. UVD could also be used with relationship states, to determine whether a state is too low or too high, but this has not been implemented. In our system UVD is used as an auxiliary reasoning to derive values not available in the normal manner, namely, directly from the DVMT data structures. The focus of diagnostic systems however is precisely on this type of reasoning to determine the problems. Davis calls this reasoning candidate generation [10] and uses it to identify malfunctioning components in digital circuits. Our "candidate generation" problem is largely solved, due to the fact that DVMT maintains all the necessary information in its data structures. It was this fact which allowed us to focus on the other reasoning types, such as FCT and CR.

§8. Inconsistency Resolving

Inconsistency resolving (IR) is necessary in cases where the other reasoning types lead to an ambiguous explanation of the initial symptom. In other words, the other reasoning types can only narrow down the possible causes to a number of candidate faults but cannot determine which of the candidates is actually responsible for the symptom. This can occur when the DM cannot determine the value of a state (i.e., the state value remains unknown), when an inconsistency is detected in the diagnosis (i.e., the analysis of a symptom does not reveal anything wrong in the primitive states), or when there are several possible failures and the DM must decide which is the one responsible for the symptom.

IR resolves inconsistencies by determining which of the possible candidate failures is more consistent with overall system behavior. It determines which of the candidate objects are faulty by determining how close to the ideal behavior that object is. For example, if the initial analysis determined that one of two sensors is bad but could not determine which one, then IR could be used to determine which of the two is closer to an ideal sensor and thereby determine which is more likely to be faulty.

In order to perform IR we need to construct a model of the ideal behavior of an object. We need to determine how an ideal sensor will behave and characterize this behavior in terms of measurable quantities which can be determined during diagnosis. For example, an ideal sensor should produce highly correlated data with consistent signal strength. An ideal channel should not distort messages

and should transmit them on time. An ideal node should produce correlated, highly rated hypotheses. The construction of these models is non-trivial. The problem we run into here are the same as those requiring evidential reasoning. In other words, it is difficult to determine what constitutes an ideally behaving sensor or a channel and it is also difficult to develop the metrics necessary for comparing an arbitrary object to the ideal one. An ideal hypothesis should be strongly supported and should in turn support other hypotheses.

Once the models are constructed, we must collect information from the system which will allow us to compare the real with the ideal behavior. This will involve collecting information over long periods of time and over large spatial areas. Statistical techniques will be necessary to determine how close a match is of the real and the ideal behaviors of an object.

The reason this type of reasoning is different from the previously mentioned ones is that it uses much more global knowledge; in other words a much higher model of the system's behavior than any of the other reasoning types. The reasoning types discussed so far used one or a small group of objects to determine how they were derived or where their derivation stopped. The faults are then deduced from this information. IR looks at the behavior of many objects (hypotheses, goals, or messages) in order to determine whether a sensor, a channel, or a node is behaving properly. IR is thus the first step in the use of higher level models of system behavior for diagnosis. Such higher level models should be able to deal with detecting inconsistencies in the SBM, in the system's domain knowledge, or in the detection component (detecting a normal situation as a symptom).

This reasoning, unlike all the other discussed in this chapter, has not yet been implemented but some of the ideas about it are discussed Chapter X.

§9. Use of Underconstrained Objects

Underconstrained abstracted objects (UAB) were defined in Chapter III as abstracted objects with one or more of their attribute values either undefined or partially defined, i.e. underconstrained. Unlike a fully constrained abstracted object, which defines a single object in the DVMT, an underconstrained abstracted object defines a class of objects in the DVMT. UAB are useful in two situations:

- When the DM needs to reason about a group of objects. In some situations an entire group of objects in the DVMT behaves similarly. In such cases it is more efficient to represent the entire group by one abstracted object and construct one set of causal pathways rather than to reason about each individual object separately.
- When DM does not have enough information about some DVMT object to fully constrain it and must therefore leave some of the object's attribute values undefined or partially defined.

Although these are two different situations, they are both solved by the underconstrained abstracted object. In the former case, the system is abstracting from a class of objects and representing some common class characteristics by the UAB. In the latter case, the system is representing a class due to missing information. Either case results in the need to reason about a class of objects which is then represented by the underconstrained abstracted object.

We have already seen how these objects are used in FCT reasoning to propagate forward the effects of a fault. This is an example of a situation where underconstrained objects are necessary both because we want to reason about a whole class of objects affected by the fault and because some information about the objects is missing.

The other use is in asking the system underconstrained questions when we do not know enough to ask a fully constrained question. The symptoms we have discussed so far referred to single instances of objects; in other words, they were fully constrained. Examples of fully constrained questions are:

- Why did KSI0004 (i.e., ksi that does exist in the DVMT system) not execute?
- Why did a merge ksi that would produce a vt hypothesis 1-8 using a vt hypothesis 1-5 not execute? (Here we have ksi whose exact attribute values we know but which does not exist in the DVMT system.)

In order to ask these questions, we must already know a lot about the system. Suppose we do not have as much information and still want to start asking some questions. We ask an **underconstrained question**. Such questions are represented by underconstrained objects. Examples of such questions are:

- Why did no merge ksis execute?
- Why did no ksi execute that would produce vt hypothesis 1-8?
- Why did no ksi execute that would utilize vt hypothesis 1-4 as a stimulus-hypothesis?

With the introduction of UAB we must deal with the issue of relationships among abstracted objects which represent the same actual object in the DVMT. An object in the DVMT system might be represented both by a fully constrained abstracted object and by a UAB that defines the class of objects in which this object belongs. Constructing the proper overlapping and inclusion links among the two types of abstracted objects is important both for proper merging of diagnostic paths and for appropriately accounting for pending symptoms explained by a fault.³

Underconstrained objects make it possible to reason about classes of objects in the DVMT system. This type of reasoning has many applications which we have not yet explored. Chapter X discusses possible uses for this type of reasoning.

§10. Checking Global Consistency via Path Values

The DVMT system is based on redundancy in both data and processing. This redundancy makes possible both the detection of errors and some of the reasoning necessary to diagnose them. The redundancy however also means that just because a failure has been identified, it does not necessarily explain the initial symptom. In order to fully explain a symptom, not just one, but all the possible pathways that led up to it must be diagnosed. The concept of **path value** (defined below) is used to integrate the results of local analysis into an overall

³Diagnostic paths need to be merged so that the same symptom is not diagnosed several times. If a class of object has already been analyzed and explained then there is no need to reason about individual members of that class.

statement about the system behavior. Another use for the path value is as a global consistency check on the overall analysis. The analysis may or may not find a set of primitive states that explains the initial situation. Because the path value represents the results of overall analysis, it can be used to check whether the locally observed behavior (represented by state value), is consistent with the overall analysis (represented by path value).

We have already discussed the state value, which represents whether a situation represented by the state occurred in the DVMT or not. The path value is also associated with each state and is related to the state value. The state value attribute represents the outcome of a particular situation. For predicate states, the state value is true if the situation has occurred, false if it has not. For relationship states, the state value is the type of relationship the state represents; $<$, $=$, or $>$. The state value is local; representing the outcome of an isolated situation. In contrast to this, the path value attribute captures a more global aspect of the system's behavior. It represents an explanation of the state in terms of its predecessor or successor states; that is in terms of events that occurred before or after the state. Specifically, it represents the result of the analysis along the state's path and thus represents prediction of the state's value based on the analysis of its surrounding states. When the analysis of the DVMT system behavior is correct, the state's value and path value will agree. In other words, the local observations will be consistent with the overall analysis. When there is a problem, either in the analysis or in the system model, the two values may not agree. The concept of path value occurs in other diagnostic systems.

In CASNET for example, it is used to control whether the state's value will be determined by a direct test of the patient or whether it will be assumed to be the same as the path value [39].

What does it mean for a state to be consistent with the overall system behavior? Since the states represent causally related events we expect the entire sequence of states to be consistent. In other words, if some situation is true, then we know all situations preceding it must have also been true. If a situation is false, then all situations following it must also be false. Thus if a fault has been found along some state's path, we expect that state to be false.⁴ For example, if sensor data is necessary in order for some hypothesis to exist, then if that sensor is bad, the data will not be sensed and neither will any of the hypotheses utilizing that data be created. If we trace the initial problem to the lack of low level hypotheses but then find no problem with either the sensor or the data in the environment, then there is an inconsistency. Either the model is wrong and does not capture everything that goes on in the DVMT system, or the initial symptom was inappropriate and the analysis was incorrect. There are two types of inconsistencies:

1. The state value is false, indicating that an error must have occurred somewhere before that state, but the state's path value is true, indicating that no such error has been found.

⁴This is actually more complex because of the AND/OR connections among the state transition arcs. The path value of a state is calculated by applying whatever logical operators there are in the neighbor list expression to the path values of those neighbors. For example, the path value of a state with three ORed predecessor states will be false if all three of the predecessor state path values are false. This is discussed in detail in Chapter VIII.

2. The state value is true, indicating that the situation did occur as expected, but the path-value of the state is false, indicating that an error has been found and therefore the state should be false.

No Justification for a False State. What does it mean when a state value is false, indicating a fault somewhere before that state, and the path value is true, indicating that no fault has been identified? It can mean two things:

1. Either there is no fault on the path and the system will eventually get to the desired state. The problem is therefore timing. Perhaps the system is too slow, perhaps the diagnosis happened too early, or perhaps the symptom is just inappropriate and does not represent a real problem. Examples are: ksi rating is relatively low so it will take some time before it executes and produces the expected hypothesis. This may or may not be a problem. Unfortunately, we can not decide whether it is or not without the use of higher level models of the system which would capture more global behavior.
2. Or no fault has been found because the part of the system where the fault occurred has not been modeled. I.e., we had had a true-false state pair with no lower level model expanding what happens in between the two states.

Unfortunately, there is no way to distinguish among these two cases. The only thing we can do is to wait long enough to eliminate the possibility that the problem is just timing. Of course, the question here is "How long is long enough?". This is an example of a situation where the Diagnosis Module functions more as an assistant to the system developer and points out suspect areas of the system behavior, rather than providing solutions. It is up to the system developer to determine whether there really is a problem in the system, and whether therefore

the model was incomplete, or whether it is just a matter of time before the desired situation is reached.

In addition to using the path value attribute for checking global consistency among the instantiated states, it is also used during the **merging of causal pathways during diagnosis**. Merging of paths is necessary in order to eliminate duplication of diagnosis. This is needed when multiple symptoms lead to the same set of causal pathways. If a state is reached whose path value is already set, then that state has already been investigated does not need to be analyzed again.

§11. Summary

This chapter described the different types of reasoning used by the Diagnosis Module to analyze the behavior of the DVMT system. The reasoning types form a hierarchy; beginning with the underlying constraint propagation mechanism which can be used to do either forward or backward chaining, through the more complex Backward Causal Tracing and Forward Causal Tracing and finally CR. Underlying all the different types of reasoning is the propagation of the initial symptom values via the constraint expressions linking the abstracted objects. All the reasoning types work by constructing a pathway through the instantiated SBM which links causally related states. The causes of the initial symptom (or the effects of the initial situation) can be explained (or predicted) by scanning this causal pathway back to the primitive causes (or forward to the highest level states represented in the model). The use of Underconstrained Objects was explained

and motivated by the need to handle cases where reasoning about a class of objects is necessary; either for efficiency or because the information available can only define a class and not each individual member. The use of path values to check the overall consistency of the analysis was explained. Simple examples of the different reasoning types were given throughout the chapter. Chapter V illustrates how these reasoning types is used to analyze real problems in the DVMT system.

Chapter v

EXAMPLES OF MODEL USE FOR DIAGNOSIS

I don't want to smother you in theory.

- David Mamet, in *The American Buffalo*

This chapter discusses in detail two experiments illustrating how the Diagnosis Module (DM) identifies the faults in the DVMT system responsible for the system's inability to derive the correct result. The last section of this chapter discusses some of the methods used to make the diagnosis more efficient.

§1. Introduction

In Chapter II we described two examples of problems in the DVMT system and gave a high-level description of the reasoning necessary to diagnose the failures which caused them. Having described the structure of the System Behavior Model (SBM) (in Chapter III) and the different types of mechanisms used to reason about the system behavior (in Chapter IV), we are now ready to discuss the details of how the DM reasons about the DVMT system behavior. This chapter describes in detail the diagnosis necessary to identify the failures in the two problems

discussed in Chapter II. Both examples are actual runs of the DVMT system where the system was not able to derive the final answer within some specified time limit. The failures responsible for this were faulty parameter settings (such as a missing communication knowledge source or inappropriate interest areas), hardware failures (failed sensors), and problematic data (signals which were rated too low). Figure 40 lists the abbreviations which will be used throughout this chapter.

In both examples presented here, it was the lack of the correct final answer that was used as the symptom to initiate the diagnosis.¹ This hypothesis spans locations in times 1 through 8. The diagnosis thus begins with the state PT and its associated abstracted object PT-HYP-OB, which represents the correct hypothesis. Since this hypothesis does not exist in the DVMT system, this initial state is false. The DM works backward from the PT state as it searches for the primitive failures in the DVMT system responsible for the lack of the correct hypothesis. During this process, it may invoke other types of reasoning, such as the Comparative Reasoning or Forward Causal Tracing, to perform more sophisticated analysis of the system behavior. The examples will illustrate the following types of reasoning:

BACKWARD CAUSAL TRACING (BCT) will be used to find a missing knowledge source (ks), non-expandable states, missing SENSED-VALUE-OB's, and messages not sent or received.

FORWARD CAUSAL TRACING (FCT) will be used to account for all

¹In a completely autonomous system, the symptom would arrive from the detection component. Since we are only dealing with diagnosis here, the initial symptoms are set by hand.

- References to objects in the DVMT will be in lower case (ks, ksi); references to structures in the model will be capitalized (KS-OB, KSI-OB).
- References to the level of the data will be in lower case letters.

pattern track	pt	pattern location	pl
vehicle track	vt	vehicle location	vl
group track	gt	group location	gl
signal track	st	signal location	sl

- In the Answer Derivation Cluster there are abstracted objects representing hypotheses at various levels of the data blackboard. There are also the corresponding states. These will be referred to as follows:

PT and PT-HYP-OB	hypotheses at pattern track level
VT and VT-HYP-OB	hypotheses at vehicle track level
GT and GT-HYP-OB	hypotheses at group track level
ST and ST-HYP-OB	hypotheses at signal track level
PL and PL-HYP-OB	hypotheses at pattern location level
VL and VL-HYP-OB	hypotheses at vehicle location level
GL and GL-HYP-OB	hypotheses at group location level
SL and SL-HYP-OB	hypotheses at signal location level

- In addition the following abbreviations will be used:

ks	knowledge source
cks	communication knowledge source
ksi	knowledge source instantiation

Figure 40: List of Abbreviations Used Throughout This Chapter

symptoms a fault has generated. Specifically, in Example II., it will illustrate how missing or low rated communication knowledge sources (cks's) can account for no messages being communicated among nodes.

UNKNOWN VALUE DERIVATION (UVD) will be used to determine values of states whose values cannot be determined directly from the DVMT system data structures. This will be illustrated by determining the values of the following states: MESSAGE-RECEIVED, KSI-RATED, COMM-KSI-RATED, SENSOR-OK, and DATA-EXISTS.

COMPARATIVE REASONING (CR) will be used to determine why the ratings of two objects are different. This will be illustrated by showing how the DM analyzes why the rating of one knowledge source instantiation was lower than the rating of another one.

Figure 25 in Chapter III illustrates the System Behavior Model clusters used during diagnosis. There are 5 model clusters:

ANSWER DERIVATION CLUSTER: represents the levels of abstraction that the data are expected to go through in order to derive the final pt hypothesis.

KSI SCHEDULING AND EXECUTION CLUSTER: represents the expected sequence of events during ksi rating, scheduling, and execution.

COMMUNICATION CLUSTER: represents the expected sequence of events during communication via message passing among two processing nodes in the DVMT.

COMM KSI SCHEDULING AND EXECUTION CLUSTER: represents the events during the scheduling and execution of communication knowledge source instantiations.

HYP RATING DERIVATION CLUSTER: represents the dependencies among the various components of a hypothesis rating.

The examples will be presented as follows. First, the general scenario will be described, including the input data, the node and sensor placement, the local interest areas, and the communication areas among nodes. Next, the set of failures will be described and the model clusters used during diagnosis will be listed. This will be followed by a walk through the trace generated by the DM as it analyzes the problem and identifies the failures. Due to the repetitive nature of the diagnosis, not all the details will be presented, but only those that illustrate different types of reasoning. The annotated traces for both examples are included in Appendix D.

We will make the following assumptions for the examples.

1. We assume that a probe exists that can determine whether a channel is functioning correctly. We do this because the state CHANNEL-OK is an example of a state where Unknown Value Derivation does not always work. In order to determine its value, we would have to use the Inconsistency Resolving mechanism, which has not yet been implemented.
2. In order to make the example presentation clearer, we have excluded the investigation of some of the pathways that are not consistent with the interest area parameters. This means that communication will only be considered at the vehicle track (vt) level and that only the following answer derivation pathway will be considered: sl → gl → vl → vt → pt. The effects of this are purely cosmetic; the results of the diagnosis are not affected by having all the possible pathways instantiated.

§2. EXAMPLE I: Missing Communication Knowledge Sources

The scenario for this example is a four node system. Four sensors are distributed over the environment such that each node receives data from the sensor in its quadrant. The data are the signals generated by a moving vehicle. There are 8 time frames containing the signals. The signals will therefore be referred to as signals 1 through 8. This scenario is shown in Figure 14 in Chapter II.

The goal of the system is to integrate these signals into a pattern track describing the motion of the vehicle. Since no node receives all the data necessary to construct the final answer hypothesis, communication among nodes is necessary. Node #1 receives no signals locally. Node #2 receives signals 5 through 8. Node #3 receives signals 1 through 4. Node #4 receives signals 3 and 6. The local interest areas and the knowledge sources allow the nodes to process its local data up to the vehicle track level, by following the $sl \rightarrow gl \rightarrow vl \rightarrow vt$ derivation path. The locations are integrated into tracks at the vt level. Since Node #1 has no local data, its local interest area parameters have been set so that it does not work at levels sl, gl, or vl. (This was done only to simplify the example presentation.) The communication interest area parameters allow the nodes to communicate at the vt level. Partial results are thus transmitted in the form of vehicle track hypotheses. Each node works only in its own area up to the vt level but can work in the entire system area at the vt and pt levels, provided it has received the necessary data from the other nodes. Any node could therefore in

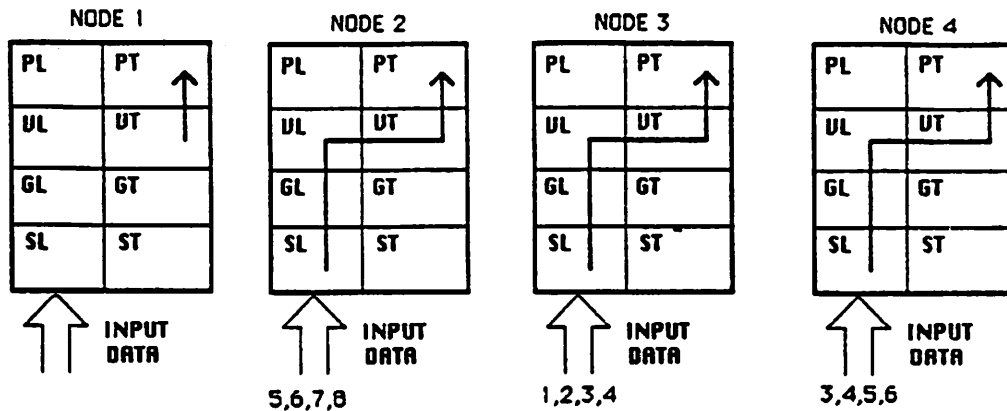


Figure 41: Interest Areas for Example I

This figure illustrates the answer derivation pathways allowed by the interest areas in the four nodes. In addition to the pathways shown, there are communication links among the four nodes at the vehicle track level.

principle derive the final answer. The interest areas are shown in Figure 41.

The symptom that initiates the diagnosis is the lack of a spanning pt hypothesis at any of the nodes, i.e., a pattern track hypothesis integrating all eight sensed locations. In order to derive it, the nodes must communicate. However, due to faulty parameter setting, the cks's responsible for sending the hypotheses among the nodes, are missing. The failures causing this problem is the lack of cks's at the vehicle track level. There are two cks's which can send vehicle track (vt) hypotheses among the nodes, hyp-send:vt and hyp-reply:vt, and thus there are two faults responsible for the lack of communication among the nodes. These faults are diagnosed by the DM using BCT. In the process, UVD is used to determine the values of all MESSAGE-RECEIVED states, since their values cannot be determined directly by looking at the DVMT data structures. Once these two faults are identified, FCT is employed to simulate their effects forward

and thereby account for any pending or future symptoms.

The diagnosis is presented as follows. First, the DM diagnoses each of the four nodes; starting each time with the missing pt spanning hypothesis. This diagnosis identifies the vehicle track segments missing due to the faulty communication. BCT is used to perform this initial diagnosis which ends when further diagnosis would require transition to another node. This occurs when the MESSAGE-SENT states are reached (see the diagram of the Communication Cluster in Figure 25 in Chapter III). These states are saved as pending symptoms and await further diagnosis. Since every missing track segment eventually leads to a MESSAGE SENT state, there will be many MESSAGE-SENT states accumulated, awaiting further diagnosis. After all four nodes are analyzed, a representative MESSAGE-SENT pending symptom will be selected from those awaiting diagnosis at Node #2. It is the diagnosis of this symptom that leads to the identification of the failures: the missing hyp-send:vt and hyp-reply:vt cks's. Once these failures are identified, FCT is used to propagate their effects forward. This requires reasoning about a class of objects (the class of all messages at the vt level) and the use of underconstrained objects is thus necessary. The diagnosis ends when FCT successfully accounts for all the pending MESSAGE-SENT symptoms at Node #2. (The diagnosis of the pending symptoms at the other three nodes is analogous to the diagnosis at Node #2 and will not be discussed.) The system is constructed such that the order of diagnosis (i.e., the order in which the nodes and symptoms are diagnosed) is irrelevant; the results will be the same regardless of the ordering.

§2.1 Diagnosis for Node #1

Node #1, in the upper left corner of the environment, receives no input data. Because it does not have any locally sensed data, this node relies on communication at the vt level to receive all its data, in the form of partial results generated by the other nodes in the system. The interest areas are therefore set such that this node can only begin working on data at the vt level. Due to the missing cks's however, this node never receives any hypotheses and does not therefore produce any results. Figure 42 shows a graphical representation of the instantiated SBM representing the diagnosis at Node #1. (The abstracted objects are not shown in the figure in order to make it less cluttered.) Diagnosis begins with the false state PT79 and its associated object PT-HYP-OB, representing the desired answer pattern track integrating locations 1 through 8 (pt 1-8). This state is false and the DM initiates diagnosis with BCT by expanding the back neighbors of this state. The back neighbors of PT are PT (since all track states are reflexive), PL, and VT in the Answer Derivation Cluster and MESSAGE-ACCEPTED in the Communication Cluster. PT and PL are related by the PrefOR operator which means that BCT first tries to instantiate another PT state, representing shorter pattern track segments, and only if no PT states can be instantiated does BCT try to instantiate the PL states. In this case, no shorter pt segments are found so no PT states are instantiated. Normally, the PL states would be tried next. Recall however, that pl is not within the interest area and that we are not expanding states outside the interest area. PL is therefore not expanded. This

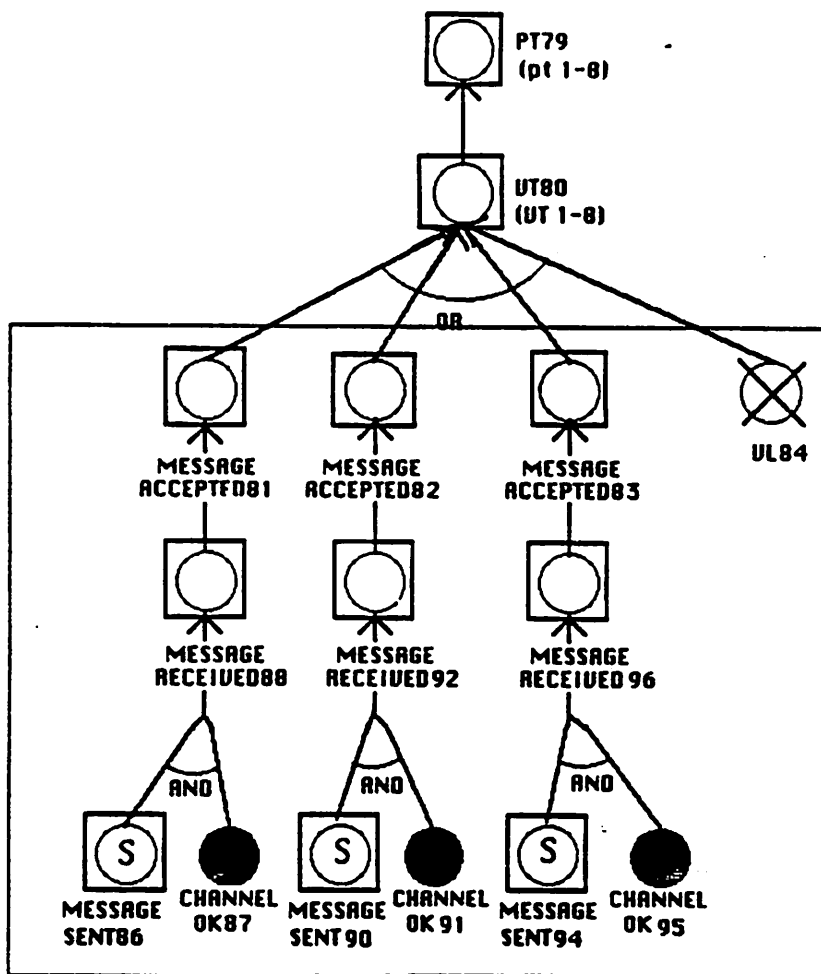


Figure 42: Instantiated SBM for Diagnosis in Node #1
 Instantiated SBM showing the result of BCT in diagnosing the lack of a pt hypothesis 1-8. This symptom is traced back to the lack of the necessary locations locally (non-expandable state VL84) and to the lack of received messages from other nodes (false states MESSAGE-SENT86, 90, and 94).

is also the case for the MESSAGE-ACCEPTED state, since the communication interest areas allow communication only at the vt level. The remaining possibility therefore is the state VT. The object which would normally lead to the derivation of the pt hypothesis 1-8 is a vt hypothesis 1-8. The DM therefore instantiates a VT-HYP-OB to represent this object and creates its corresponding state, VT80. Since a vehicle track 1-8 does not exist in the DVMT system, the state VT80 is false. This is the only back neighbor of the initial symptom state.

BCT continues diagnosis by expanding the back neighbors of false states until a true state is reached. Since VT80 is false, its back neighbors are expanded next. The back neighbors of a VT state are VT, VL, and GT, in the Answer Derivation Cluster, and MESSAGE-ACCEPTED state, in the Communication Cluster. Again, the VT and VL states are related by the PrefOR connective, indicating preferred order of instantiation. Here the choices are shorter vt segments (VT), vehicle locations (VL), and, since vt is a communication level, messages received and accepted from other nodes (MESSAGE-ACCEPTED) states from the other three nodes in the system. (The gt level is not in the interest area and is no GT states are therefore instantiated.) Since no shorter vt segments exist, no VT back neighbors are instantiated. The VL state is tried next. Here, the 8 vl hypotheses objects necessary to derive the desired vt 1-8 hypothesis are instantiated by creating their corresponding abstracted objects. All these objects are out of the node's interest area however and are therefore all grouped under one non-expandable VL state, VL84. This terminates the diagnosis for this pathway. A fault has been found, the non-expandable VL state, and therefore an F path value, indicating

an identified fault, is propagated back up through the model. When the path values of the other back neighbors of the VT state become known, they will be combined using the OR logical operator which links the back neighbors of the VT state, to determine the overall VT path value. This value will then be used to determine the path value of the initial symptom state.

Since the communication among the nodes is supposed to take place at the vt level, the MESSAGE-ACCEPTED back neighbor of the VT state is expanded. There are three other nodes in the system and therefore three separate MESSAGE-OBJECTS will be created; an object representing a vt hypothesis 1-8 expected from each of the three nodes. These objects are not grouped together under one MESSAGE-ACCEPTED state, because their diagnosis eventually involves symptoms at three different nodes. Three separate MESSAGE-ACCEPTED states are therefore created, one for each MESSAGE-OBJECT. These are the states MESSAGE-ACCEPTED81, MESSAGE-ACCEPTED82, and MESSAGE-ACCEPTED83. These states are determined to be false since no vt hypothesis 1-8 exists in Node #1 that was received from either of the three other nodes. Diagnosis continues with the MESSAGE-ACCEPTED81 state which is associated with a MESSAGE-OBJECT representing the vt 1-8 hypothesis expected from Node #2.

Since this state is false, its back neighbor, the state MESSAGE-RECEIVED, is expanded, resulting in the creation of the state MESSAGE-RECEIVED88. No new object is created because both of these states refer to the same object: the MESSAGE-OBJECT representing the vt hypothesis 1-8. The evaluation of the

state MESSAGE-RECEIVED88 presents a special problem however, because the DVMT system does not keep track of messages that were transmitted across the communication channels but were not accepted by the receiving nodes. In other words, the value of this state cannot be determined directly by examining the DVMT data structures, as was the case with the states instantiated to this point. Instead, UVD must be used.

UVD works by examining (and, if necessary, expanding) the states surrounding the unknown value state and tries to determine that value from the values of these neighboring states. In this case, UVD continues to expand the back neighbors of the state MESSAGE-RECEIVED88. This results in the instantiation of the states MESSAGE-SENT86 and CHANNEL-OK87 (along with their associated objects). These states represent the sending of the message from another node and the state of the communication channel linking the two nodes. CHANNEL-OK87 is determined to be true by using a simulated probe; MESSAGE-SENT86 is determined to be false by examining the executed knowledge sources at Node #2 and seeing that no cks's executed. Since both MESSAGE-SENT and CHANNEL-OK must be true in order for the MESSAGE-RECEIVED to be true, UVD concludes that MESSAGE-RECEIVED88 must be false. The diagnosis for this path ends, because the state MESSAGE-SENT86 is saved for later diagnosis by Node #2. Since this state is false, an F path value is propagated back through the pathway.

The diagnosis of the two remaining states, MESSAGE-ACCEPTED82 and MESSAGE-ACCEPTED83, proceeds as above, with the values of the states

MESSAGE-RECEIVED92 and MESSAGE-RECEIVED96 determined by UVD to be false. Two additional symptoms, MESSAGE-SENT90 and MESSAGE-SENT94, are produced, which await further diagnosis at nodes #3 and #4 respectively. F path values are propagated back up through the model.

At state VT80, these path values are combined with the path value generated from the non-expandable state VL84. VT80 could have been achieved via either of the MESSAGE-ACCEPTED states or via the VL state. All MESSAGE-ACCEPTED states end with a false MESSAGE-SENT state and therefore generate F path values. Recall that the path value for the state VL84 is also false, because that state is non-expandable. Combining these path values produces (OR F F F F) which results in an F path value for the state VT80. This indicates that faults have in fact been found preceding this state and the state's locally determined false value is therefore consistent with the F path value. The path value of the initial symptom state PT79 is simply the value of the state VT80, since VT80 is the only back neighbor of PT79. The calculation of the path value of PT79 concludes the diagnosis for this node. The diagnosis was successful because a set of faults was found which explained the initial symptom. The faults were the non-expandable state VL84 and the three pending symptom states MESSAGE-SENT86, 90, and 94. These symptoms will be diagnosed later.

§2.2 Diagnosis for Node #2

Node #2 is in the upper right corner of the environment and receives signals in locations 5 through 8 as its input data. It generates a pattern track integrating

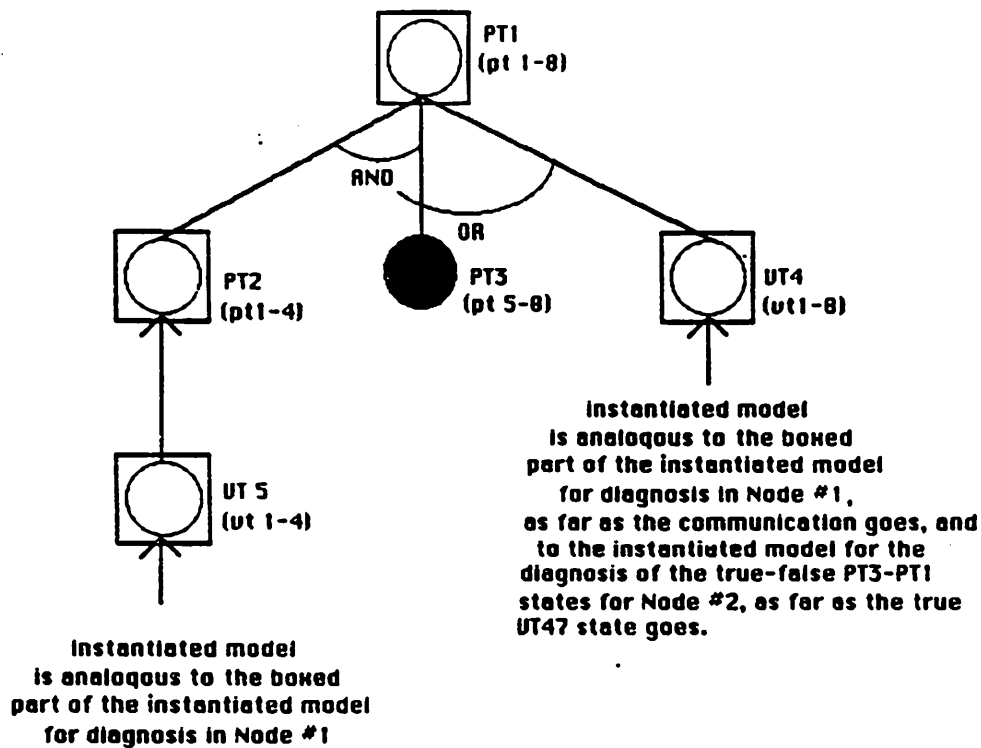


Figure 43: Instantiated SBM for Diagnosis in Node #2-Part I
 First part of diagnosis in Node #2 showing the instantiated SBM that shows how the lack of the pt 1-8 hypothesis can be explained in terms of the lack of the vt 1-8 hypothesis (false state VT4) and the lack of a vt segment 1-4 (false state VT5).

these signals but, due to the lack of communication, does not receive the vt hypotheses that would allow it to extend its 5-8 track to the full spanning pt 1-8 hypothesis and thus derive the final pattern track answer. Figure 43 shows the first part of the instantiated SBM for this node.

As in Node #1, diagnosis begins with the lack of the spanning hypothesis pt 1-8. This is represented by an abstracted object PT-HYP-OB1 (not shown in figure), attached to the state PT1. As before, since PT1 is false, its back neighbors are expanded. This is analogous to the first step of diagnosis in Node #1, but this

time there are shorter pt tracks, so some PT states are created. Specifically, one true and one false state, with the attached objects representing the existing pt segment 5-8 (state PT3) and the missing pt segment 1-4 (state PT2). As before, a VT state is instantiated with its associated abstracted object representing the desired vt 1-8 hypothesis. Diagnosis continues in a depth-first manner, with the expansion of the back neighbors of the false PT2 state. This time however, there are no shorter existing pattern track segments and therefore only the VT neighbor is expanded, resulting in the creation of state VT5. This state represents the missing vehicle track segment 1-4. Since communication is to take place at this level, the back neighbors of this state include MESSAGE-ACCEPTED, in addition to the VT and VL. As before in Node #1, the vt hypothesis could be received from any of the three nodes in the system, so three separate MESSAGE-ACCEPTED states are instantiated, one for each node which was expected to send a message to Node #2. Their associated objects represent the vt hypothesis 1-4. The instantiated model is analogous to the boxed part of Figure 42, except of course that the numbers of the states are different. These states are false so their back neighbors are expanded, resulting in the creation of three MESSAGE-RECEIVED states, MESSAGE-RECEIVED10, MESSAGE-RECEIVED11, and MESSAGE-RECEIVED12. As before, Unknown Value Derivation must be used to determine the values of these states. They are determined to be false, since no cks's can be found at the sending nodes which sent the desired message. As before, three pending MESSAGE-SENT symptoms are produced, one for each of the three nodes that was expected to send the missing track segment to Node

#2. These states are of course false and are stored for further diagnosis.

No VT states are expanded for the VT5 state since no shorter tracks exist in Node #2. For the VL back neighbor, four VL-HYP-Objects are created, one for each of the 4 locations necessary to form the vt track 1-4. All four fall outside of the node's interest area and are therefore marked as non-expandable. Only one state is therefore instantiated for all four objects, the VL9 state (not shown in Figure 43 because it is analogous to the VL84 state in Figure 42). This terminates diagnosis for the left-most branch of the tree. An F path value is propagated back up through the tree and will be combined later with the path values from the PT3 and VT4 states to generate the final path value for the initial symptom PT1.

Diagnosis continues with the true state PT3, and its associated abstracted object representing the existing portion of the track pt 5-8. Since the associated object does exist in the DVMT, this state is true. When a true state is encountered during BCT, the DM must look for the problem in between the true and the false states, at a lower cluster of the model, which represents the states occurring in between the true state and the false state. In this case the DM must find out why the pt 5-8 was not extended to the pt 1-8. The DM expands the lower cluster, the Ksi Scheduling Cluster, and finds that the reason the pt 5-8 was not extended, is due to the missing pt 1-4. The diagnostic pathways merge at this point at the state PT2, which represents the missing pt 1-4 segment. The instantiated model for this part of the diagnosis is in Figure 44. Note that two KSI-RATED states are instantiated since there are two types of knowledge sources that could use the existing 5-8 vt and derive the desired 1-8 vt; the concatenate-tracks ks and

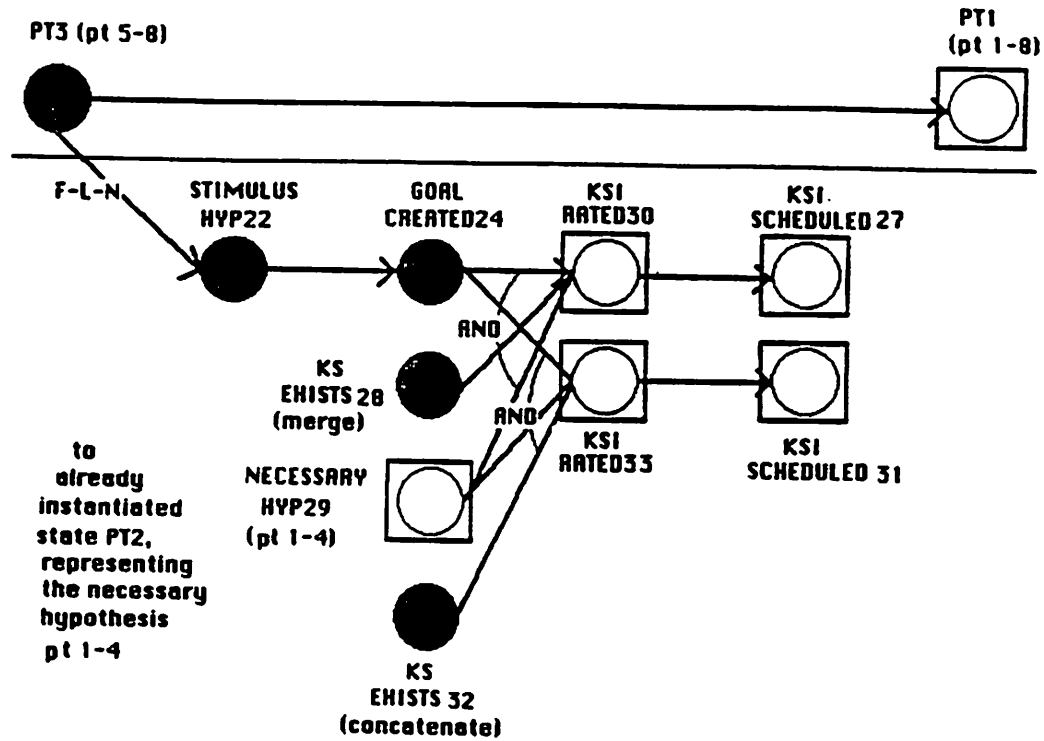


Figure 44: Instantiated SBM for Diagnosis in Node #2-Part II
 This figure shows how the true-false pair PT3-PT1 is diagnosed by BCT. The lack of the pt 1-8 hypothesis is traced to the lack of the segment pt 1-4 (false state NECESSARY-HYP29 which merges with existing false state VT5).

the merge-tracks ks, represented by the states **KS-EXISTS32** and **KS-EXISTS28**, respectively.

During the instantiation of this portion of the model, the Unknown Value Derivation must be used again to determine the value of the state **KSI-RATED**, since this information is not kept by the DVMT system. Here UVD first expands the forward neighbors of states **KSI-RATED30** and **KSI-RATED33**, the states **KSI-SCHEDULED27** and **KSI-SCHEDULED31** respectively. However, since both the **KSI-SCHEDULED** states are determined to be false, this information provides no help in determining the value of the **KSI-RATED** states. All it tells us is that the system never got as far as scheduling the desired knowledge sources; it might have rated them or not. In general, a false state provides no information about its preceding states since they could be false (due to a failure further back in the processing), or they could be true and the error could have occurred in between the true predecessors and the false state. In order to determine this, the DM must expand the back neighbors of the **KSI-RATED** states. If all of those states are true, and if no other events occur in between them and the **KSI-RATED** states, then the **KSI-RATED** states can be assumed to be true. If any of the back neighbors are false, then the **KSI-RATED** states must be false (since the back neighbors are linked by an AND connective). The back neighbors to expand here are: **GOAL-CREATED**, **KS-EXISTS**, and **NECESSARY-HYP**. The **GOAL-CREATED** state has already been expanded and determined to be true. (There is only one **GOAL-CREATED** state for both **KSI-RATED** states, since the same goal stimulated both ksi's, this is the state **GOAL-CREATED24**.)

Two **KS-EXISTS** states are created, one for each type of applicable **ks**, and both are determined to be true; i.e., the **ks**'s exist in the node. One **NECESSARY-HYP** state is created, **NECESSARY-HYP29**, since the same hypothesis (pt 1-4) is necessary for both knowledge sources. This state is false, since the pt 1-4 does not exist. Because all three of the back neighbors of **KSI-RATED** must be true before a **ksi** can be rated, the **KSI-RATED** states must be false, since the **NECESSARY-HYP29** state is false. This concludes the Unknown Value Derivation for the **KSI-RATED** states.

The **DM** picks up the diagnosis where it left off before **UVD** had to be invoked to determine the value of the **KSI-RATED** states. The **GOAL-CREATED24** state is true, so no fault lies in that direction. A **T** path value is propagated back through the model, and will be combined with the path values for the **KS-EXISTS** states and the **NECESSARY-HYP** states. The **KS-EXISTS** states terminate the diagnosis, because they are primitive states. Since they are true, no failure has been identified and a **T** path value is propagated back through the model. The remaining state is **NECESSARY-HYP29** which is false. The **DM** therefore continues by expanding its back neighbors in order to determine why the pt 1-4 this state represents is missing. The state **NECESSARY-HYP** however has no back neighbor, since it is the first state of the **Ksi Scheduling Cluster**. In such cases, the **DM** expands the upper back neighbors instead, which lead to a model cluster at the next higher level of the model hierarchy. In this case it is the **Answer Derivation Cluster** and the paths merge at an already existing **PT2** state, which represents the missing pt 1-4 track segment. Since this state has

already been diagnosed and explained (in terms of the lack of local data and messages not sent from other nodes) diagnosis ends here with the merging of diagnostic paths at the state PT2. Since the path value of PT2 is false, this value is propagated back through the model. The three path values necessary to calculate the path values for the KSI-RATED states are thus: T (for the states KS-EXISTS28 and KS-EXISTS32), T (for the state GOAL-CREATED24), and F (for the state NECESSARY-HYP29). Combining these values by the AND connective gives an overall value of F for that pathway. This value is then propagated back up through the model to the state PT3. When the path values of all three states are known (PT2, PT3, and VT4), they will be combined to determine the overall path value for the initiating symptom state, PT1.

The final part of diagnosis for this node is the false state VT4, representing a 1-8 track at the vehicle level (see Figure 43). The back neighbors of a VT state are VT, VL, and MESSAGE-ACCEPTED states. Since a shorter vt segment exists (vt 5-8), two VT states² are created: VT47 representing the existing track 5-8, and a state representing the missing segment 1-4, which is merged with the already diagnosed state VT5. Since VT neighbors were instantiated, no VL states are expanded (recall that VT and VL are related by the PrefOR operator and only one is instantiated). Three MESSAGE-ACCEPTED states are instantiated, one for each of the three neighboring nodes. These states eventually lead to three additional MESSAGE-SENT symptoms that are saved at their respective nodes

²These two states are not shown in the figure because their diagnosis is analogous to the diagnosis in Node #2.

for later diagnosis.

What remains to be diagnosed then is the true→false pair VT47→VT4. This diagnosis is analogous to the diagnosis of the PT3→PT1 states discussed above and the instantiated model is analogous to the model describing that diagnosis in Figure 44. As in the diagnosis of the PT3→PT1 true→false pair, the aim is to discover why the existing vt 5-8 track was not extended to the full length 1-8 track. The lower Ksi Scheduling Cluster is instantiated and again, this diagnosis merges with the false missing vt 1-4 segment, represented by the already diagnosed state VT5. The result is an F path value, which is propagated back through the model. The three path values for the three back neighbors of the initial symptom (PT1) are: F (for PT2), F (for PT3), and F (for VT4). These are combined by the logical operators linking these states. The resulting expression is:

(OR (AND F F) F),

which is of course F. This successfully concludes the diagnosis for Node #2. The lack of a spanning pattern track hypothesis 1-8 was explained by the missing messages from the other nodes (all the false MESSAGE-SENT symptoms) and by the lack of the necessary data locally (the false VL9 state).

§2.3 Diagnosis for Node #3

Node #3 is in the lower left corner of the environment and receives data from locations 1 through 4 from its sensor. It therefore generates a pattern track integrating these locations but cannot produce the final answer because it

is missing the track integrating locations 5 through 8. The diagnosis for this node is similar to the diagnosis for Node #2. The same parts of the model are instantiated and same types of failures are identified. The difference of course being that here it is the track 5-8 that is missing, not 1-4 as was the case in Node #2. Again, MESSAGE-SENT symptoms are generated, which are saved for further diagnosis.

§2.4 Diagnosis for Node #4

Node #4 is in the lower right corner of the environment and receives data in locations 3 through 6. It integrates these locations into a pattern track linking times 3 through 6 but does not produce the overall answer because it never receives the track segments 1-2 and 7-8. Diagnosis proceeds as before, generating an explanation in terms of the missing local data in locations 1-2 and 7-8, and producing a number of false MESSAGE-SENT symptoms which are saved for later diagnosis. The diagnosis of Node #4 concludes the individual node diagnosis of the system. The final step will be the diagnosis of the pending MESSAGE-SENT symptoms at the four nodes.

§2.5 Diagnosis for the Pending Symptoms at Node #2

The pending symptoms are divided into four groups by the node that needs to do the diagnosis. Since the diagnosis of these symptoms is analogous for all four nodes, only the diagnosis of symptoms in Node #2 will be discussed here. The DM first identifies the reason for the failed communication, in terms of the missing

cks's, and then uses FCT to simulate the effects of this failure on the DVMT system behavior and thereby account for the remaining pending MESSAGE-SENT symptoms at the node. The instantiated model for this diagnosis is in Figure 45, part A.

There are six pending MESSAGE-SENT symptoms at Node #2, representing the various vehicle track segments that other nodes expected to receive from Node #2. Diagnosis begins with MESSAGE-SENT111, which represents the vt 5-8 hypothesis that was expected but never received by Node #3. Since this state is false, Backward Causal Tracing is invoked to instantiate the back neighbor of this state, this results in the creation of the state MESSAGE-EXISTS217. This state is true, since a vt 5-8 hypothesis does exist at Node #2. The problem must therefore occur in between these two states and the lower Communication Ksi Scheduling Cluster is therefore instantiated next. The front lower neighbor of the true state MESSAGE-EXISTS217 is instantiated, the state COMM-KSI-RATED221. As before, Unknown Value Derivation is used with this state to determine that it is false. Note that here, unlike in the KSI-RATED state in the Ksi Scheduling Cluster, the objects representing the two types of cks's are grouped under one state. They do not need to be treated separately because they represent simpler processing than those in the Ksi Scheduling Cluster; no GOAL-CREATED and NECESSARY-HYP states exist which necessitate separate treatment of the ks types.) The UVD involves the expansion of the back neighbor of the COMM-KSI-RATED221 state, the state COMM-KS-EXISTS220. This is a primitive state and represents the identified failure: the missing communication knowledge

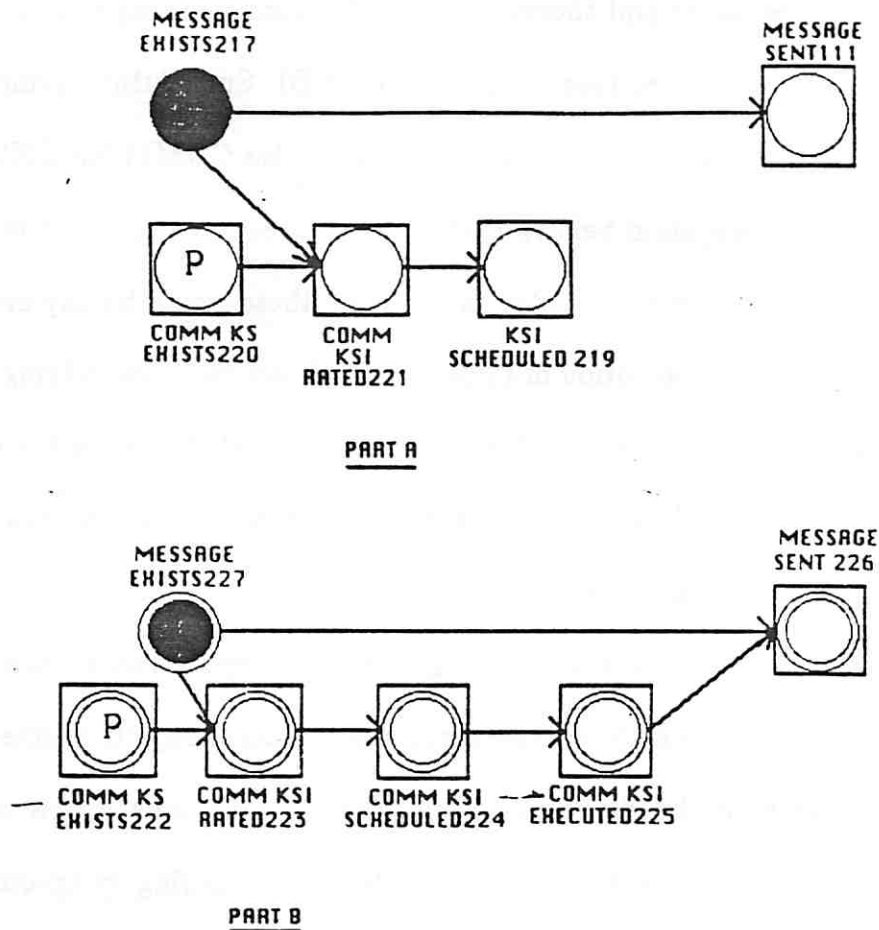


Figure 45: Instantiated SBM for Diagnosis of Pending Symptoms in Node #2

The upper part of the figure shows how BCT traces the cause of the false MESSAGE-SENT111 state to the lack of a communication ks (false primitive state COMM-KS-EXISTS220). The lower part of the figure shows how FCT propagates the effects of this fault forward and thereby accounts for all pending MESSAGE-SENT symptoms.

sources hyp-send:vt and hyp-reply:vt.

At this point the FCT is invoked to simulate the effects of the identified failures on system behavior and thereby account for both pending and future symptoms caused by this failure (see Figure 45, part B). Such future symptoms are any states that are in some way effected by the false COMM-KS-EXISTS220 state. In terms of the system behavior, these are any events in some way dependent on the existence of these two cks's. In this case, these would be any events regarding the scheduling or execution of these cks's and any messages relying on these cks's for transmission. In terms of the SBM, the affected states are any states forward of the KS-EXISTS state, where forward means both at the same level of the model hierarchy and at any higher levels.

Recall that FCT works by propagating forward an underconstrained abstracted object which represents the whole class of objects affected by the fault. At each step in the diagnostic cycle, that is each time a new object/state is created, any abstracted objects associated with pending symptoms are checked to see whether they overlap with the newly created underconstrained object. If so, then their pending symptom has been successfully explained by the identified fault and need no longer be diagnosed.

Getting back to the example, FCT first creates underconstrained objects representing the missing communication knowledge sources. (These are the objects COMM-KSI-OB129 and COMM-KSI-OB130, representing the hyp-send:vt and hyp-reply:vt cks's respectively. They are not shown in the figure but can be seen in the trace in Appendix D.) The corresponding state is then created, COMM-

KS-EXISTS222. (In the diagrams, the states attached to underconstrained objects are represented by two concentric circles, instead of the usual single circle.) FCT continues simulating the fault's effect on the DVMT by expanding the front neighbors of this state. This results in the creation of the state **COMM-KSI-RATED223.** This state is false and could not have been achieved via any other pathway. This means that any pending symptom states which overlap with this one, via their abstracted objects, can be explained by the simulated fault. In this case there aren't any overlapping **COMM-KSI-RATED** symptoms and FCT continues forward. The states **COMM-KSI-SCHEDULED224** and **COMM-KSI-EXECUTED225** are instantiated next. Both are false, since no **cksi**'s exist, either on the scheduling queue or already executed, for the two missing **cks**'s. Since the state **COMM-KSI-EXECUTED225** has no forward neighbor at the same level, its forward upper neighbor is expanded,. This is the **MESSAGE-SENT** state in the Communication cluster above. Here FCT creates another underconstrained object, a **MESSAGE-OBJECT** representing any hypothesis at the vehicle track level. The object's level attribute is set to **vt** and all other attributes are left empty. The resulting underconstrained object represents the class of objects affected by the missing **cks**'s, namely all message hypotheses at the **vt** level. This time the underconstrained **MESSAGE-OBJECT** does overlap with the **MESSAGE-OBJECTS** associated with the various pending **MESSAGE-SENT** symptoms. The FCT state **MESSAGE-SENT226** is created and BCT is used to ascertain that there is no other way to achieve this state. This does indeed turn out to be the case and as a result all **MESSAGE-OBJECTS** which overlap with the underconstrained

MESSAGE-OB can be accounted for by the identified faults, the missing cks's. This means that their associated MESSAGE-SENT states that have been pending diagnosis at Node #2 are now explained by the failure. This explanation of the pending symptoms terminates diagnosis at Node #2. The pending symptoms at the other three nodes are diagnosed analogously and will not be discussed in detail here. Diagnosis for Example I thus ends when all four nodes have been analyzed and all symptoms generated during this diagnosis have been traced back to the missing communication knowledge sources.

§3. EXAMPLE II: Low Sensor Weight and Malfunctioning Sensor

This example illustrates how the DM identifies multiple failures: a low sensor weight parameter and an inconsistency among sensor data which points to a malfunctioning sensor. In the process, use of CR is demonstrated. The example scenario is shown in Figure 15 in Chapter II. A single-node receives data from two sensors, A and B. Sensor A covers the entire area, sensor B covers the right half of the area. The right half of the environment is therefore sensed redundantly by both sensors. The sensor weight parameters, which control how the sensors weigh their input data, are set up so that the area sensed exclusively by Sensor A is weighed strongly, as is the area sensed by Sensor B. The sensor weight parameter for the right half of the area for Sensor A is set low however. This is intended to simulate a situation where Sensor A's accuracy decreases in some area and that area is therefore sensed redundantly by an auxiliary sensor, in this case Sensor

B. The result of these parameter values is that the data is sensed strongly by Sensor A in the left half of the environment, and weakly in the right half. Sensor B produces highly rated signals in its half of the environment, due to its high sensor weight parameter.

If Sensor B was functioning correctly, this would present no problem. However, in this scenario. Sensor B is malfunctioning and is producing random noise data. Because of the high sensor weight parameter, this data is rated very highly, thus overshadowing the correct data as sensed by Sensor A in that area. This results in the correct data in the right half of the environment, as sensed by Sensor A, being weighted low, which leads to low rated hypotheses and ksis at all levels of the data blackboard. As a result, the correct data is not worked on and instead the highly rated noise segments are processed by the node.

The specific data in this case are 8 signals generated by a vehicle moving from left to right through the environment. All 8 locations are sensed by sensor A but only the first 4 are sensed strongly. The last 4 fall within the area affected by the low sensor weight parameter and are therefore rated low. As a result, the knowledge sources scheduled to work on this data are also rated low. The DVMT system first derives a pattern track integrating the first 4 highly rated locations of the correct track (pt 1-4). It then spends its time working on the noise produced by Sensor B and never extends the pattern track 1-4 to include the remaining locations.

Diagnosis proceeds as follows. The initial symptom, as before, is the lack of a spanning hypothesis, pattern track hypothesis integrating locations 1 through

8. BCT is invoked and finds that the first part of the data is integrated into a pt track 1-4, but that this track is never extended. BCT therefore continues backward through the Answer Derivation Cluster and discovers that the necessary data, locations 5 through 8, never reached the vl level. The next step is to find what prevented the existing gl hypotheses from being extended up to the higher levels. BCT finds that this was due to the low rating of the knowledge source instantiations that were to work on this data. At this point Comparative Reasoning (CR) is invoked to determine the reasons for the low rating of the ksis. This leads to the discovery that the sensor weight parameter for Sensor A is very low and also that there is an inconsistency among the two sensors, as far as the data in the right half of the environment goes.

The diagnosis begins with the missing pt 1-8 track, represented by the symptom state PT1 and its associated object PT-HYP-OB1. Refer to Figure 46; the abstracted objects are not shown in the figures. This state is false and BCT therefore continues to expand the state's back neighbors. These are shorter pattern tracks (PT) and a vehicle track (VT) of the same length. A shorter pattern track does exist, the segment pt 1-4. Expansion of the PT back neighbor therefore results in two states and their objects: one representing the existing track segment, PT-HYP-OB2 (pt 1-4), and one representing the missing track segment, PT-HYP-OB3 (pt 5-8). These are attached to states PT3 and PT2 respectively. Since the pt 1-4 exists, PT3 is true; since pt 5-8 does not exist, PT2 is false. The VT neighbor expands into a single state, VT4, and its associated object, VT-HYP-OB4, representing the full track at the vehicle track level. Since no

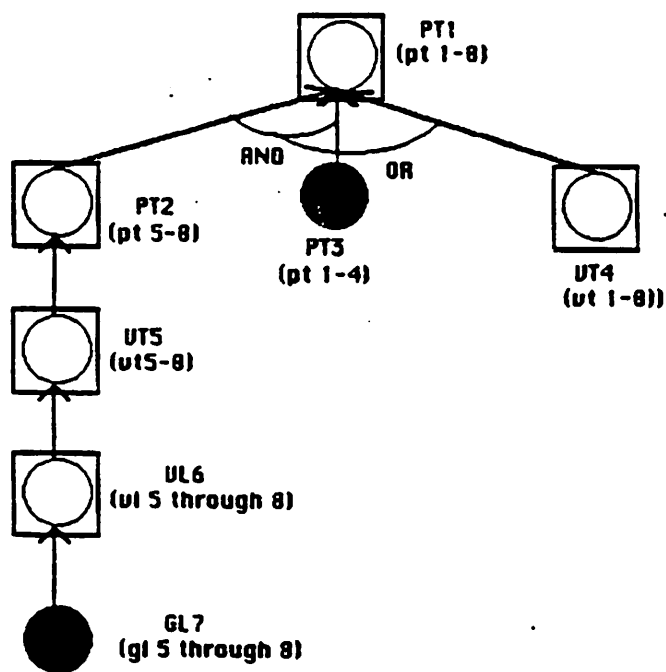


Figure 46: Instantiated SBM for Example II-Part I

The lack of the pt 1-8 hypothesis is first narrowed down to two regions: the existing pt 1-4 hypothesis was never extended (true-false pair PT3-PT1) and the necessary locations 5-8 were never driven up to the vl level from the gl level (true-false pair GL7-VL6).

such track exists, this state is false.

Diagnosis continues with the expansion of the back neighbors of the state PT2, in order to determine why the pt 5-8 was not created in the DVMT. The back neighbors of PT are PT and VT, but because no shorter segments exist for the PT 5-8 track, the PT neighbor is not instantiated. The VT neighbor is instantiated to represent the 5-8 track at the vehicle level. This is represented by the state VT5 and its associated object VT-HYP-OB5. Again, since this state is false, its back neighbors are expanded. These are VT (shorter vehicle track segments) and VL (vehicle locations necessary for the track). Because these two neighbors are linked by the PrefOR operator, only one will be instantiated and since there are no shorter track segments, it is the location neighbor that is instantiated. Because four vehicle locations are necessary to produce a vehicle track 5-8, four VL-HYP-OBJECTS must be created, one for each location. However, since none of these hypotheses exists in the DVMT system, they can all be grouped under one state, a false VL state. The result is the state VL6 with its four associated objects, VL-HYP-OB6, VL-HYP-OB7, VL-HYP-OB8, and VL-HYP-OB9. (The abstracted objects are not shown in the figures but are included in the traces in Appendix D.)

BCT continues diagnosis by expanding the back neighbor of the false VL6 state. The back neighbor of VL is GL, so objects at the GL level are instantiated next, along with their states. The result is four objects, GL-HYP-OB10, GL-HYP-OB11, GL-HYP-OB12, and GL-HYP-OB13. As before, all these object can be grouped together under one state. This time however, that state is true,

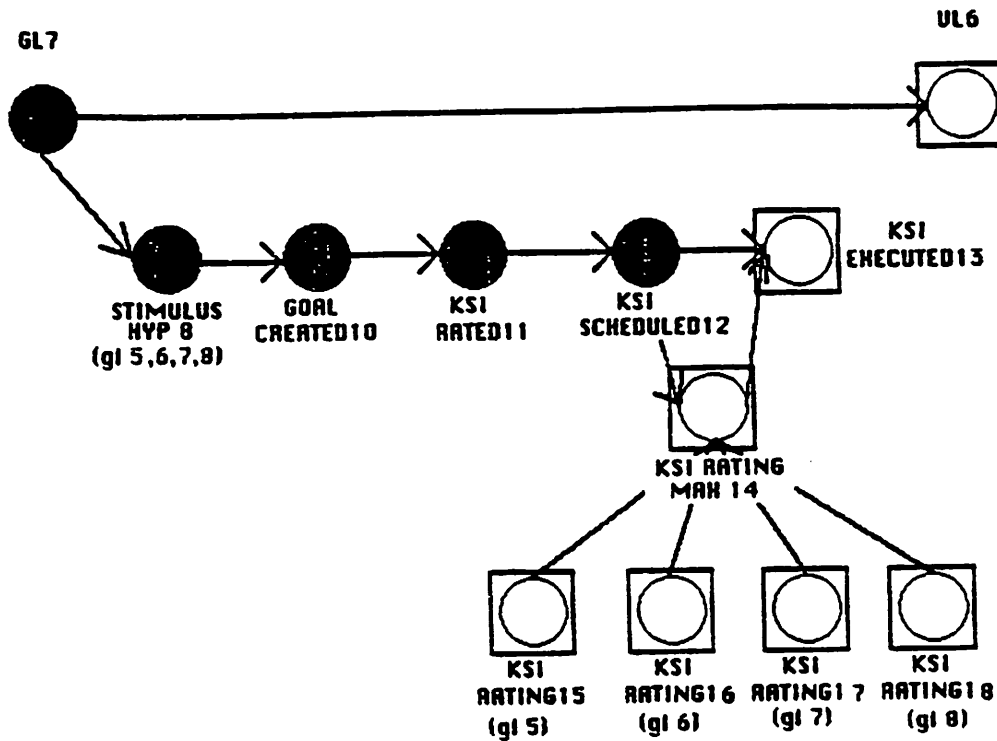


Figure 47: Instantiated SBM for Example II-Part II

The analysis of the GL7-VL6 true-false pair determines that the reason the gl hypotheses were not extended to vl was because the necessary ksi's were too low rated (false states KSI-RATING15-18).

because all these objects do exist. Since the data necessary to produce the vehicle locations 5 through 8 do exist at the gl level, the problem must lie in between the gl and the vl levels, in the ksi scheduling and execution process. This break in the expected processing is represented by the true→false pair GL7→VL6. The DM therefore switches to a lower cluster, the Ksi Scheduling Cluster, shown in Figure 47. Here the DM traces forward from the existing gl hypotheses to see where the correct processing stopped; whether the necessary ksIs were scheduled and if so, whether they executed. The DM first determines that the stimulus hypothesis was produced (true state STIMULUS-HYP8) and that it stimulated

the creation of the appropriate goal (true state GOAL-CREATED10). It then determines that the necessary ks was rated (true state KSI-RATED11) and that it was inserted onto the scheduling queue (true state KSI-SCHEDULED12). The next state should have been the execution of this ksi and the creation of the vl hypotheses. This is where the correct processing stopped however, because the state KSI-EXECUTED13 is false, indicating that the ksi never ran. Since it is always the highest rated ksi on the scheduling queue that executes, this ksi's rating must have been lower than the maximum. This fact is represented by the state KSI-RATING-MAX14, which is false.

At this point the DM invokes CR in order to determine why the rating of the ksi that would work on the correct data is so low. Let us call these ksi's **problem ksis** and the maximally rated ksi on the queue the **model ksis**. Notice that this naming comes from viewing the low rated ksis as a problem as far as deriving the rest of the desired track, and viewing the maximally rated ksis as a model with which to compare the low rated ksis. It is not meant to indicate that the problem ksis are not working on good data or that the model ksis are.

The DM now switches to CR and to a cluster representing the derivation of the ksi rating, the Ksi Rating Cluster, shown in Figure 48. The back neighbor of the state KSI-RATED-MAX14 is therefore a state representing the rating of a knowledge source instantiation. In this case, there are four ksis, one for each of the four gl hypotheses in locations 5 through 8. Four KSI-RATING states are therefore instantiated; KSI-RATING15 through KSI-RATING18, corresponding to hypotheses in gl 5 through 8. Only one of these, gl 5, will be discussed in

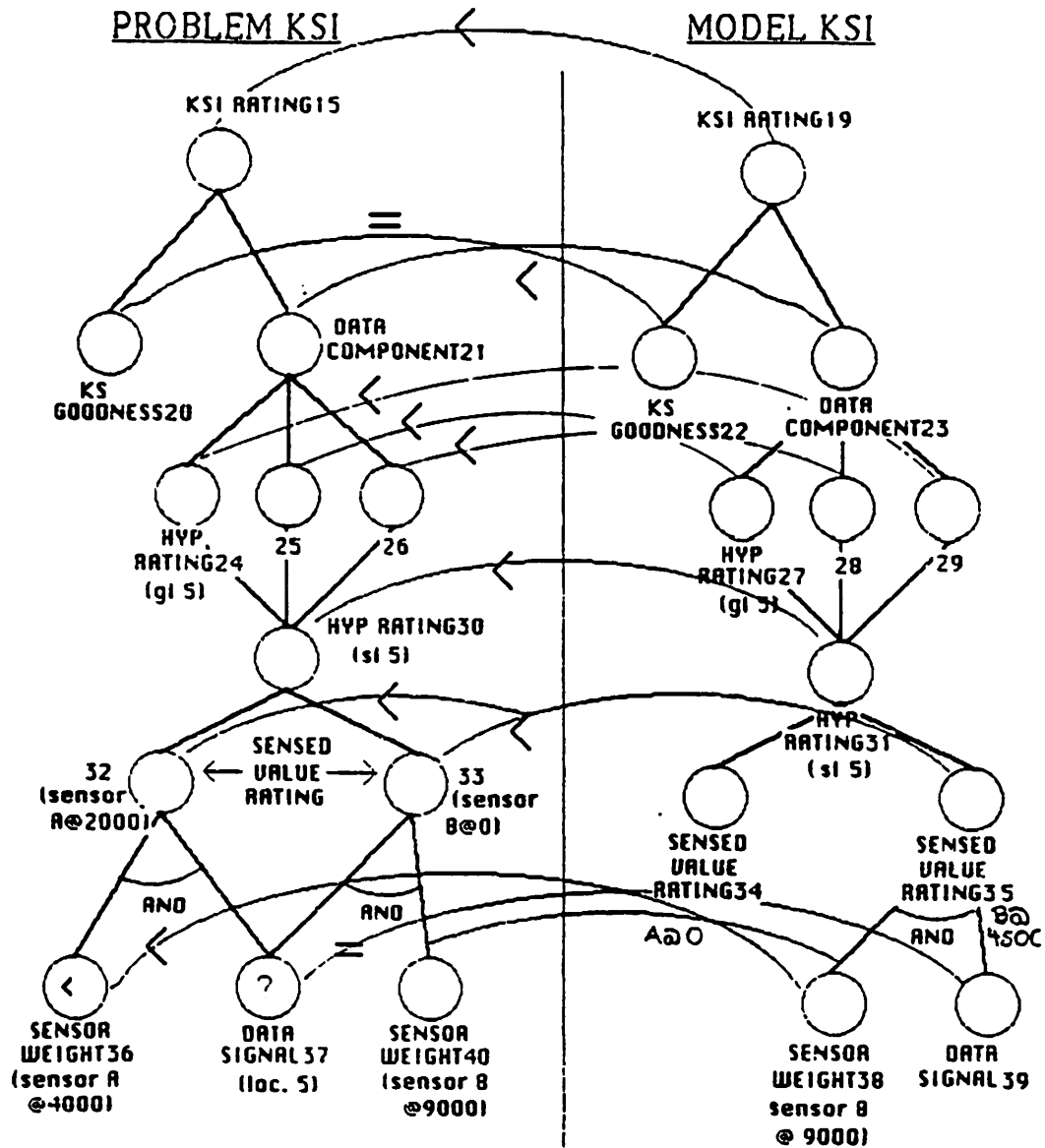


Figure 48: Instantiated SBM for Example II-Part III

This figure shows the parallel instantiation of the model representing the ksi rating derivation for the ksi being diagnosed (left half of the figure) and the ksi being used as a standard for comparison (right half of the figure). CR is invoked here which uses the qualitative relationships among the corresponding states in the parallel diagnoses.

detail. The diagnosis of the others is analogous and is not discussed here. The trace in Appendix D shows only the parts of the example discussed here.

Before CR can continue, a model object must be found for the low rated ksi. Such an object must be of the same type of the problem object and it must of course be rated higher than the problem object. In this case we are looking for a knowledge source that takes a hypothesis at the g1 level and produces a corresponding hypothesis at the v1 level. Such a ksi indeed exists; it is one of the highly rated ksIs stimulated by the highly rated hypotheses produced as a result of the noise sensed by Sensor B. The DM chooses this highly-rated ksi as the model with which to compare the problem ksi. The model ksi is represented by the state KSI-RATING19.

Notice that there are now two parallel instantiations of the Ksi Rating Cluster; one for the problem ksi and one for the model ksi. The two clusters will be instantiated one state at a time and compared, in an attempt to find an explanation for the low rating of the problem ksIs.³ CR uses relationship states rather than predicate states. The search through the model (i.e., which neighbors will be expanded next) is now determined by the type of relationship found among the problem state and the model state (i.e., $<$, $+$, or $>$), rather than the state value (true or false), as was the case with predicate states. CR continues to expand back neighbors, as long as they can explain the current state's relative value with respect to the parallel state. A state's relative value is explained by

³Only the diagnosis of the ksi for hypothesis in location 5 will be described in detail; the analysis of the remaining 3 ksIs (ones corresponding to locations 6 through 8) is analogous.

its predecessor states' values if they have the same relationship. For example, a < state is explained by its preceding < states, but not by > or by = states. (Chapter IV explains the details of CR.)

The value for state KSI-RATING15 is of course < (this is guaranteed by the process selecting the model object; if no appropriate model exists, no object will be instantiated, the problem state will not have a parallel state and diagnosis will stop.) In order to understand why KSI-RATING15 is low⁴ the DM expands its back neighbors to see if any of them are abnormally low. Since a ksi rating is a function of the ks goodness (a parameter which determines the quality of a knowledge source) and the data component (the ratings of the hypotheses the ksi is working with), the back neighbors of the state KSI-RATING are KS-GOODNESS and DATA-COMPONENT. These are instantiated and their values are determined from the values of their corresponding objects in the DVMT system. The resulting states are KS-GOODNESS20 and DATA-COMPONENT21.

Similarly, the back neighbors of the model ksi rating state, KSI-RATING19 must be expanded, so that the comparisons can continue. This expansion produces the states KS-GOODNESS22 and DATA-COMPONENT23. Before the values of these states can be determined, the parallel states must be matched. (Recall that the value of relationship states is determined by comparing the ratings of the problem and model objects.) This is done by evaluating the parallel-state attribute of each of the problem states. This attribute specifies the criteria

⁴Low here really means "lower than the model object" since there is no such things as absolutely low or high.

for state matching in a declarative fashion so it can be changed easily. Currently the criteria are that the state must be of the same type, if the state represents a hypothesis, the parallel hypotheses must be at the same level, and the value of the model object must be higher than the problem object. In the case of the ksi rating, the state matching is easy, since we have only one state of each type on both sides (the problem and the model) of the diagnosis. Evaluating the expression in the parallel-state attribute of the expanded states results in the matching up of the problem state KS-GOODNESS20 with the model state KS-GOODNESS22, and the problem state DATA-COMPONENT21 with the model state DATA-COMPONENT23. The relationship of = is found for the KS-GOODNESS states, because the ks goodness values are identical. The relationship of the DATA-COMPONENT states is <, because the value of the data component rating of the problem ksi is lower than the value of the data component rating of the model ksi.

The next step is to select a subset of the expanded back neighbors to expand further. As in BCT, we want to continue expanding only those states that explain the current problem. The current problem is a low ksi rating and we have determined that one of the components influencing this rating is normal (ks goodness) and one is below normal (data component); where "normal" means "same as the parallel state". Clearly the normal value could not have caused the ksi rating to be low. The KS-GOODNESS20 state is therefore a dead-end as far as the diagnosis is concerned, because this state is not causally related to the KSI-RATING15 state. The low DATA-COMPONENT21 state however is responsible for the low

KSI-RATING15 and we therefore follow this state backward, by expanding its back neighbors.

The value of the data component of a **ksi** is a function of the data (i.e., the stimulus and the necessary hypotheses). In this case, there were three **gl** hypotheses, one for each event class, whose rating determined the rating of the data component. (Knowledge sources that transform lower level hypotheses into higher level ones often combine several low level hypotheses of different event classes into one higher level one.) The back neighbor of **DATA-COMPONENT**, **HYP-RATING**, is therefore instantiated into three states, one for each of the three hypotheses of the problem **ksi**.⁵ These states are **HYP-RATING24**, **HYP-RATING25**, and **HYP-RATING26**. Their associated abstracted objects are objects representing group location hypotheses, **GL-HYP-OBs**.

The state matching is more difficult here than before because there are three parallel states to choose from on the model side; each problem **HYP-RATING** state has to select one of the model **HYP-RATING** states. In this case a heuristic is used to select the appropriate model state: the **DM** looks for a model state that minimizes the difference between the ratings of the two objects while maintaining the constraint that the model rating must be higher than the problem rating. The reason for doing this is so as not to waste the difference between the two ratings.

For example, let us suppose that we have two problem states to match with

⁵No grouping of similarly behaving objects is done during **CR**. Therefore a state is created for each object, resulting in three **HYP-RATING** states.

two model states. The problem states pA and pB have the ratings 1000 and 1500 respectively. The model states mA and mB have the ratings 3000 and 1200 respectively. Suppose the state matching process selects the first of the model states, mA. The two ratings to compare are then 1000 of pA and 3000 of mA; the result is a < value of the state pA. However, when we try to match up the B states, the problem state pB has a value of 1500 which is higher than the model state mB, 1200. These states cannot therefore be causally linked and the diagnosis will continue with only the A states. This reduces the number of causal pathways to investigate and may cause the DM to miss some important problems further down the rating derivation pathway. In order to avoid this problem, **anytime there is more than one candidate model state to match, we select the one that minimizes the difference between the states and still maintains the desired relationship among the values of the two states, namely that the problem state is lower than the model state.**

In the current example, this difference minimization results in the following state pairs: HYP-RATING24 and HYP-RATING29, HYP-RATING25 and HYP-RATING27, and HYP-RATING26 and HYP-RATING28. The values of these states are <, since all the hypotheses ratings are lower than the model hypotheses ratings. Diagnosis therefore continues with the expansion of the back neighbors of these states. We begin with the state HYP-RATING24 and expand its back neighbors. The back neighbor of this state is another HYP-RATING state, representing the rating of the hypothesis at the sl level from which the gl hypothesis referred to by the state HYP-RATING24 was derived. The resulting

state is HYP-RATING30. We also expand the back neighbor of the parallel state which results in the instantiation of the state HYP-RATING31. The state matching is trivial here since there is only one model state to choose from. The two states are matched, the value of state HYP-RATING30 is determined to be lower than its parallel state HYP-RATING31, and diagnosis continues by expanding the back neighbors of this state. This is the state SENSED-VALUE-RATING and, since a signal hypothesis is derived from however many sensed values there are for the signal, this neighbor is instantiated into two states, because both sensors A and B cover the area for location 5. The resulting states are SENSED-VALUE-RATING32 and SENSED-VALUE-RATING33, representing the signal sensed by sensor A and B respectively. Similarly, the back neighbors of the model state HYP-RATING31 are expanded into SENSED-VALUE-RATING34 and SENSED-VALUE-RATING35 since the model hypotheses could also be sensed by both sensors.

Recall now that the problem ksi was to work on the correct signal in location 5. Since this signal was sensed only by Sensor A, only that sensor will have a value for it. Hence the actual values of the two SENSED-VALUE-RATING states; 2000 for the Sensor A value (SENSED-VALUE-RATING32), 0 for the Sensor B value (SENSED-VALUE-RATING33). Similarly, the noise signal represented by the model states is sensed only by sensor B so the actual value of SENSED-VALUE-RATING35 is 4500 and the actual value of SENSED-VALUE-RATING34 (representing the signal sensed by sensor A) is 0. Since only one of the model states is higher than both of the problem states, both are matched

to this state. The result is the two pairs: **SENSED-VALUE-RATING32** and **SENSED-VALUE-RATING35** and **SENSED-VALUE-RATING33** and **SENSED-VALUE-RATING35**. As before, the idea in matching the states, is to explore as many of the problem states as possible. We could have matched the states 32 and 35, and 33 and 34. This however would have resulted in $<$ and $=$ values and only the state **SENSED-VALUE-RATING32** would have been investigated further. This would reduce the search and, as we shall see, would cause the DM to miss the fact that the data signal value was inconsistent with the sensed values of the two sensors.

The result of the state matching then gives two states to follow, both with $<$ value: **SENSED-VALUE-RATING32** and **SENSED-VALUE-RATING33**. Since the search is depth-first, we first follow the **SENSED-VALUE-RATING32** state. The rating of the sensed value is a function of both the actual value of the signal (**DATA-SIGNAL**) and the sensor weight parameter (**SENSOR-WEIGHT**), the back neighbors of the state **SENSED-VALUE-RATING** are therefore **DATA-SIGNAL** and **SENSOR-WEIGHT**. We expand these neighbors for both the problem state **SENSED-VALUE-RATING32** and the model state **SENSED-VALUE-RATING35**. This results in states **SENSOR-WEIGHT36** (representing the low sensor weight for the right half of sensor A's area) and **DATA-SIGNAL37** (representing the actual signal for location 5), and **SENSOR-WEIGHT38** (representing the high sensor weight for sensor B) and **DATA-SIGNAL39** (representing the noise signal of the model ksi, a signal in location 7). (Ignore, for the moment, the additional **SENOR-WEIGHT40** state shown in the figure. This state will

be generated when the **SENSED-VALUE-RATING33** state's back neighbors are expanded.) The state matching here is trivial, since only one state exists of each type on both the problem and the model sides of the investigation. The result is the two pairs: **SENSOR-WEIGHT36** and **SENSOR-WEIGHT38**, and **DATA-SIGNAL37** and **DATA-SIGNAL39**. Since the sensor weight of A is lower than sensor weight of B, **SENSOR-WEIGHT36** value is $<$. The state **SENSOR-WEIGHT** is a primitive state and we have thus identified the first problem: the low sensor weight parameter of Sensor A which was at least partially responsible for the low rating of the problem **ksi**.

The value of the signal in location 5, as determined from the sensed value and the sensor weight, happens to be the same as the value of the noise signal. The value of state **DATA-SIGNAL37** is therefore $=$ to its parallel state, **DATA-SIGNAL39**, and the investigation ends here. However, there are more problem states to follow and, as it turns out, following these states will lead to an inconsistent value for the data signal value, which will point to the second problem: a malfunctioning sensor.

DM continues the diagnosis with the state **SENSED-VALUE-RATING33**, which represents the value sensed for location 5 by sensor B. Since B did not sense location 5, this value is 0. DM next expands the back neighbors of **SENSED-VALUE-RATING33** and creates **SENSOR-WEIGHT40** (representing sensor weight for sensor B) and again, **DATA-SIGNAL37**, since we are dealing with the same location as before. (The back neighbors of the parallel state of **SENSED-VALUE-RATING33** have already been expanded since this was also the parallel state of

the SENSED-VALUE-RATING32 analyzed before.) In determining the value of the state DATA-SIGNAL37, which must be recalculated now that another sensed value for this signal has been instantiated, the DM discovers that a consistent value cannot be determined. According to the value sensed by sensor A (and its sensor weight parameter), the data signal value should be 5000. But according to the value sensed by sensor B, the value should be 0, since sensor B did not produce a signal for this location. This points out a problem and all objects involved in this problem are suspect; that is both sensor A and sensor B could be malfunctioning. For now, the diagnosis ends here by simply pointing out the inconsistency. If the Inconsistency Resolving mechanism was implemented, the diagnosis would continue by comparing the behavior of the two sensors with the behavior of an idealized sensor. The sensor that was closer to the model would be considered correct, the other one would be assumed to have failed. This mechanism has not yet been implemented but some ideas for it are discussed in Chapter X.

The diagnosis has now identified both problems: the low sensor weight parameter for Sensor A and the inconsistent data signal which makes both Sensor A and Sensor B suspect. We still have to finish diagnosing the remaining HYP-RATING states, HYP-RATING25 and HYP-RATING26. This diagnosis however does not reveal any more problems, because it merges with the existing instantiated model at the state HYP-RATING30. An annotated trace for this example is in Appendix D.

§4. Reducing the Complexity of the Diagnosis

This section discusses the methods we have used to reduce the complexity of the diagnosis. Why do we have to worry about this issue? In a typical run, the DVMT creates hundreds of objects in each of its processing nodes. In order to diagnose a given situation, a subset of these objects has to be represented in the instantiated SBM, the model then has to be searched in an attempt to find the causes for the situation. The large number of objects and states both contribute to making the diagnosis an expensive process. It is expensive both in terms of the time it takes to create and process the object/state structures, and in terms of the memory space they occupy.

In order to make the diagnosis feasible for problems of reasonable size, we had to devise methods for reducing both the time and the space requirements. Recall that diagnosis proceeds by instantiating the appropriate part of the system model at each diagnostic step. This process ends when a primitive node is found or when the model can no longer be expanded. The model instantiation consists of creating two types of structures: the abstracted objects, which represent objects in the DVMT, and the states, which represent either some predicate describing the object's status in the system, or its relationship to another object. In this context, reducing the complexity could mean either the reduction in the number of structures created (i.e., reducing the size of the instantiated model), or reduction of the cost of instantiating each structure in the model. We have chosen three techniques for complexity reduction by reducing the size of the instantiated

model. These techniques are briefly described below and discussed in detail in the remainder of this section.

OBJECT GROUPING reduces the number of states created by taking advantage of the fact that the state values can fall into very few classes. Typically these are non-expandable,⁶ true, and false for the predicate states, and less-than, equal-to, and greater-than for the relationship states. In cases where the splitting attributes⁷ of an object cause a number of objects to be created, these objects can then be grouped into a small number of classes according to the values of their corresponding states. Instead of creating a separate state for each object, one state is created for each different state value and all objects falling into that category are attached to it. Because there is a fixed number of classes into which a state value may fall, there is a limit on the number of states created, whereas there is no limit on the number of objects created, since that depends on the data being analyzed and the system parameter settings.

UNDERCONSTRAINED OBJECTS allow the representation of a class of objects by one object. This reduces the number of objects that have to be created and, consequently, also reduces the number of states.

HIERARCHICAL MODEL STRUCTURE allows the diagnosis to be performed at different levels of abstraction, starting with the highest possible, and dropping down to the more detailed levels only when the more general diagnosis has already pinpointed the problem to that area. By postponing detailed diagnosis, the hierarchical structure helps reduce the number of states and objects that have to be created and examined.

⁶These are states referring to objects whose existence is not possible, given the current interest area parameter settings.

⁷These are attributes that may require the creation of multiple objects from an uninstantiated object.

The rest of this section will discuss in detail how each of these mechanisms reduces the complexity of the diagnosis by reducing the number of structures that have to be created during the model instantiation.

§4.1 Object Grouping

To understand object grouping method it is important to recall that the model consists of two parallel networks: one network containing the abstracted objects and the other containing the states the objects are expected to undergo. Prior to the object grouping, each time a new state was instantiated in the model, the necessary number of objects was created and then a separate state was created for each object. This quickly led to a combinatorial explosion of both objects and states.

Consider the situation in Figure 49 where a pattern track of length 8 is being analyzed. The initial PT-HYP-OB representing the track and its associated state PT, is first expanded into 8 PL-HYP-OB objects representing the individual locations at the pattern location level. A state is created for each of these objects resulting in the creation of 8 states. Since none of the objects exist in the DVMT system, all these states are false. Since the states are false, the problem must lie at some point before these states are reached, therefore the back neighbor states and their corresponding objects are expanded; these are states representing the hypotheses at the vehicle location level. But because each signal at the pattern level is actually generated from 2 signal types at the vehicle level, each of the 8 pattern locations is now split into 2 vehicle locations. This leads to 16 VL-

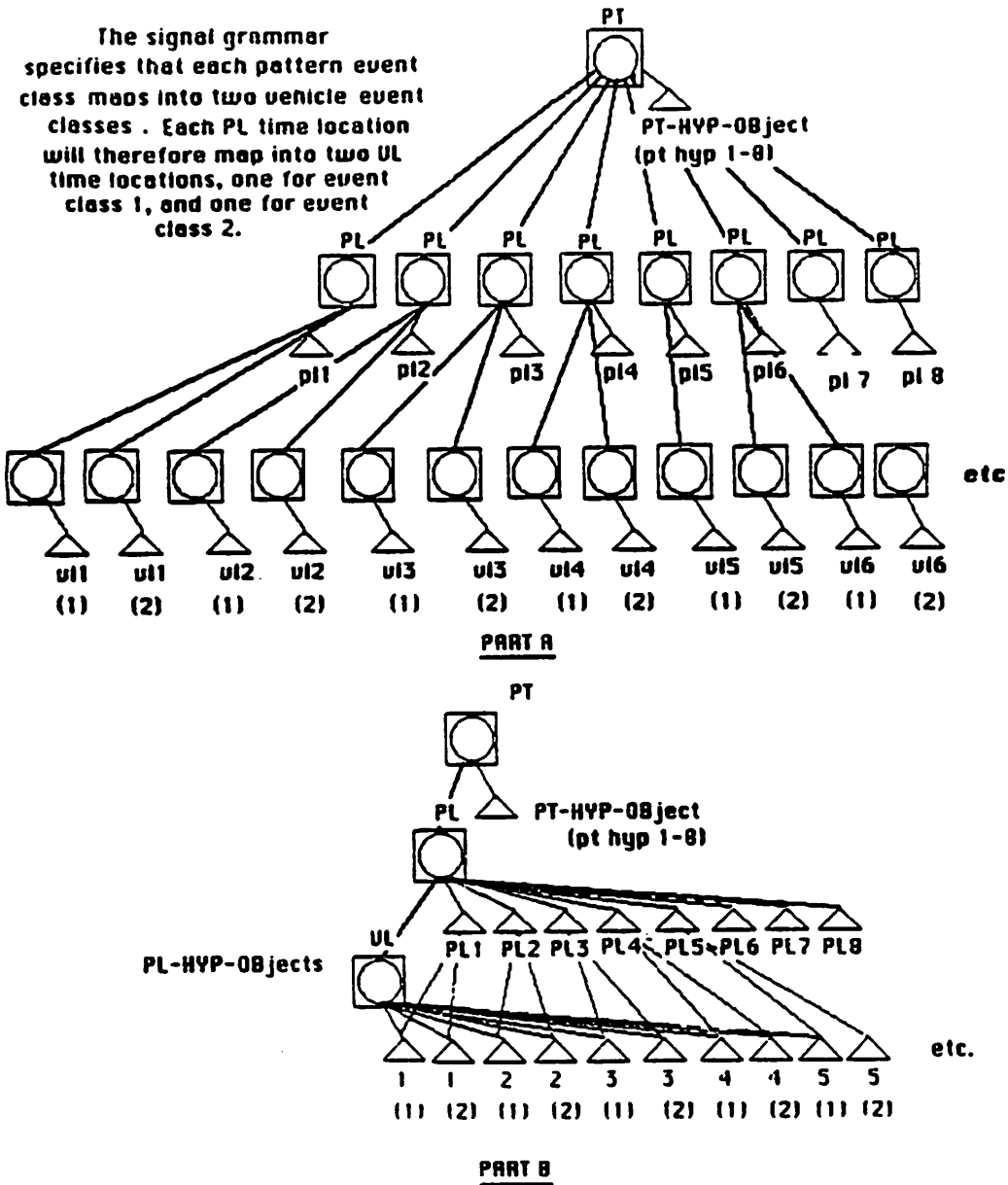


Figure 49: Reduction of the Number of States by Object Grouping
 This figure illustrates the reduction in the number of states created when similarly behaving objects are grouped under one state. Without such grouping the diagnosis of even an uncomplicated situation in the DVMT would be too costly to be useful. Part A shows the number of states and objects that would be created without grouping together similarly behaving objects under one state. Part B shows the instantiated model for the same diagnosis using object grouping.

HYP-OB's and their corresponding 16 states. Let us suppose that none of these objects exist, so all the state values are again false. Each state's back neighbors are expanded again; the states representing hypotheses at the group location level. (This further expansion is not shown in the figure.) Again, the system signal grammar indicates that each vehicle signal is composed of 2 group signals. The 16 objects at the vehicle location level therefore map into 32 objects at the group location level. Since a separate state is created for each of these objects, this leads to 32 gl states. Again, all these states are false, resulting in the expansion of their back neighbors, the states representing the hypotheses at the signal location level. Once again, there is a split in the signal grammar, because each group hypothesis is made up of 2 signal hypotheses. This results in two SL-HYP-OB's being created for each GL-HYP-OB. The number of objects therefore comes to 64, as does the number of their corresponding states.

Thus in order to analyze a pattern track of length 8 with the given system signal grammar, without object grouping, we would need to create

$$1 \text{ (pt)} + 8 \text{ (pl)} + 16 \text{ (vl)} + 32 \text{ (gl)} + 64 \text{ (sl)} = 111$$

objects and the same number of states, for a grand total of 222 structures.

Letting L represent the length of the original pattern track, PV, VG, and GS representing the number of signals at the lower level that make up one signal at a higher level, for the vl, gl, and sl levels respectively, the number of objects created to analyze missing data for a pattern track of length L is:

1 (original track at pt level) + L + (L * PV) + (L * PV * VG) + (L * PV * VG * GS). The number of states is the same as the number of objects since a separate

state is created for each object.

In the above example, if we examine the state structures created, we discover that their values are identical. Thus the only thing that differs for all the states of a particular type, is that they point to different objects. Object grouping takes advantage of the fact that the state values can fall into a very small number of categories. Typically, these values are true (indicating that some situation in the system exists), false (indicating that a situation does not exist), and non-expandable (indicating that the state cannot be true because of the interest area parameter settings and does not therefore need to be expanded). For relationship states there can also be three classes: less-than, equal, and greater-than. Thus regardless of how many objects the DM creates to represent some situation, they will be divided into a small, fixed number of classes by their state values.

The state value, together with the type of reasoning being done, determines which part of the model will be instantiated next. During BCT, for example, all false states will have their back neighbors expanded, and all true states will have their lower front neighbors expanded. Since all states with the same value are treated the same way, it is sufficient to create just one state for each group of objects whose states would have the same value. We thus obtain a dramatic reduction in the number of states necessary. In the example above, we still must create all 222 objects but the number of states is now 5 instead of 111. Furthermore, the maximum number of states can be determined from uninstantiated SBM, and is not a function of the data being analyzed or the DVMT parameters. Specifically, the maximum number of states instantiated in order to analyze some

CLUSTER	# of uninst. states	max # of states
Answer Derivation	11	33
Ksi Scheduling	9	27
Comm Ksi Scheduling	6	18
Communication	5	15

Figure 50: Maximum # of States Created for Each Cluster

situation is the number of states in the cluster times the number of possible state values (3 in most cases).

If we consider the number of structures created as a measure of the complexity, we have succeeded in reducing the worst case complexity of the model instantiation from

$O(F(\text{data being analyzed, DVMT parameters, number of states in cluster}))$

to $O(\text{number of states in cluster} * 3)$, which is constant, independent of the data being analyzed. Thus object grouping, while it does not reduce the number of objects created, does provide an upper limit on the number of states. The upper limit being the number of states in the model times the number of distinct values these states can have. Figure 50 lists the maximum number of states for each of the model clusters.

§4.2 Underconstrained Objects

In the above example, we were able to create fewer states by recognizing that all objects attached to a state with the same value are treated identically. We

could save even more if we could create just one object to represent a whole group of objects. Of course, this is only possible in cases where the diagnosis does not require the finer resolution afforded by representing each object individually. It turns out that exactly this type of a situation occurs when we want to simulate the effects of an identified failure on future system behavior, in order to account for any pending symptoms. In such cases it is often possible to leave out the details and concentrate on propagating forward only those object characteristics that will determine the types of objects affected by the identified fault. We can represent such a class of objects by underconstraining some attributes in the object's definition. For example, if we want to represent all hypotheses at the vt level, we use the object VT-HYP-OB and leave the time-location-list, event-class, and node attributes empty.

Consider the example where a communication knowledge source is missing. There are several symptoms pending diagnosis which are due to this missing ks. Normally, we would diagnose each symptom separately. Depending on the model being instantiated, this would lead to at least $(n * \text{\#-of-objects-in-model})$ objects, where n is the number of symptoms being diagnosed. Typically, the number of actual objects would far exceed just those in the uninstantiated model due to the object splitting attributes. The number of instantiated states would be the same number of states in the uninstantiated model, since they would all be false due to the failures.

Rather than diagnosing each symptom separately, we would like to deduce from the identified failure that no messages requiring the missing knowledge

source will be sent. In other words, we would like to be able to represent the entire class of objects affected by the missing ks as one object, an underconstrained object, and reason about it, rather than all individual messages affected. We can do this by leaving all but the level attribute of the message-objects empty. This will describe all messages occurring at the vehicle track level, regardless of their event class, destination node, or time-location-list.

Underconstrained objects thus allow a major reduction in the size of the instantiated model. Unlike the object grouping described above, the underconstrained objects reduce both the number of states and the number of objects created. When used for forward simulation of identified faults, the maximum number of objects that must be created is the number of uninstantiated objects. We therefore have a reduction from

$O(\# \text{-of-symptoms} * \# \text{-of-objects-in-the-model})$ to

$O(\# \text{-of-objects-in-the-model})$ number of created objects.

Since the number of uninstantiated objects is a function of the model, not the data or the number of symptoms, we have once again made the number of created objects independent of the data.

§4.3 Hierarchy

The use of hierarchy helps reduce both the number of states and the number of objects being instantiated, by focusing the diagnosis into the most general problem area first and examining the intervening detailed states only when necessary. The use of a hierarchical model in diagnosis is analogous to searching of sorted

files, if we view the number of states that must be instantiated as analogous to the number of elements that need to be examined. The difference between a flat and a hierarchical structure then becomes analogous to the difference between a sequential and binary search. Suppose we are looking for the i th element in an array containing n sorted elements. In a sequential search, we would have to scan all i elements before finding the desired one. Similarly, suppose we are looking for the cause of some error which happens to be in the i th of n states. We would have to instantiate all i states before reaching the one responsible for the error. Now consider a binary search, where we look at the $n/2$ th element and, if it is larger than i , we consider only the elements 1 to $n/2$, otherwise we consider the elements $n/2$ to n . Analogously, with a hierarchical model, we look at a state in the upper level of the hierarchy, which represents some m th state, where $m \ll n$. If this state is false, indicating that the error occurred before it, we have successfully avoided instantiating the intervening m states and the error must lie in the remaining $1-m$ states.

Let us consider as worst case the situation where the DM begins diagnosis with some symptom, scans a complete model cluster, and finds that the symptom lies beyond that cluster. For flat models where the number of states is n , the complexity of this worst case is $O(n)$, analogous to the same complexity for sequential search [18], since all n states must be instantiated. Now consider a hierarchical model, where at the top level of the hierarchy we place every m th state from the flat model. In the worst case we thus have to instantiate n/m states, where m is the step size determining the selection of the top level hierar-

chical states from the flat model. The worst case complexity in this case is thus $O(n/m)$, as opposed to $O(n)$ in the flat model case. As with sequential search, the success of a particular diagnosis depends on the distance of initial symptom from the failure. The further away the symptom is, the more effective the hierarchy will be in reducing the number of states that have to be instantiated.

§5. Summary

This chapter described in detail the analysis done by the DM in diagnosing two example problems in the DVMT system. Both problems began with the lack of a spanning pattern track hypothesis. The diagnosis traced back through the system model to uncover the primitive causes of these problems. Example I discussed a four-node DVMT system where the problem is the result of missing communication knowledge sources. This example illustrated the use of Forward Causal Tracing to predict the effects of the identified fault on future system behavior and thus eliminate the diagnosis of future symptoms caused by the same fault. Example II discussed a single-node DVMT system where the combination of a faulty sensor on one hand and a low sensor weight parameter on the other prevented the system from generating a spanning pattern track hypothesis. This example illustrated the use of CR as a heuristic to help focus on the problem areas. The description of the diagnosis included the instantiated models produced for each of the examples. The corresponding annotated traces are in Appendix D.

The last section of this chapter discussed three methods for reducing the com-

plexity of the diagnosis by reducing the size of the instantiated model. The **object grouping** reduces the number of instantiated states by grouping related objects together and creating one state for the whole group. Use of **underconstrained objects** to represent a class of objects reduces the number of objects (and consequently also the number of states) that must be created to represent a particular situation. The **hierarchical structure** of the model reduces both the number of states and objects that must be instantiated by delaying detailed diagnosis until necessary.

C h a p t e r VI

BUILDING A CAUSAL MODEL

**There seems to be no simple way to explain
that these concepts are perfectly obvious.
- J. Kilpatrick**

This chapter is intended for those readers who would like to construct a similar type of a causal model for a different application. It summarizes some principles for construction of causal models and focuses on dealing with the combinatorial explosion inherent in modeling a problem-solving system. Specifically, it discusses the following:

- The selection of states to represent in the system model.**
- The organization of the states into the model hierarchy.**
- The selection of the abstracted objects.**
- The construction of the constraint expressions linking the attributes of the abstracted objects and expressing the dependencies among them.**
- The types of abstractions necessary to reduce the system complexity sufficiently enough to make it feasible to reason about the large space of possible system behaviors.**

§1. Choosing the States

The correct behavior of the DVMT system is modeled by a state transition diagram. This state transition diagram represents the correct sequence of states the problem-solving system is expected to go through as it constructs the map of the vehicle motion through the environment from its input data. This section discusses the issues in selecting what to represent as a state and how to organize these states in order to capture the causality among the events. There are three major points here:

1. What aspects of the system to model. Even though this is a causal model and therefore, in principle, able to detect and diagnose previously unknown failures, the range of the failures is still limited by what the model represents. It is important that the model builder be aware of this and not expect magic to take place once the model is used by the diagnostic interpreter.
2. How to represent enough of the system behavior to allow at least the narrowing down of a problem in cases where the error cannot be fully diagnosed.
3. How to make the model as concise as possible. Whenever possible, the specifics of the data should be abstracted and contained in the functions in the constraint expressions so that the model is invariant with respect to the specific data.

Choosing What to Model. How do we choose which aspects of the DVMT system behavior to represent as states in the System Behavior Model (SBM)? The exact choice depends on the application. We may decide, for example, that we are not interested in certain types of faults and can therefore omit modeling that part of the system. The problem of limiting what to model is encountered anytime a

causal model is constructed. Weiss expresses this for his system, CASNET, which models the mechanisms of glaucoma.

A state network can be thought of as a streamlined model of disease that unifies several important concepts and guides us in our goal of diagnosis. It is not meant to be a complete model of the disease [39].

Having decided what parts of the system behavior to represent we have to decide how closely the represented states should correspond to the results of single events in the DVMT system. In some cases the model builder can make the analysis of the system behavior easier by making certain situations explicit, even though those situations are not the result of a single event. For example, there is no single event causing the rating of a knowledge source to be low, but by representing explicitly the state where the knowledge source rating is not the highest on the knowledge source queue, the analysis of certain failures is made easier by focusing the attention of the Diagnosis Module (DM) into the appropriate part of the model. There are states that are used as reasoning transition points. For example, the states KSI-RATING-OK and KSI-RATING-MAX indicate that a transition should be made to Comparative Reasoning to investigate why the ksi rating was so low. The model building process is therefore very much influenced by the model builder's knowledge about the likely system failures. The model still represents the correct system behavior but it is skewed to make the analysis of certain types of failures more efficient.

Narrowing Down the Problem. Due to the choices of the model builder, not all errors will be diagnosable and the diagnosis of some may be more difficult because of the way the system behavior is represented. If the model is constructed

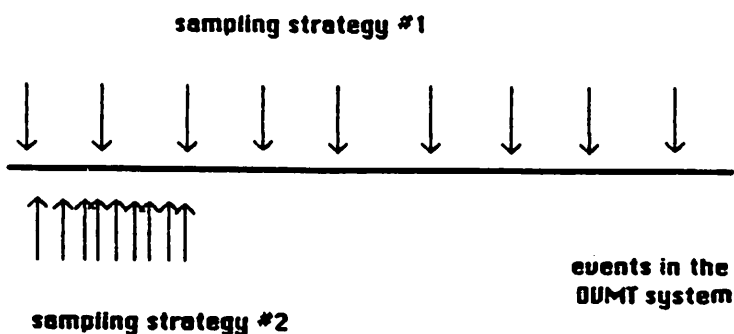


Figure 51: Strategies for Choice of States in the SBM

Given a limited model size, we have to choose which of the many states in the system behavior we will represent in the model. It is important that we represent enough of the system behavior to allow as much coverage as possible. This is akin to placing guards around an area to be protected. It is better to spread them out (as in the arrows facing down in the figure) than to clump them all together (as in the arrows facing up) and leave much of the area unguarded.

well, then it will allow sufficient narrowing down of the problem to still be useful. In other words, the states of the system represented in the model must represent enough of the space of the possible system behaviors so that the DM can at least identify the general area of the system where the problem occurred. Figure 51 illustrates the different strategies for "state sampling" in the DVMT system representation. If we view the system behavior as a stream of events, then choosing what to represent is analogous to selecting an appropriate sampling rate of these events. This rate (i.e., which events we represent in the model) depends on what type of resolution we want to achieve; the more detailed the model, the more events we can diagnose. Even a high-level model is useful however, because it narrows down the problem, making it easier for the user to complete the finer diagnosis.

In modeling the DVMT system behavior, we found that representing the dis-

tinct levels of abstraction of the data during the transformation of the input signals into the final answer is appropriate as the highest level model of the DVMT system behavior because it quickly narrows down the problem. This is the Answer Derivation Cluster shown in Figure 25 in Chapter III.

The diagnosis usually begins with a state in this high level model. The problem is narrowed down by finding the points where the expected answer derivation stopped. For example, we might find that the expected hypothesis was created at some level on the data blackboard but was not driven up to the next level. When such a break in the expected processing is found, the diagnosis switches to a lower level model, which allows reasoning about the system at a finer resolution.

In some cases the model will not have a representation of the events occurring in between the normal and the abnormal state, because the model builder chose not to include those events. An interesting extension of the DM would be the capability to recognize the situation where the model is incomplete and prompt the user for more information, which would then be integrated into the model.

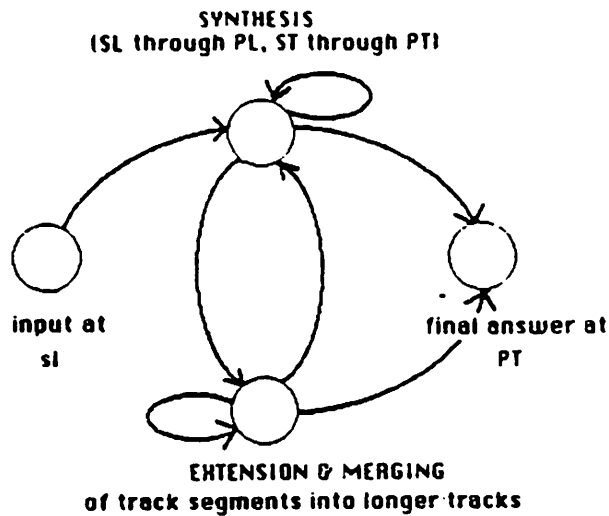
Model Conciseness. Finally, there is the issue of model conciseness and its effect on state selection. We would like the model to be as concise as possible. The criteria for making the model concise are the same ones used in programming in deciding what to make into a subroutine. The model builder must try and parametrize as much of the system behavior as possible. To do this, he must first identify the repeated sequences of actions in the system and then try and represent them in a general way. This representation is then placed in a separate model cluster which is instantiated any time the set of actions it represents oc-

curs. Examples of this type of modeling are the KSI-SCHEDULING-CLUSTER and the COMM-KSI-SCHEDULING-CLUSTER. These clusters and their relationship to the higher level answer derivation cluster and communication clusters are discussed in Chapter III.

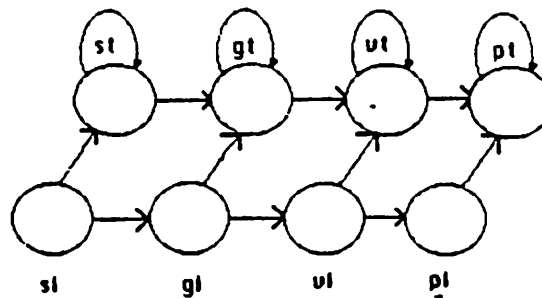
The desire for model conciseness has to be balanced against the complexity of the constraint expressions linking the abstracted objects in the model. The more concise the model, the more information it contains, and therefore the more complex the individual constraint expressions linking the abstracted object attributes will be. Figure 52 illustrates two versions of the answer derivation model. The model shown in part A [6] is clearly a much more concise description of the answer derivation process than the model actually used, shown in part B. However, constructing the constraint expressions linking the hypotheses objects in this model is a much more demanding task than constructing them for the model in part B.

§2. Organizing the States into a Hierarchical Model

We have selected the situations we want to represent as states in the SBM and now have an unordered set of states. We now face the same problem as the actor in a Monty Python skit: "I don't want you to get the impression that it's just the number of words, getting them in the right order is just as important". How do we link these states so that they represent the proper causal pathways in the system behavior and how do we partition them among the different levels of



PART A



PART B

Figure 52: Two Versions of the SBM Modeling the Same Process
 This figure shows two ways of representing the same process: the derivation of the pt answer from the sensed signals at the sl level. Part B shows the currently used model, where each intermediate level of abstraction is shown. Part A shows a more concise model, where some of the intermediate levels are grouped together under one state. The upper model, when instantiated, would resemble portions of the lower model.

the model hierarchy? The issues to consider here are:

- appropriately separating the causal pathways,
- placing each state at the place in the hierarchy that makes diagnosis most efficient, and
- making the construction of the constraint expressions linking the abstracted objects as simple as possible.

Causal Pathways. Figure 53, part A illustrates the events that take place when a newly created hypothesis is inserted onto the data blackboard. The states these events induce in the system are also shown. We have selected a subset of these states to represent in the model and we have to link them up so as to represent the appropriate causal relationships among them. We could construct a model exactly duplicating the events like the one shown in Figure 53, part B. Note that in this model duplicates the exact sequence of events/states as they occur in the system but only partially captures the causality relationships among those events. In other words, the events/states from separate causal pathways are interleaved in one pathway in the model. For example, the state **KS-EXISTS** has no causal predecessor states in the model and yet is preceded by the state **GOAL-CREATED**, which is in no way related to it. Similarly, the state **NECESSARY-HYP** follows the state **KS-EXISTS** even though there is no causal relationships among those two, or among the state **NECESSARY-HYP** and any of the preceding states. In order to facilitate inferences based on causal relationships, we must make causally related states contiguous and separate them into distinct causal pathways. A model that achieves this aim is shown in Figure 53,

part C. In order to construct the causal model we need, we must first separate the states into causally related sets (i.e., separate the dependent and the independent events/states) and then arrange these in the proper sequence. Independent (i.e., causally unrelated) events should be represented by separate pathways. Dependent (i.e., causally related) events should be represented sequentially, on the same pathway. This type of a state organization then allows the system to make inferences relying on the causal interactions among the states. For example, in the model shown in Figure 53, part B, if the state GOAL-CREATED is false, we cannot say anything about the states KS-EXISTS and NECESSARY-HYP, because these in no way depend on the state GOAL-CREATED. In the model in Figure 53, part C however, if the state GOAL-CREATED is false, we can make exactly the set of inferences we need; namely, that this implies that the states KSI-RATED, KSI-SCHEDULED, and KSI-EXECUTED will all be false but tells us nothing about the states KS-EXISTS and NECESSARY-HYP.

Organising the States into a Hierarchy. The hierarchical organization determines the order in which the states will be instantiated; i.e., it imposes a search order on the model. By putting some states at a higher level in the model we are determining that those states will be examined first when reasoning about the system. This means that we should use our knowledge of the likelihood of certain events being the primary causes of other events and put these primary-cause states at the higher levels of the hierarchy so they are looked at first. If we imagine the diagnosis as viewing the system behavior through an increasingly more powerful microscope, then the process of deciding where in the hierarchy to

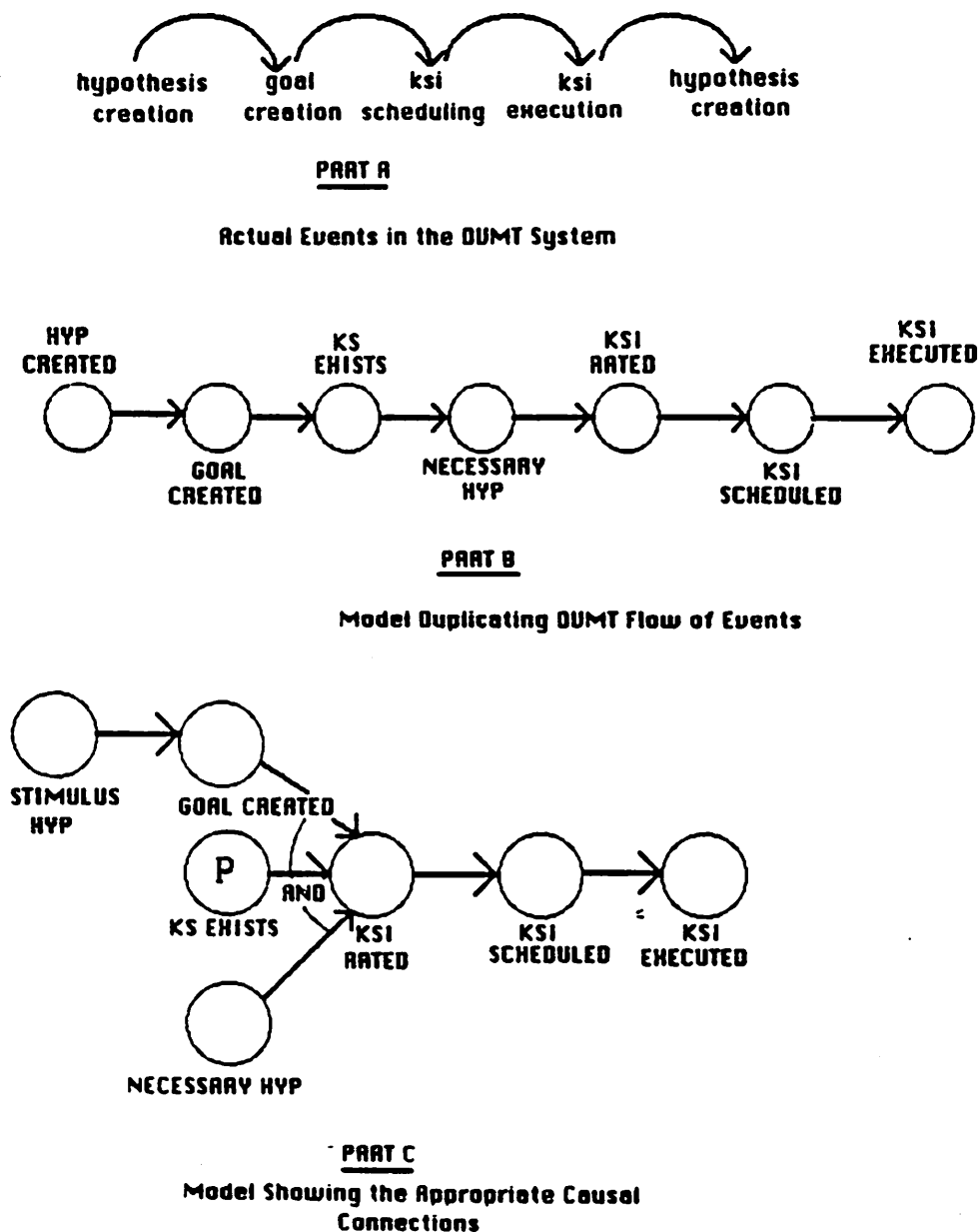


Figure 53: Construction of Causal Pathways

This figure shows two different ways of modeling system behavior. The behavior to model is in part A, showing the events following the creation of a hypothesis. We can represent the events exactly as they occur in the system, as shown in part B. This representation captures the timing of the events. However, this representation is not a causal model, since the linked states are not necessarily causally related. The model in part C however does represent the causal relationships among the states explicitly. Our approach to diagnosis relies on such causal representations.

place the states is akin to deciding which features will be visible at what degree of magnification. Deciding where in the model hierarchy to place the state involves balancing the following criteria.

1. We want to utilize our knowledge about what is the most likely cause of a situation. The states representing these primary causes will be higher in the model hierarchy.
2. We need to consider the ease of instantiation of a particular state. In other words, how difficult is it to determine the values of the state and its object from the system data structures or the surrounding states. Because the diagnosis begins at the higher levels of the model, the states at those levels will be instantiated most often and it is important that their instantiation be efficient.
3. We need to balance the ease of instantiation of a state with the amount of information that state provides about the system.
4. We want to organize the states so as to make the construction of the constraint expressions linking the objects as easy as possible by reducing the amount of contextual information required for evaluating the expressions.

The model builder must balance the above criteria. It would be interesting to construct several alternative models and experiment with their efficiency in diagnosing the system behavior. Ideally, the hierarchical structure should be determined automatically by the system, based on the above criteria. This way it could be changed as the system characteristics varied.

Finally, whether it is the construction of the selection of states to model, the construction of the causal pathways, or the organization of the states into

a hierarchy, the model builder must take into account how difficult it will be to construct and evaluate the constraint expressions linking the abstracted object attributes. The rule here is that **the more closely related the objects are, the closer they should be in the model.** This then makes it easier to capture those dependencies in the constraint expressions. This rule implies that states referring to the same object should be contiguous. (Of course, only if they are in the same cluster.)

§3. Choosing the Abstracted Objects

This section discusses the need for abstracted objects as separate entities in the SBM as well as how the different object types are selected and how their attributes are chosen. Abstracted objects in the SBM represent the corresponding objects in the DVMT system; i.e., the data structures representing the hypotheses, goals, and the knowledge sources. The SBM model must contain some representation of these objects in order to reason about the system behavior. The objects are represented as separate entities for efficiency reasons, to avoid the duplicate representation of similar sets of attributes. Since each state represents some aspect of the behavior of a DVMT object, that object must be represented somewhere in the system; either as part of the state or as a separate data structure. Since several states may need to refer to the same object, it is more convenient to represent that object as a separate entity rather than to repeat its representation in each state that refers to it.

Currently, the abstracted object types correspond closely to the object types in the DVMT system. In other words, the hypotheses, goals, and knowledge source instantiations have their corresponding abstracted object types in the SBM. There are three cases however, where this correspondence is not exact:

1. **When some object needs to be made explicit in the SBM that is not explicitly represented in the DVMT by its own data type.** An example of this is the representation of the signal in the environment by the object DATA-OB and the representation of the signals sensed by the individual sensors by the object SENSED-VALUE-OB. In the DVMT system the lowest level object representing a the data is a hypothesis at the signal location (sl) level. But problems may arise before the hypothesis representing some data is actually created and inserted onto the data black-board at the sl level. In order to reason about these problems we must be able to represent explicitly the intermediate objects involved, namely the original signal in the environment and the individual sensed signals, before they are combined into a single sl hypothesis. This is called **new object type creation**.
2. **When several object types in the DVMT system are combined into one object type in the SBM in order to make the model more concise.** This occurs when the structure and more importantly the behavior of two objects is similar enough to warrant combining these two object types into one. An example of this in the SBM is the combination of both the hypothesis and the goal objects into an object type MESSAGE-OB when these objects are being used to communicate among the nodes in the DVMT system. This is called **object type merging**.
3. **When a given object behaves differently enough in different contexts that it warrants the creation of two separate object types,**

one for each context. An example of this is the different behavior of local knowledge sources and communication knowledge sources when it comes to rating, scheduling, and execution of the knowledge source instantiation. In the case of local knowledge source, a stimulus hypothesis is necessary, which must create a goal, the goal then stimulates the creation and rating a knowledge source instantiation. In order to be instantiated, the knowledge source type must exist and the necessary data must also exist as hypothesis on the DVMT data blackboard. In the case of a communication knowledge source, the message to be communicated, whether it is a hypothesis or a goal, alone stimulates the instantiation of a knowledge source, assuming that knowledge source exists. No special goal is created and no necessary data is needed. We have therefore created a different data type for each of the two knowledge source types; the local knowledge source, represented by KSI-OB, and the communication knowledge sources, represented by COMM-KSI-OB. **This is called object type splitting.**

Even with the above differences among the DVMT and the SBM objects, the correspondence among the object types is still very close. This is not necessary however. We could represent what are currently object attributes as separate objects. We could therefore have the following data types: TIME-LOCATION-LIST-OB, EVENT-CLASS-OB, TIME-REGION-OB, or LEVEL-OB. We could also aggregate separate object types and represent a large class of these as new object type. This would allow the representation of the entire data blackboard, the entire goal blackboard, the various knowledge source instantiation queues, and the combinations of various hypotheses, goals, and knowledge source instantiations as separate object types. Having these explicitly represented in the SBM would then allow reasoning about the states of these objects. We could then rea-

son about the various loads of the queues, the states of these objects with respect to the ratings of their composite primitive objects, or with respect to the time or spatial distribution of these objects. Chapter X discusses some implications of allowing the representation of and reasoning about these objects. We believe that such representation would greatly increase the system's capability to reason about both low level domain knowledge and high level system states.

Having decided which data types to represent as separate abstracted object data types we must choose which of their characteristics to represent in the model. In other words, we must choose what the variable attributes should be. (Recall that the variable attributes of the abstracted objects represent the objects' characteristics. This is in contrast to the fixed attributes, which contain information necessary for instantiating the objects. See Chapter III for a more detailed description.) In cases where the data types are the same in both the DVMT and the SBM, we must select a subset of the DVMT object attributes. In the other cases we must either choose the attributes or add new ones. In the **exact correspondence case** and in the **object type splitting case**, we simply decide which aspects of the object behavior we need to reason about and then represent the attributes necessary. In the **object type merging case** we need to add new attributes to differentiate among the originally different types of objects. Thus for example the MESSAGE-OB has an attribute MESSAGE-TYPE which is either hypothesis or goal. We also need to merge some attribute names. For example, in the case of the MESSAGE-OB, the attribute representing the vehicle path is called time-location-list in the case of a hypothesis, and is called time-region-list,

in the case of a goal. In the MESSAGE-OB these attributes are merged into one, whose value is either time-location-list or time-region-list, depending on whether the message object is of type hypothesis or type goal.

§4. Constructing the Constraint Expressions Among the Objects

This section discusses the issues involved in the construction of the constraint expressions linking the abstracted object attributes. It is these constraint expressions that capture the correct behavior of the DVMT system. They are constructed by the model builder, based on her understanding of the system she is modeling. We can view these expressions as specifying the DVMT system behavior in a higher level language. We can thus view the construction of the constraint expressions as reprogramming the DVMT system in this higher level, declarative language. The syntax of this language is the syntax of the constraint expressions (described in Chapter VIII). The syntax allows a large number of ways of expressing the DVMT system behavior due to the system complexity and the large number of interdependencies among the DVMT system objects. This section discusses some rules that make this task a little more constrained. There are two general principles to follow here:

1. minimize the complexity of the constraint expression,
2. minimize the interaction among the abstracted objects.

Minimising the Constraint Expression Complexity. We want to minimize the constraint expression complexity both with respect to their construction

and with respect to their evaluation. The constraint expressions are expressed in terms of functions operating on arguments. The functions are constructed by the model builder. The arguments are obtained either from another state or object attribute, which has already been evaluated, or are expressed as another constraint expression which must be evaluated. We want to choose the simplest possible functions and we want to minimize both the number of arguments necessary and the ease of obtaining those arguments. There is a tradeoff between minimizing the number of arguments and the complexity of the functions operating on those arguments. In some cases an extra argument may save us a lot of computation. However, we have to consider how difficult it is to obtain that extra argument. We have tended to choose simpler functions when possible, at the expense of more arguments, because in general we are trying to minimize the procedural knowledge contained in the model. To summarize then, we want to choose the simplest functions possible, both from the point of view of their construction and from the point of view of their evaluation. We want to take advantage of using existing values where possible and inheriting them from other states or objects. (We must however balance this desire against the second principle, that of minimizing the interaction among the objects. This is discussed below.) When specifying the arguments, we again want to consider how difficult they are to specify and how difficult they are to obtain. The rule is very simple: Get the necessary information from an object as close as possible, including self. In other words, if we can evaluate some constraint expressions using some other attributes at the same object, use them instead of going to the attributes

of neighboring objects.

Minimizing the Object Interaction. Minimizing the interaction among the abstracted objects means that we want to construct the constraint expressions so as to rely as little as possible on the values of the attributes of other states and objects. In other words, if we can obtain the value of some object attribute using the attributes of the same object, choose this way over one which requires using the attributes of the neighboring object in the model. The reason behind this is the same as that motivating the modularity of programs: The more localized and minimal the interactions among modules, the easier it is to construct, debug, and change them. The same rule holds here. The less the evaluation of an abstracted object depends on the values of its neighboring objects, the easier it is change that object's definition if that becomes necessary as the model is used. This minimal interaction rule also affects the ease of evaluation of the constraint expressions. The further away in the model some value is, the more object or state attributes have to be accessed and the more expensive it is to evaluate the constraint expression.

In order to follow the minimal interaction principle, we must sometimes introduce a new attribute into an abstracted object. This attribute holds a value that is used repeatedly in the evaluation of the constraint expressions and would have to be either recomputed each time or accessed repeatedly via the neighboring object or state attributes. An example of this case is the addition of the goal-type attribute to the definition of the abstracted object GOAL-OB. In order to calculate the level, event-classes, and time-region-list attribute values, we must

have access to the attributes of the higher level hypothesis at the higher level of the model hierarchy. Rather than to continue accessing this, we access it once, calculate the goal-type attribute value, which then allows us the more efficient calculation of the remaining attribute values of the goal-object.

§5. Types of Abstraction Necessary to Reduce the DVMT System Complexity

Having discussed the principles we used in constructing each part of the system model, we would like to step back and describe the types of abstractions and the general methods for reducing complexity we found recurring in modeling and reasoning about the DVMT system. These methods fall into the following categories:

1. Parametrizing a group of objects in order to represent the entire group by one parametrized abstracted object (occurs during model construction).
2. Parametrizing groups of states in order to represent them by one state in the model (occurs during model construction).
3. Allowing the existing data to constrain the search during diagnosis (occurs during instantiation).
4. Selecting a representative from a group of related objects and reasoning about it (occurs during instantiation).
5. Grouping similar object together and reasoning about them as a group in order to reduce the search (occurs model during instantiation).

6. Abstracting the common characteristics of a group of objects in order to represent and reason about the group by one abstracted object, usually an underconstrained object (occurs during model instantiation).

These methods fall into two broad categories: those used in constructing the model, which we call **representation abstractions** and those used in instantiating and using the model, which we call **reasoning abstractions**. The first two methods in the list above are **representation abstractions** and the remaining ones are **reasoning abstractions**. This section describes and motivates each of these methods for reducing the complexity of the DVMT in order to represent and reason about it. Figure 54 illustrates these broad categories and their relationship to the DVMT system, the SBM, and the instantiated SBM.

Representing Groups of Objects as one Abstracted Object. Much of the DVMT system behavior varies depending on the system parameters and the data. One problem we had to face in modeling the system was the construction of an concise abstraction of the wide range of possible system behaviors as the data and the system parameters change. In our formalism this translates to constructing a concise representation of a variable number of objects in the DVMT system. We have solved this problem by grouping certain objects in the DVMT system into classes and representing the entire class as one abstracted object in the system model. This object is parametrized with respect to the data dependent attributes, whether they be the actual input data (the sensor signals) or whether they be system characteristics such as the signal grammar, the number of nodes in the system, or the number of knowledge sources in a node.

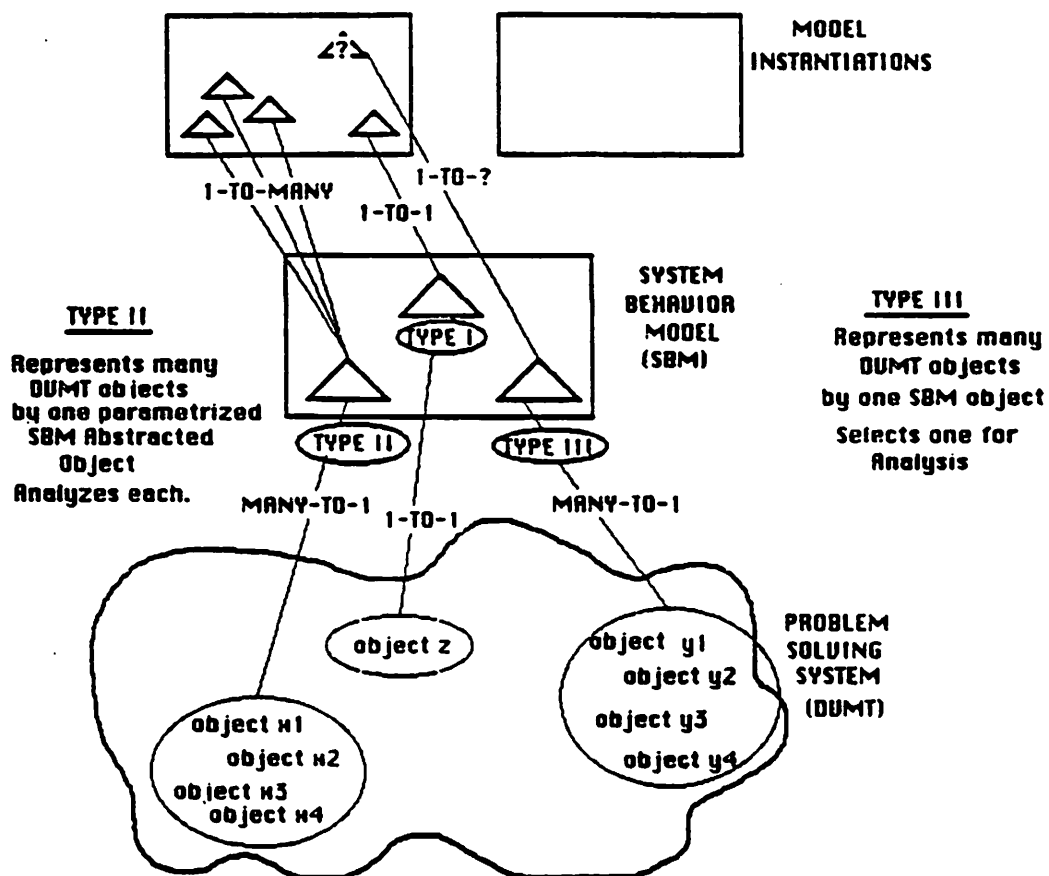


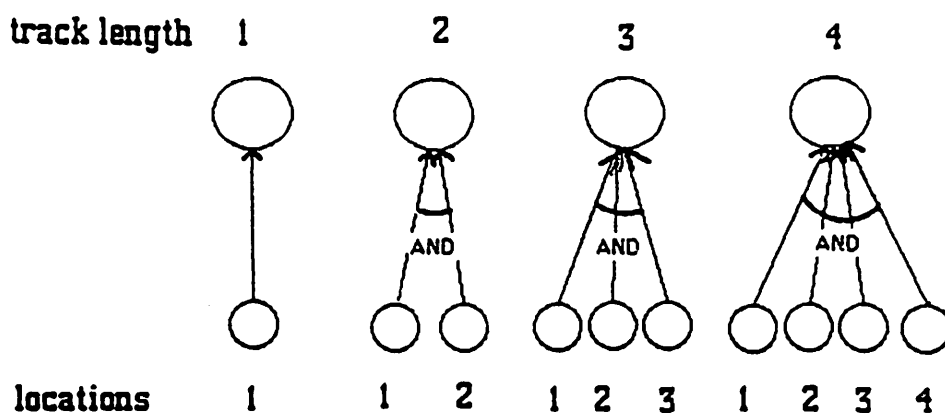
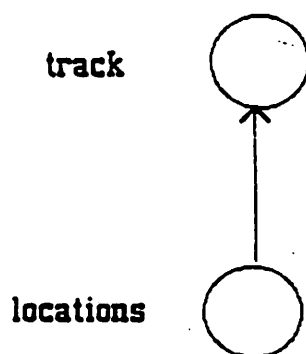
Figure 54: The Types of Abstractions Used in Model Construction and Reasoning

This figure shows the types of abstractions necessary to represent a complex system. The bottom part of the figure represents the DVMT. The middle part represents the uninstantiated SBM, and the upper part represents two instantiations of the SBM.

For example, consider the aggregation of locations into tracks. The model needs to be able to represent tracks of arbitrary lengths. One way to do this is to represent tracks of all possible lengths as separate objects and then represent their relationship to the locations. Figure 55, part A illustrates how this might be done. This is clearly not a very efficient way to represent the simple fact that we need n locations to form a track of length n . We therefore chose an alternative representation, shown in Figure 55, part B. In order to represent the location to track transformation for a general case, we model the group of locations as one state/object pair and a track of arbitrary length as another state/object pair. We are thus collapsing tracks of arbitrary length into one state/object and the variable number of locations into another state/object. When the exact number of locations becomes known during model instantiation, we create as many objects as necessary in order to represent each of the locations. In this way we can represent relationships among variable number of objects. There are three classes of data-dependent objects which need to be parametrized:

1. those depending on some system parameters such as the signal grammar or the number of nodes in the system;
2. those depending on the relationship among the object structures and the desired data (track to location parametrization);
3. those depending on the existing data (longer tracks to shorter track segments).

We have thus encoded a group of objects in the DVMT as one abstracted object in the SBM. Such parametrized objects require a special procedure when

PART A.PART B.**Figure 55: Representing a Class of Objects in the Uninstantiated Model**

This figure shows how we deal with representing many related objects. For example, when representing the many locations necessary to derive a track, we represent all locations by one parametrized object in the uninstantiated SBM. At instantiation time, when the number of locations is known, the actual number of objects is created. The object attribute that determines the actual number of objects created is called splitting attribute. Part A of the figure shows a non-parametrised way of representing this variable number of objects. Part B shows the parametrized method.

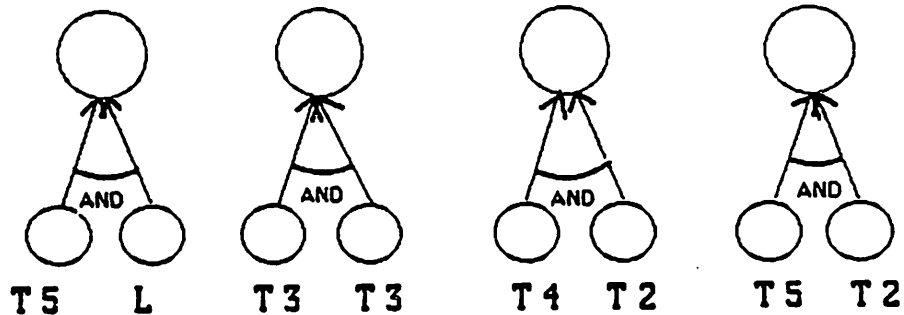
they are instantiated since one abstracted parametrized object may expand into a number of abstracted instantiated objects, corresponding to their actual number in the DVMT system. What we need here is to recognize when this should occur and know how many objects need to be created. In other words, we need to know the names of the object attributes which are data-dependent (and therefore parametrized) and whose values will determine the number of the actual instantiated objects. This information is contained in one of the control attributes in each abstracted object, called the **splitting-attributes**. This attribute contains the names of all the object's variable attributes that have been parametrized. During the object instantiation, when one of those attributes is evaluated, the number of values obtained determines the number of objects to instantiate. A separate object is then created and the parametrized attribute receives one of the values obtained initially. This causes a split in the branching of the graph, these attributes are called **splitting attributes**. Splitting attributes are thus attributes which have been parametrized for concise representation of objects in the system model. Examples of splitting attributes are **time-location-list** and **event-classes** of object hypothesis, **goal-type** of object goal, **ks-type** of object ksi, **sensor-id** of object sensed-value, and **nodes** of object message. This type of abstraction which is many-to-one during the model construction and one-to-many during the model instantiation is called type I in Figure 54.

Representing Groups of States as One State. In the above example we grouped a number of objects into a class and represented that class by one abstracted object. In the following example we will illustrate a similar technique,

but this time we will group together states and represent a number of possible series of events as one state in the model. This is necessary in cases where the number of the possible series of events is very large and we cannot represent all the possible series of steps of some process. A good candidate for this type of representation is the aggregation of shorter track segments and locations into longer tracks because the number of ways in which the shorter track segments and locations can be combined to form longer tracks is very large. The two choices here are shown in Figure 56. Notice that these states (the track states) point to themselves (the back neighbor of a track state is the state itself) and are therefore called **reflexive states**. During the model instantiation this state is expanded into as many states as necessary to represent how the longer track was derived from the shorter segments and individual locations. Reflexive states provide a mechanism for grouping a variable number of sequential states into one when representing the individual states explicitly would be too expensive.

Constraining the Search by Existing Data. In some cases the number of ways of deriving some object is too large unless we are constrained by the existing data. This is the case with track elongation. Suppose we are trying to analyze why a track of length 8 was not created. We could consider all the possible combinations of shorter track segments and locations and analyze why they were not created. In these cases not only would the combinatorics be prohibitive but it would not even be useful to explore all the possible derivation paths, since the system would not explore all of them either, being constrained by the data it has. We can reduce the number of pathways to explore by allowing the existing data

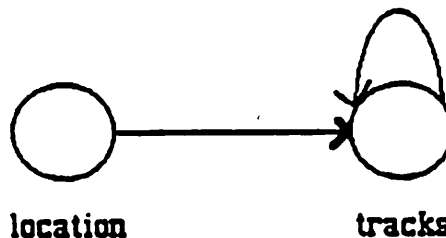
tracks of length 6



Legend

T n \longrightarrow track of length n
L \longrightarrow location

PART A.



PART B.

Figure 56: Representing a Series of Steps in the Uninstantiated Model

This figure shows how we concisely represent a series of related events in the DVMT. In this case, we need to represent the extension of a track into longer tracks. Rather than representing each possible track length in the uninstantiated model and then selecting the relevant ones at instantiation time (part A), we represent all tracks at one level by a parametrised track object (part B). At instantiation time as many of these objects are created as necessary, to represent the number of steps it actually took to create the current track.

to constrain the search during diagnosis. Rather than exploring all the possible ways of deriving some longer track we find the longest existing segment of the desired track and then analyze why it was not extended further.

Selecting a Representative Object from a Class of Objects. Another use of abstraction involves the selection of a representative object from a group of objects and analyzing its behavior rather than the behavior of each of the individual objects. In order for this to be effective, we must guarantee that the set of faults diagnosed when analyzing the representative object is the same as the set of faults diagnosed when each of the objects is analyzed separately. It turns out that track elongation satisfies this condition. In order to create a longer track, the DVMT system will aggregate shorter track segments and locations. Therefore at any time there will be a number of shorter track segments. In this case we choose the longest segment and analyze why it was not extended further. This does not reduce the number of faults we can identify because the undiagnosed shorter track segments fall into one of two categories. In one case the shorter, undiagnosed segments were later extended into a longer segment and there is no fault. In the other case the reason they were not extended is the same as the reason the longest track of the group was not extended. This fault will be identified by analyzing why the longest track segment was not extended. This is abstraction of type III in Figure 54.

Grouping Together Similarly Behaving Objects. In cases where a single object in the model is expanded into a number of objects in the instantiated model it may be possible to group some of these objects together and diagnose

each group as a unit. This is more efficient than creating a separate state for each of the objects and then repeating identical diagnostic paths with each of the states. In the initial stages of this project, every object created in the instantiated SBM had its corresponding state created and attached. During diagnosis then, each of these state-object pairs would be processed. This led to a combinatorics problem, even in relatively simple cases. Consider the diagnosis of the following situation. A PT hypothesis ranging from time 1 through time 8 is missing. In order to diagnose it, the system instantiates the SBM and traces the problem to the lack of the necessary data for this hypothesis. Because the hypothesis has 8 time locations, there are 8 locations missing. That means 8 objects and 8 states at every level, PL, VL, GL, SL, and SENSED-VALUE. Not only that, but because the system signal grammar tree has a branching factor of 2 at each level (this factor varies depending on the grammar used), we actually have 2 vl hypotheses for each pl hypothesis, 2 gl hypotheses for each vl hypothesis, and 2 sl hypotheses for each gl hypothesis. Thus for each pl hypothesis we end up with 8 sl hypotheses. Since there were 8 pl hypotheses this adds up to the grand total of 64 objects and 64 states at the sl level. Figure 49, part A in Chapter V illustrates the instantiated SBM representing this situation.

Note that there are eight pathways to follow during diagnosis. Notice also, that they all reduce the same problem: missing data. It would be much more efficient if we could recognize that all these objects behave similarly and therefore require the same diagnosis and can be grouped and diagnosed together. In other words, all the objects behaving similarly can be grouped under one state. This

state is then the only one that needs to be followed during diagnosis because the predicate or value it represents is the same for all its associated objects. The resulting instantiated model is illustrated in part B of the figure. Note that here there is only one diagnostic path to follow through the model.

In order to achieve this more efficient diagnosis we need to add the following information to the model: we need to add a control attribute to each state specifying the criteria for grouping the objects associated with the state, and we need to add object-to-object links, since now we may not be able to determine which objects are directly related just by looking at their associated states. The attribute that contains the object-grouping criteria is called **object-lumping-specs**. The information is specified declaratively so it can be changed easily as the object grouping criteria change.

Currently, the grouping is performed by applying a function to some combination of the state or object attributes. The result of this function determines the number of equivalence classes for the objects. A separate state is then created for each of those equivalence classes and all the objects in that class are attached to that state. Figure 57 illustrates this iterative grouping process.

This object grouping greatly reduces the number of paths that need to be examined during diagnosis. There is no loss of resolution however, because as soon as the object behavior changes and requires a different diagnostic pathway, the system creates a separate state for it, as specified in the grouping criteria. An example of a criterion for grouping objects is the existence of a corresponding object in the DVMT. All the abstracted objects which do not have a corresponding

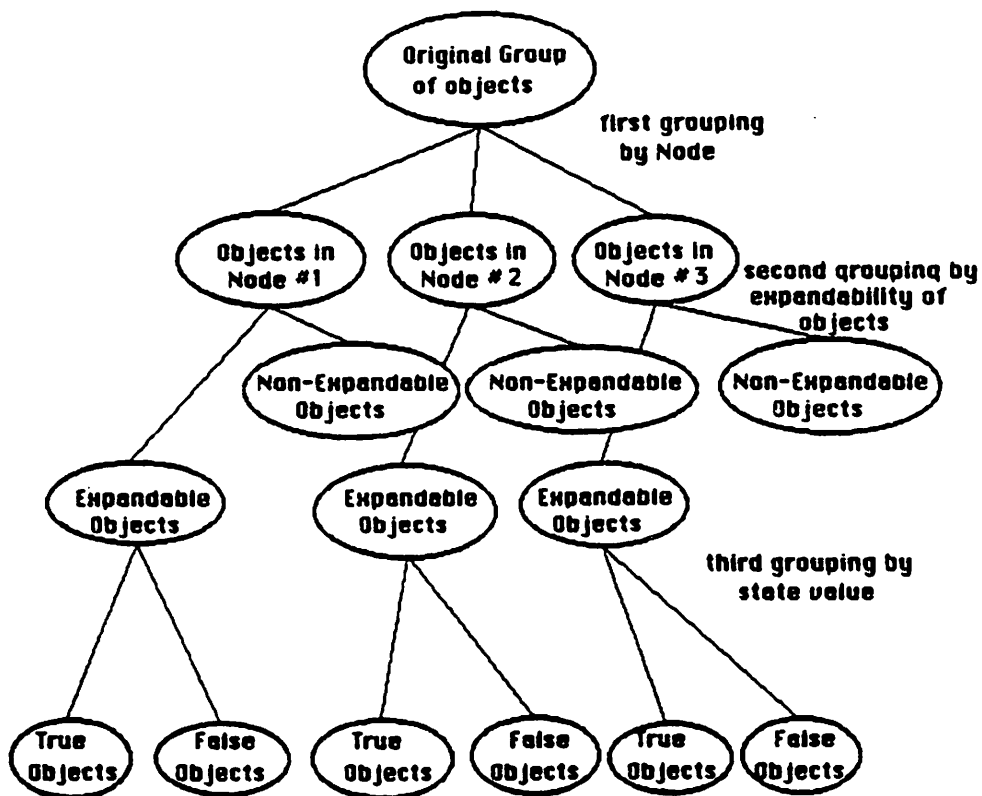


Figure 57: The Iterative Object Grouping Process

This figure shows how a group of objects is progressively divided into differentiated groups. For each leaf group of objects a separate state will be created.

object in the DVMT are grouped together under one state (which is false) and all those that do are grouped under another (which is true). This type of object grouping can only occur when dealing with parametrized objects; i.e., objects containing splitting attributes using type I abstraction.

Underconstrained Objects. Another way of using abstraction when reasoning about the system is to group together a number of objects and represent the whole class as one abstracted object in the instantiated model. This is done by underconstraining some of the attribute values of the abstracted object. Underconstrained objects are useful when it is known that a group of objects will behave identically and we can therefore save time by reasoning about the group as a whole. A situation where this is useful is the simulation of the effects of an identified fault. Suppose the system has identified a bad sensor in the DVMT system. It would be useful to propagate the effects of this sensor forward through the model and thereby not only explain any pending symptoms, which were caused by the same fault, but also account for future symptoms. In this case the simulation involves reasoning about the class of hypotheses generated from data in the area covered by the failed sensor. It is clearly more efficient to represent this whole class as one object rather than reasoning about all the possible hypotheses. Examples where this occurs is during the forward simulation of an identified fault. Chapter IV discusses this in detail.

This is different from grouping together objects and representing them as a class in the uninstantiated model, as described above because these are not for representation purposes, to be expanded later during model instantiation, but for

reasoning about a smaller number of objects by abstracting away the unessential characteristics. It is thus more closely related to the above described object grouping.

§6. Summary and Conclusions

This chapter outlined some issues in the construction of a problem-solving system model in our formalism. Specifically, we discussed:

- The rules to follow in the selection of the states and their organization into the model hierarchy;
- The selection of the abstracted objects and the construction of the constraint expressions linking these objects;
- The types of abstractions necessary in order to make if feasible to model a complex problem-solving system such as the DVMT;
- How the model is dependent on the DVMT system architecture and its domain.

Discussing the problems we have encountered and suggesting some solutions should make it easier for someone to construct a similar model for their domain.

It would be dishonest to suggest that the construction of the model paralleled this, hopefully systematic, presentation of the principles involved. Last but not least, we must share an observation that we made over and over while working on this problem. Construction of this type of a model is much like programming and system design in general. As much as we would like to provide an algorithm

for the model construction, it is still very much an art. There are dependencies among the various components. Often we would find that we had to go back and choose a different state or modify an existing abstracted object in order to simplify some constraint expression. We have had to alter the existing hierarchy in order to follow the "minimal interaction" rule for constraint expression construction. In other words, the hopefully systematic description of the model structure and construction presented here in no way parallels the more or less chaotic methods used in its construction. Much like programming, this process is neither top-down nor bottom-up but rather middle-out and side-to-side, until something workable is obtained.

And finally, a word should be said about the use of causal models in general. An unfortunate side-effect of the use of causal models is that often the misconception arises that because this model reasons from first principles, it can, by some sort of magic, do more than a traditional knowledge-based system, that it is by definition more robust and more knowledgeable. While it is often the case that systems using causal models are more robust and can perform more sophisticated reasoning, this is only due to the fact that the models they use provide them with such abilities, by giving them more knowledge. No magic takes place just because a system uses a causal model.

In the context of our work this means that the system will only be able to detect and diagnose the errors in the parts of the system represented in the model. It is therefore up to the model designer to make this choice and decide which errors the system will be able to detect and diagnose. While the use of

knowledge-base and specifically, a knowledge-base structured into a causal model, can make some errors easier to reason about, the system's ability is still limited by both the knowledge-base and the reasoning it can do.

Chapter VII

RELATED WORK

§1. Introduction

Ideally, in this chapter we would discuss different approaches to reasoning about problem-solving system behavior. We would compare them and explain how our work solves a different set of problems or how it solves similar problems in a different way. Unfortunately, we are not aware of any work that attempts to model and reason about a problem-solving system. We therefore look for related work from a different perspective. If we redefine what we mean by "related work", there is much research in AI that is relevant. We can find related work from three directions:

- **systems with related problems:** diagnosis,
- **systems working in related domains:** modeling problem-solving systems, and
- **systems using related methods:** causal models.

In some cases two of these criteria come together, as in the work of Genesereth and Davis [15,10], where the problem (diagnosis) and the method (causal model)

is the same but the domain (digital circuits) is different. Our work is in fact very different but a comparison is useful because it points out how the characteristics of the different domains effect the problem and the methods used for diagnosis.

We will concentrate on AI diagnostic systems which use causal models to represent their task domain. There are many such systems using various task domains: digital circuits and hardware (work of Davis, Genesereth, and Kelly [20]), large electronic systems (work of McDermott in Microwave Stimulus Interface [25]), nuclear reactors (work of Nelson [27]), and, of course, medicine (work of Weiss on CASNET ([39]) and Patil on ABEL ([30])). In this chapter we will describe some of these systems, emphasizing their relationship to our work. We will compare the causal models they use with the System Behavior Model (SBM), and their reasoning techniques with the types of reasoning described in Chapter IV. We will point out how the various model structures and reasoning methods are determined by the characteristics of the domains. For example, medicine is characterized by empirical knowledge and as a result medical diagnosis systems have to deal with uncertainty. The work in digital circuit diagnosis is characterized by the black-box view of their systems and as a result most of their effort is spent in devising testing techniques. Our work differs in both of these aspects:

- **We assume correct (but not necessarily complete) knowledge of the system we model and reason about.**
- **We assume that the internal structure of the system can be examined.**

We will come back to these points later in this chapter.

The systems we will describe here are:

CASNET: a medical diagnosis system, which diagnoses the causes of glaucoma, selecting the appropriate tests to administer as well as the therapy.

ABEL: another medical diagnosis system, which diagnosis acid-base related disorders.

DART: a hardware circuit diagnosis system, which diagnoses faults in combinatorial digital circuits.

DAVIS' circuit analyser: whose basic design is very similar to DART but which has significant extensions that allow it to diagnose a much wider set of errors.

We will first give a short description of each system and then discuss its features and compare them with those of the SBM.

Before we continue, one important point must be made. Diagnosis and detection are very closely related tasks, with many features in common. It is often difficult to say where one stops and the other begins. Even though the focus of our work is diagnosis, we feel that much of this work is equally relevant to detection. We delimit the two problems as follows:

DETECTION: Recognising inappropriate behavior in a system. Such recognition involves the use of redundant information which is necessary in order to provide independent standard for judging whether the behavior is correct or not. Regardless of the detection technique used, there must be a redundant source of information that provides this standard. The detection criteria may be in one of several forms. It may be in the form of the correct answer, in the form of replicated answers,¹ or

¹A common replication technique in hardware is Triple Module Redundancy or TMR.

in the form of some generalized expectations about the system behavior.

DIAGNOSIS: Explaining some detected misbehavior (a symptom) in terms of one or more predefined diagnostic categories. Diagnosis can be further broken down into determining likely explanations and confirming or denying them by means of testing. The exact problem-solving techniques used here depend on both the type of representation used for the problem and the emphasis of a particular system on one or another aspect of the overall problem (i.e., determining the suspects, generating the tests, or applying the test results to prune the suspect list).

We will point out which problem each system is trying to solve as part of its description. This will help us to think more clearly about the various characteristics of the systems and the models they use.

It is often very difficult in AI to separate the reasoning used to solve some problem from the formalism chosen to represent that problem. The result is that it is difficult to determine whether the researcher is struggling with a real problem, inherent in the task domain, or whether the problems arise from inappropriate choices of the representation formalism. As we describe and compare the diagnostic systems here, we will try to separate the aspects necessary due to some characteristic of the domain, from the ones that are merely artifacts of the chosen formalism.

§2. Medicine: CASNET

Description of the System. CASNET was developed by Weiss and Kulikowski at Rutgers to diagnose, evaluate, and treat patients with glaucoma [39]. CASNET performs both test and therapy selection, and diagnosis. It is the diagnostic knowledge that is represented by a causal model. (Detection, as defined above, is not addressed. It is up to the patient to come in with some symptoms. Correction (i.e., therapy selection) is done by table-lookup.) The causal model consists of states and directed arcs. The states represent selected abnormal situations involved in the mechanism of glaucoma development. Some states are designated as starting states (having no previous causes), others as final states (causing no other states). The arcs represent the causal links among these states. Each arc has an associated weight which encodes the probability that the two states it links are indeed causally related.

Disease processes may be characterized by pathways through the network. A complete pathway, from a starting to a terminal state, usually represents a complete disease process, while partial pathways, from starting to nonterminal states, represent various degrees of evolution within the disease process [39].

A state in the model has two values associated with it: status and weight. Both values represent evidence confirming or denying the situation represented by the state. The status is the result of tests, administered to the patient, designed to confirm or deny the specific condition represented by the state. A status can be confirmed (meaning that tests indicate the situation represented by the state has occurred), denied (meaning that tests indicate the situation does not exist),

or undetermined (meaning that no conclusive evidence exists either way). Status is thus the result of a local, direct measure of the situation.

The state's weight is determined from the causal-strength values associated with its incoming or outgoing links. The weight represents an "independent estimate of its likelihood that derives from the strength of causal associations between the state and its nearest confirmed or disconfirmed relatives [39]". The weight of a state is used to focus the diagnosis; to determine which states will be tested and which causal pathways will be followed. If the weight exceeds some threshold, the status of that state will be determined by applying the tests associated with the state. The weight captures an independent measure of likelihood of some state, based on the strength of its causal connections to the surrounding states.

There are thus two ways to determine whether some situation occurred or not: by direct tests (the state's status) and by evidence from surrounding states (the state's weight). It is this redundant information that allows CASNET to check its own consistency during diagnosis. If the two values agree, then the overall diagnosis is consistent and therefore probably correct, otherwise, either the tests, the diagnosis, or the model must be questioned.

Diagnosis begins with one or more states in the model confirmed, representing one or more abnormal situations. (Recall that these states represent abnormal situations, and a confirmed state therefore indicates a pathological condition. This is in contrast to the SBM, as well as some of the other systems described here, where the states represent normal situations, and where a confirmed state therefore indicates expected behavior.) Diagnosis continues by tracing back through

the model, deciding which pathways to follow by first selecting a subset of states for testing, based on their weight, and then applying the tests to the selected subset to confirm or deny them. Only the most promising pathways are followed (those whose weight and status values are high), and it is only when these do not lead to a consistent diagnosis that the weaker ones are tried. The disease category is determined by using a table that associates certain states with diseases. The therapy is selected from a table relating the diseases to the appropriate therapies.

Discussion. The CASNET formalism is similar to the SBM formalism in several important ways:

- The CASNET states are analogous to the SBM predicate states in that they represent a set of causally related situations. The fact that they represent an abnormal rather than a normal sequence of events does not detract from the formalism; presumably the states could just as easily represent correct sequence of states and the same reasoning mechanisms could still be utilized.
- Each state in CASNET has two values associated with it: the status and the weight. These are analogous to the SBM value and path value state attributes. The status (value in SBM) is based on local evidence; the results of tests. The weight (path value in SBM) is a global measure of evidence that the state is true or false, based on its surrounding states. In both cases the weight and status together, like the path value and value in SBM, provide the diagnostic system with two independent sources of information and are used to detect inconsistencies in the diagnosis, the tests, or the model.
- Both CASNET and the SBM support different types of reasoning. In CASNET, the focus is on diagnosis, although the authors mention that it can also be used to predict future course of an identified disease. This idea is the

same as the simulation of the effects of a fault on DVMT behavior (Forward Causal Tracing described in Chapter IV). The CASNET simulation is much simpler however because it does not deal with underconstrained objects.

There are important differences among the two systems. These result from both the nature of the task domain (medicine versus a problem-solving system) and the choices of the system designers on what type of reasoning to emphasize.

- CASNET is not structured as a hierarchical model and as a result does not reason at multiple levels of abstraction.
- CASNET can neither represent nor reason about classes of situations, as we can using underconstrained objects.
- CASNET can only represent what we call predicate states; situations whose outcome is true or false. Our system can, in addition to such states, represent and reason about relationship states. The fact that these states can represent relationships among objects in the DVMT system provides our Diagnosis Module with the capacity to compare different situations. This is useful because it allows the system to in effect choose a model for some behavior from within the situations in the system. Systems with such a capability would not need to rely on externally constructed models.
- A nice feature of CASNET is the representation of causal strengths on the state transition arcs. This allows the system to represent and reason about a situations with different levels of belief. This knowledge is then used for focusing the diagnosis. However, the reason for having these causal strengths is due to the fact that the knowledge represented in CASNET is uncertain. The formalism must therefore provide a way of encoding the belief that certain states are in fact causally related. We have not provided any such feature in the SBM since there is no question of probable causes

or consequences in the system we model, the DVMT. Our knowledge of the system we model is not uncertain, due to direct inspection of the internal states, and we therefore do not need to represent probabilistic strengths with the state relationships.

- **CASNET**, like other medical systems, pays much attention to testing: deciding which tests to apply, when, and how to interpret the results. This is necessary because testing of patients is expensive, potentially harmful, and, again, both the application and the interpretation result in more uncertain knowledge. Our Diagnosis Module on the other hand does not spend much energy devising complex and optimal testing strategies for the following reasons:
 - Testing is cheap; no risk to patient, no cost.
 - Almost an arbitrarily detailed view of the system is available to us, unlike most of the other diagnostic systems described here, which are forced to take the black-box view of their systems.
 - We know that the results of the tests we apply are correct. This is in contrast to medicine, where there is much uncertainty in interpretation of the test results.

§3. Medicine: ABEL

ABEL was developed by Patil at MIT to diagnose acid-base related disorders [30]. One of the major design goals of ABEL was the ability to approximate more closely the reasoning of physicians by reasoning at multiple levels of detail. ABEL's causal model, unlike that of CASNET, is therefore structured as a hierarchy. At each level of the hierarchy, ABEL's causal model, like the SBM and

CASNET, is represented by a state transition diagram. The states represent either pathological states or some statement about a physiological situation (ABEL mixes pathological states with normal states in the model). The state transition arcs in ABEL are objects themselves due to the complexity of information they represent. Namely, relationships among attributes of neighboring states. Both the states and the arcs are structured as a hierarchy. States expand into other states at lower levels and links expand into series of links at the lower levels.

The ABEL model hierarchy has several levels. At the lowest level of the model hierarchy are the patho-physiological states, and their links, describing the acid-base mechanisms in the body. There are various intermediate levels and the highest level is syndromic knowledge, which relates symptoms at the clinical level to one another. States can be primitive, if they have no substructure. That is, if they cannot be defined in terms of more primitive states at a lower level. States are called composite when they can be expressed in terms of lower level states. Links follow the same structure: a primitive link cannot be expanded into a series of links, a composite link can.

ABEL begins by instantiating a portion of the model in order to organize some set of findings into a consistent set of states and links in the model. This instantiated model is called a Patient Specific Model or PSM. Since there will often be some inconsistent findings, ABEL can instantiate several PSM's, one for each competing disease hypothesis. ABEL uses several operators in the construction of PSM's. These are:

Initial Formulation: creates an initial set of states from the patient's com-

plaints and preliminary laboratory findings.

Aggregation: summarizes a description at one level of the hierarchy by a single state or link at another level of the hierarchy.

Elaboration: is the opposite of aggregation and breaks down a higher level state or link into its composite ones at a lower level.

Projection: projects the causes or effects of a particular state or link along the same level of the hierarchy in order to explain where a state came from or its future course (this is related to Backward and Forward Causal Tracing described in Chapter IV).

In order to construct a PSM, ABEL first associates a set of clinical disease categories with the original findings. Once the initial diseases are postulated, ABEL begins to construct the PSM, using all four of the operators described above. This involves determining all states and links that can be concluded at all the possible levels of detail but using only the initially known data.

Once a PSM is established, ABEL spends considerable effort in devising tests to discriminate among competing possibilities. In fact, most of ABEL's problem-solving seems to be in the planning of the most optimal tests to administer to the patient. At the end of the PSM building operations, ABEL may be left with a number of competing hypotheses. It must then make choice among these in order to select the appropriate therapy. ABEL uses several strategies for selecting among the hypotheses. These strategies are based on the relationship among the hypotheses beliefs as well as the number of competing hypotheses. For example, if a hypothesis in a group has a belief that is much lower than the others, then it will be discarded.

Discussion. Like CASNET, ABEL has many similarities to our work but the problem it addresses is really different. Its main focus is on devising optimal testing strategies and on explanation of the result in a manner that would be satisfactory to physicians.

- The ABEL states, like CASNET states, are analogous to the predicate states in the SBM.
- Although ABEL has many states that represent situations where some component is "lower than normal" or "higher than normal", ABEL does not make use of an equivalent of our relationship states, nor does it use qualitative reasoning. It seems that using qualitative reasoning would make much of ABEL's work simpler. It would reduce the number of states that needs to be represented by allowing the parametrization of the type of relationship some value had to its normal value. For example, rather than requiring three states to represent the three relationships: lower-than, normal, and higher-than, one state which would name the relationship type would be sufficient.
- ABEL mixes normal (physiological) and abnormal (pathological) states in a single representation. This is fine for their purposes, but this type of inconsistent representation might cause problems if ABEL were to expand its set of reasoning types to include simulation, for example, simulating the effects of therapies.
- The links in ABEL are represented by objects because they contain complex information; the relationship among the attributes of the neighboring states. Although SBM does not even represent links as separate objects, it seems that the set of constraint relationships among the abstracted objects is analogous to the link object in ABEL, in terms of the information it contains.

- The ABEL hierarchy has some similarities with the SBM hierarchy but is not identical. The similarity lies in the links. The SBM hierarchy is in terms of links: lower level clusters can be viewed as expansions of higher level links. A high level link in SBM, for example, the link connecting VT to PT states, is expanded at a lower level model into the groups of links and states STIMULUS-HYP, GOAL-CREATED, KSI-RATED, KSI-SCHEDULED, and KSI-EXECUTED. ABEL has a similar ability. For example, a high level link between diarrhea and dehydration is expanded at a lower level into diarrhea, lower GI loss, sodium loss and water loss, and finally dehydration. However, ABEL can also expand states into their component states at the lower levels of the model hierarchy. ABEL does this in order to generate fine probes that will help establish various states at higher levels. Since testing is not an issue for us, we do not need this type of finer substructure of the model states. There is no state substructure in the SBM. Having such substructure would mean, for example, that objects could be expanded into their component objects at lower levels. That is, a hypothesis would be expanded into a level, an event class, and a time location list. It is not clear what exactly such a capability would do for the Diagnosis Module.
- The ABEL formalism supports several types of reasoning. The *projection* mechanism is analogous to the Backward and Forward Causal Tracing in the Diagnosis Module. Of the four mechanisms ABEL uses to construct a PSM, our system uses *projection*. We achieve the same effect as aggregation and elaboration by having projection across different levels of the hierarchy. In other words, where ABEL moves across the levels by aggregation and elaboration, we use a generalized projection mechanisms that can traverse the various model levels.

- ABEL has no reasoning corresponding to our comparative reasoning, nor can it deal with classes of objects.
- ABEL, like all medical diagnosis programs and unlike the Diagnosis Module described here, must deal with uncertain information. Much of the mechanisms in ABEL are a consequence of this. For example, the need for maintaining several competing hypotheses and then having strategies for deciding which of them is the correct one. This problem never occurs in the Diagnosis Module because everything is either confirmed or disconfirmed at each step of the diagnosis.
- ABEL's strategy for diagnosis is a function of the traditional diagnostic methods in medicine where the physician has to make sense of a collection of symptoms at various levels of detail provided by the patient, as well as a battery of standard preliminary tests. The problem here is that some of the symptoms may be irrelevant or distorted. The first step for ABEL is to construct a set of consistent hypotheses about the diseases in the patient. We do not have this problem because we start with ONE symptom and analyze it systematically by going backward or forward through the model. An analogous approach in our system would mean taking all the symptoms at once and integrating them into several competing instantiations (a la PSM's) of the SBM. We do not do this, because the DVMT system is not as complex, and we can therefore investigate each symptom individually. A similar end-effect is achieved however by using Forward Causal Tracing which accounts for any pending symptoms generated by the identified fault.
- It is not clear from the literature how ABEL formulates the initial disease hypotheses from the laboratory data. It seems that the model of pathophysiological processes should be used here, but this is not mentioned in the literature describing the system.

It seems that the aims of the two systems, ABEL and the Diagnosis Module, are very different. ABEL attempts to construct a consistent interpretation of a large set of disparate symptoms, using its multi-level, richly constrained model of the patho-physiological processes in acid-base disorders. Only in cases where confirmation is necessary does ABEL resort to testing of a particular state in the model. Most of the work in ABEL is done in determining which states to test in order to obtain the maximal amount of differentiating information. The system can be thought of as working in a batch mode and only occasionally interacting with the environment by obtaining data about the patient.

In contrast, the Diagnosis Module described here operates in a highly interactive manner. It simulates the correct behavior of the DVMT system at various levels of abstraction and at each step in the diagnosis it checks to see whether that step has been taken by the system. This is possible because the DVMT system is much less complex than the human body, because it is not important to minimize testing, and because the DVMT internal states are accessible.

§4. Digital Circuits: DART

DART was developed at Stanford by Genesereth to diagnose faults in digital circuits [15]. "The program accepts a statement of a system malfunction in a formal language, suggests tests and accepts the results, and ultimately pinpoints the components responsible for the failure". DART uses a hierarchical model of the circuit in order to focus the diagnosis.

DART uses a causal model (i.e., a model of the circuit's correct structure and function) to diagnose the faults. The model represents both the structure of the circuit and its intended behavior. (Contrast this with the previous systems, where no separation was made between structure and functions. In medicine such separation might mean representing individual organs (structure) and their behavior in response to some events (function).)

The model is hierarchical so that a single component at one level may consist of a number of subcomponents at a lower level. Both the structure and the behavior are represented in a predicate calculus-like language. The structure is represented by specifying the components, their inputs and outputs, and the interconnections among them. For example, component x , an adder with two inputs, would be represented by the statement:

$$\text{ADDER}(x) \text{ AND } \text{IN1}(x)=a \text{ AND } \text{IN2}(x)=b$$

Behavior is represented by rules that express not only the type and behavior of the device but also a statement that the device functions correctly. For example, the rule describing the behavior of an adder is:

$$\text{ADDER}(x) \text{ AND } \text{OK}(x) \text{ AND } \text{IN1}(x)=a \text{ AND } \text{IN2}(x)=b \implies \text{OUT}(x)=a+b.$$

DART makes the usual assumptions made in hardware FT about the failures: only a single point of failure exists and the failure is a permanent one. However, these are supposedly only made for efficiency and are not necessary in order for DART to work. It is not clear how DART deals with the problems which arise when these assumptions are relaxed.

DART begins diagnosis with a symptom, which is expressed as a violation of some expected output. It continues by first generating all suspect fault candidates and then eliminating all but one. If this is a primitive component, diagnosis stops, otherwise the faulty component's subcomponents are analyzed. The initial set of suspects is determined from the representation of the circuit structure. For example, if the output of some adder is not what we expected, then the suspects will be the adder and both of the adder's inputs. If the inputs are themselves the result of some previous operation, then those components and their inputs are substituted for the original inputs. This process continues until a statement is obtained that contains only component candidates (as opposed to values at the input or output ports of those components). This set of candidates is then pruned by generating tests that discriminate among the suspects. These tests are generated by DART using its representation of the circuit's behavior. The reasoning involved is analogous to the Backward and Forward Causal Tracing described here. Since only the inputs and the outputs of the circuit can be observed, the tests are expressed in terms of inputs and some desired outputs. The test consists of applying the inputs and observing the outputs. If the outputs are as expected, based on the behavior model of the circuit, then the component being tested is functioning and can be eliminated from the suspect components set.

Discussion. The main focus of DART is in generating the tests that would discriminate among the set of suspect components. In order to generate these discriminatory tests, DART does reasoning analogous to Forward and Backward

Causal Tracing. It uses Forward Causal Tracing to determine what the results of some inputs should be if the devices are behaving correctly. It uses Backward Causal Tracing to compute the original set of suspects from the initial symptom and thereafter anytime an input needs to be defined the component that generated and its inputs.

What DART does in order to determine whether some component failed is similar to Unknown Value Derivation. In the Diagnosis Module, this type of reasoning is the exception; a last resort when the necessary values cannot be determined by direct probes (look-ups in the DVMT data structures). That is the only time that we have to treat the DVMT as a black box. DART on the other hand treats its circuits as a black box all the time. The availability of the intermediate problem-solving states allowed us to get beyond the type of reasoning necessary in systems like DART. These systems solve the testing problem. We have avoided dealing with this problem by assuming availability of the intermediate system state.

Much of DART's success relies on multiple uses of some output. It is this type of redundancy that allows the elimination of suspect candidates. It is not clear how DART deals with cases where no such redundancy exists.

The work in DART is in generating the discriminatory tests. Representing the system seems easy and so does the simulation. In our case much of complexity comes from the representation issues and the simulation part that is difficult because we have modeled a more complex system.

§5. Hardware: Davis's circuit analyzer

The basics of Davis's system are almost identical to DART, both in problems it is trying to solve and in the methods used. The representation formalisms differ slightly (DART is based on predicate calculus and resolution whereas Davis uses constraint networks) but this does not seem to make any difference with respect to performance. Where Davis's system differs is in its use of explicit fault models to help in troubleshooting when no consistent diagnosis can be obtained. In this way his system can handle problems without making assumptions such as the single, permanent failure. His system does this by explicitly representing the assumptions about the circuit, and then relaxing them one at a time (in some predefined order) and seeing what new causal pathways this exposes.

§6. Summary of Comparisons of AI Diagnostic Systems

All the diagnosis systems discussed here use some form of hypothesize and test problem-solving. They all begin with a symptom which allows the initial generation of suspected faults. This set of suspects is then pruned using various testing techniques. It seems therefore that this at least is a common feature of all diagnostic problems.

The systems differ however in how they approach each of the stages in the diagnosis process and where the bulk of the problem-solving lies. These differences are due both to the nature of the domain and to the type of assumptions made by the designers. Both Genesereth and Davis assume the system is a black box.

In other words, only the inputs and the outputs can be observed and the faulty components must be established by indirect measures. Both rely on redundancy to eliminate the suspects. The focus of these systems, as well as ABEL, is in devising appropriate discriminatory tests. Our approach to testing was determined by the uniqueness of the DVMT in that it maintains the history of its problem solving. Intermediate results are retained until the solution is formed and the testing problem is therefore trivial. (The potential problem is that in a realistic system we may not be able to store indefinitely all the details of intermediate processing. But we could still deal with that by storing certain key points and using the model to derive the rest.) Not having to worry about testing has allowed us to concentrate on other types of reasoning such as reasoning about classes of objects and comparative reasoning.

Two important factors distinguish the medical diagnosis programs from the others, including the Diagnosis Module described here. They must minimize testing and they must deal with uncertain information. Many features of these models are a function of these characteristics of the medical domain. For example, the causal strength on links in CASNET and a different type of a link, the association link, in ABEL. It is due to the uncertainty inherent in their domain that many of the medical programs worry about integrating categorical and probabilistic reasoning. This does not seem necessary when diagnosing systems whose behavior is, at least in principle, completely known.

In regards to representation, the main difference between the medical and the hardware diagnosis systems is in the separation of the structure from the

behavior. In the medical systems, no attempt is made to represent the structure of the organism whereas in the hardware systems the structure is represented separately from the behavior. Another difference is in the structure of the model hierarchies. In DART, the hierarchy is both in terms of the structure and in terms of the behavior. In the SBM, the hierarchy is only in terms of the behavior (events are hidden at the higher levels of the hierarchy). In ABEL, the hierarchy is structured so as to make certain states explicit in order to define the disease for therapy purpose or to select the appropriate tests. The hierarchy in ABEL, unlike that in DART, seems to be less systematic and includes many pre-compiled rules about expected behavior pathways.

The systems differ in how they detect and handle inconsistencies. In both CASNET and ABEL, an ordered set of possible disease hypotheses exist. The system tries to establish the highest rated hypothesis and if this fails, i.e., if the diagnosis is inconsistent, it moves to the next one. In Davis's system inconsistent diagnoses are handled by the explicit representation of the assumptions about the circuit behavior and the gradual relaxation of these assumptions. In our case, the Diagnosis Module does detect inconsistent diagnoses but we have made no attempt to resolve them.

Chapter VIII

IMPLEMENTATION DETAILS

This chapter describes the implementation details of the Diagnosis Module (DM). The details of the model structure are discussed first; including the representation of the abstracted objects and states, and the structure of the constraint expressions linking the abstracted objects. The chapter concludes with a detailed description of the model instantiation.

§1. Introduction

Both the states and the abstracted objects are represented by record structures (frames), implemented as association lists. The state transition arcs are not represented explicitly as separate objects. Instead, each state has a set of neighbor attributes which specify that state's neighboring states as well as the relationship (AND, OR, or PrefOR) among these neighbor states. The state and abstracted object attributes fall into three categories:

DEFINING ATTRIBUTES: contain the information related to the objects or events in the DVMT system. In the uninstantiated System Behavior

Model (SBM) these attributes contain the expressions that allow the evaluation of the particular attribute. In the instantiated model (ISBM) these attributes contain the results of the evaluated expressions and thus define a particular situation or object in the DVMT system.

CONTROL ATTRIBUTES: contain information necessary to instantiate the state or object, such as the order of the attribute evaluation. Control attributes are empty in the instantiated model.

STRUCTURE ATTRIBUTES: contain information about the model structure, such as links among states, objects, or state-object links. In the SBM these attributes contain the names of the neighboring states or the name of the abstracted object associated with the state. In the instantiated model these attributes point to the actual instantiated neighbors or objects.

This chapter is organized as follows. First, the structure of the abstracted objects is described, followed by the structure of the states. The neighbor lists, which describe the relationships among neighboring states, are described next. Finally, the structure of the constraint expressions linking the abstracted objects is discussed. The last section of this chapter integrates all this information by giving a detailed example of how the SBM is instantiated to represent a specific situation in the DVMT. As before, we will follow the convention of capitalizing all state names and referring to DVMT objects in lower case letters. Thus "VT" represents the state VT in the System Behavior Model, whereas "vt" represents a vehicle track hypothesis in the DVMT.

§2. Abstracted Objects

Abstracted objects represent the actual objects in the DVMT system. There is a number of different types of abstracted objects, each corresponding to an object type in the DVMT system. Some of these object types are represented explicitly in the DVMT system, for example hypotheses (hyps), goals, or knowledge source instantiations (ksis). Others are not explicitly represented in the DVMT system but we have found it useful to represent them as separate object types in the SBM. Examples of these objects are sensor, channel, or data (meaning the input sensor data). Chapter VI discusses in detail the criteria for choosing the different abstracted object types. This section presents a detailed description of the abstracted objects' attributes and their use.

In addition to the three attribute categories described above, the object attributes can be classified into two other categories: the **variable attributes**, which are different for each object type, and the **fixed attributes**, which are the same for every abstracted object type. Note that we are speaking here of the attribute types, not their values. Therefore fixed attributes implies the same attributes in each object, but with different values. The variable attributes are the object's defining attributes. For example, the object hypothesis contains attributes necessary to describe a hypothesis in the DVMT system, such as its level, event class, time location list, and the node it belongs to. Since each object in the DVMT system has different types of characteristics, each SBM abstracted object (AO) must have different types of defining attributes. The variable attributes are

a subset of the corresponding DVMT object attributes. They are only a subset of the actual DVMT object attributes because we have chosen not to model every aspect of the DVMT objects.

In the SBM, the attributes contain the constraint expressions that are evaluated to the actual attribute value when the model is instantiated. In the ISBM, the attributes contain the evaluated constraint expressions, i.e., the values of the actual object characteristics. These constraint expressions will be described in more detail in the next section.

The fixed attributes are the same for each type of AO and contain the control and the structure attributes. The fixed object attributes contain information necessary to instantiate the abstracted objects; i.e., to evaluate their variable attributes with respect to some set of objects in the DVMT. Since the fixed object attributes contain either information regarding the model instantiation or the model structure, they have no equivalent in the DVMT object attribute list. The rest of this section is devoted to the detailed discussion of the fixed abstracted object attributes. Figure 58 shows an example of a DVMT object and its corresponding abstracted object in the system model, illustrating the relationship among the defining AO attributes and the corresponding DVMT object attribute, and the relationship among the defining, control, and structure attributes. Figure 59 lists the names of the fixed object attributes, and Figure 60 lists the names of the variable attributes for three abstracted object types: hyp (HYP-OB), goal (GOAL-OB), and a ksi (KSI-OB).

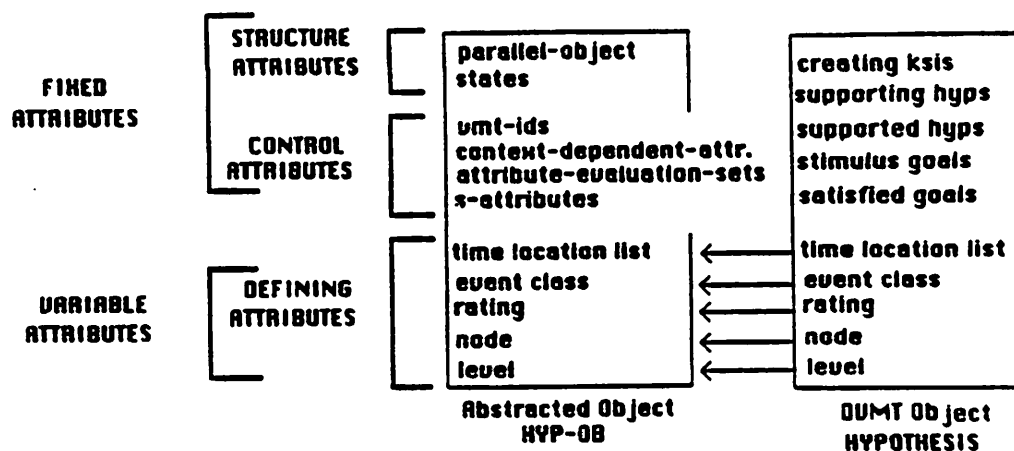


Figure 58: DVMT Object and its Abstracted Object Representation
 The structure of the DVMT object hypothesis and its abstracted object representation in the SBM. Note that not all of the DVMT object's attributes are represented in the SBM and that there are additional attributes in the SBM. These attributes control the object instantiation during diagnosis.

FIXED ABSTRACTED OBJECT ATTRIBUTES

name	name of object
aux object attributes	names of variable attributes which point to other abstracted objects
states	names of states this object is attached to
vmt-ids	names of corresponding DVMT objects
expandable-state-function	determines whether the object's existence is consistent with system parameters
s-attributes	controls cases where several object need to be instantiated from one parametrized object description in the SBM
fl-attributes	contains names of attributes that should be set when propagating forward the effects of an identified failure.
fl-overlapping-objects	links fully constrained objects to their overlapping underconstrained objects and vice versa
attribute-evaluation-sets	specifies order of attribute evaluation
context-dependent-attributes	lists names of context dependent attributes

Figure 59: A List of the Abstracted Object Fixed Attributes

Creating Multiple Object Instantiations. Recall that in many cases one object in the SBM may be instantiated into several objects in the ISBM. This occurs when the abstracted object is parametrized with respect to some dynamic aspect of the DVMT system. It therefore represents a class of objects and the exact number of the objects in that class depends on either data or some parameters in the DVMT. In order to instantiate the correct number of objects the DM must know which of the variable attributes of each object require the creation of multiple objects. Such attributes are called **splitting attributes** because they cause a one-to-many branch in the links connecting the abstracted objects. Their names are listed in the fixed object attribute called **s-attributes**. The **splitting attributes** work in the following way. When a variable attribute is evaluated, the DM checks to see whether it is a splitting attribute. If it is, then a separate object is created for each of the elements in the list of values produced by evaluating the constraint expression in that attribute. For example, when the DM instantiates the location hypotheses necessary to construct some track hypothesis, there will be more than one location object for each track object (since a number of locations is required to make up a track). The exact number of the location objects will be a function of the track length. The location object's **time-location-list** attribute is therefore a splitting attribute.

For example, suppose we are instantiating the pattern location hypotheses (PL-HYP-OB'S) necessary to derive a pattern track hypothesis (PT-HYP-OB) that has a time location list of length 8. When the time location list of the PL-HYP-OB is evaluated, the result is the 8 time locations which are necessary

VARIABLE ATTRIBUTES FOR THREE ABSTRACTED OBJECT TYPES

HYP-OB	GOAL-OB	KSI-OB
time-location-list	active-trl	stimulus-goals
	inactive-trl	output-hyps
event-class	event-classes	
rating	rating	rating
level	level	
node	node	node
	stimulus-hyps	stimulus-hyps
	goal-type	ks-type

Figure 60: Variable Object Attributes

A list of the abstracted object variable attributes for objects representing hypotheses, goals, and ksi's.

to make up the track. If this were a normal attribute, then its value would be that list of eight time locations. However, because this attribute is a splitting attribute, a separate PL-HYP-OB will be created for each location, resulting in eight PL-HYP-OB abstracted objects.

Order of Attribute Evaluation. In some cases there may be several splitting object attributes. Since the evaluation of one may depend on knowing the value of another, we must make sure that they are evaluated in the correct order. This is the function of the **attribute-evaluation-sets** attribute which contains groups of lists specifying the order of the variable attribute evaluation. The attributes in any one group can be evaluated in any order but the attributes in the n th group must be all known before any of the attributes in the $(n+1)$ th group can be evaluated.

Merging of Objects. We have discussed the splitting of objects when more than one object is necessary for the construction of another. What happens when the reasoning proceeds in the other direction and therefore requires the inverse of splitting, or merging? For example, this happens when a group of locations is to be merged into one track. We do not specify explicitly the merging criteria as part of each object. Instead, we let the DM's mechanism for detecting and merging identical objects take care of this case. Every time a new object is instantiated, the set of existing objects is checked to see if an identical one already exists. If it does, the new one is not created and instead the existing one is used. In this way diagnoses for separate symptoms are merged.

Linking the Abstracted Objects with their DVMT Instances. Each abstracted object defines one or more objects in the DVMT system. Part of the instantiation process is to determine whether such objects actually exist in the DVMT. The function of the `vmt-ids` attribute is to point to these objects. This attribute contains the names of the actual objects in the DVMT system which correspond to the abstracted object. It is this attribute that is used to evaluate the state value attribute of predicate states. If no actual object exists in the DVMT, then the value will be false, otherwise it will be true.

Use of Context. Some of the variable attribute constraint expressions are context dependent. This means that different constraint expression must be used to evaluate these attributes depending on the way the model is being instantiated. We want to be able to instantiate an object given the attribute values of either of its neighboring objects. Since the constraint expressions will be different depending on which object's values are known, all possible relationships must be specified in the abstracted object. The names of the context dependent attributes are contained in the `context-dependent-attribute` in each abstracted object. These, in addition to the type of object whose values are known, are then used to select the appropriate constraint expression for the variable attribute evaluation.

§3. Constraint Expressions

The constraint expressions represent the dependencies among the actual objects in the DVMT system by representing the relationships among the attributes

of the abstracted objects. It is through the evaluation of these constraint expressions that the expected behavior of the DVMT system is simulated and it is this simulation that is at the heart of our approach to reasoning about the DVMT system behavior. In the uninstantiated SBM, the defining (i.e., the variable attributes) of the abstracted objects contain constraint expressions. These expressions are evaluated at instantiation time and the resulting values are placed in the corresponding attributes in instantiated abstracted objects.

The constraint expressions are represented by functions which operate on arguments. We can characterize the constraint expressions by the type of function and the type of arguments the function operates on. The arguments may be constants, be the values of some already evaluated attribute (either of the same object or at some other abstracted object in the instantiated portion of the model), or other complex constraint expressions which need to be evaluated. In most cases the arguments are values of some already evaluated object or state attributes. These may be the attributes of the same object or they may be the attributes of some neighboring object. Most often this is the object instantiated immediately prior to the current one. Occasionally an object needs to be specified which is not a neighboring object but can be specified by the path to follow to get to it. A path expression defines a unique attribute by describing how to get to that attribute by traversing several states or objects. Every path expression is of the form

(PATH <name of state or object> <attribute name>)

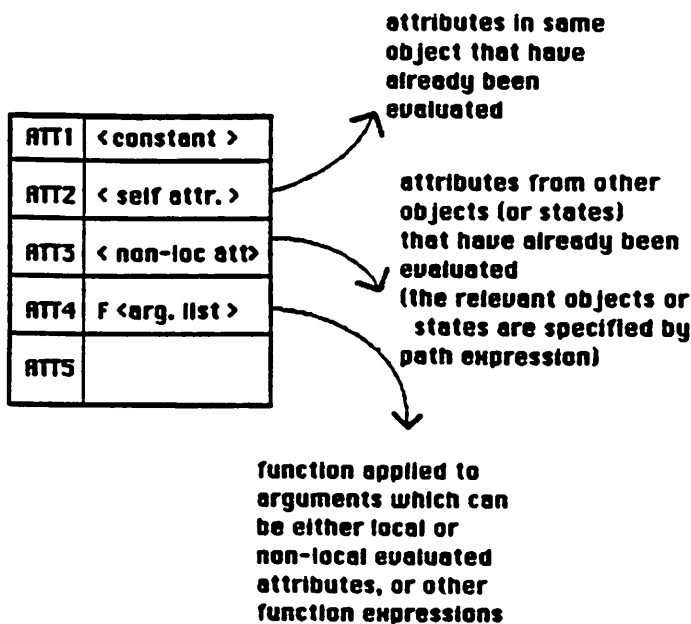


Figure 62: The Types of Dependencies Among Attributes
Different ways of obtaining the attribute values for an abstracted object. An object's attributes may depend on the attributes of neighboring objects or on its own attribute values.

(PATH (PATH self object-ptrs))

states. When evaluated, this path expression returns the names of all states attached to the abstracted object B.

The functions used to calculate the values of the variable attributes range from very simple table-lookup functions to quite complex transformations duplicating the functionality of the DVMT system. An example of a simple table-lookup function is the determination of event classes for one level of the data given the event classes at another level. An example of a more complex function is the determination of the type of goal to create for some stimulus hypothesis and some output hypothesis.

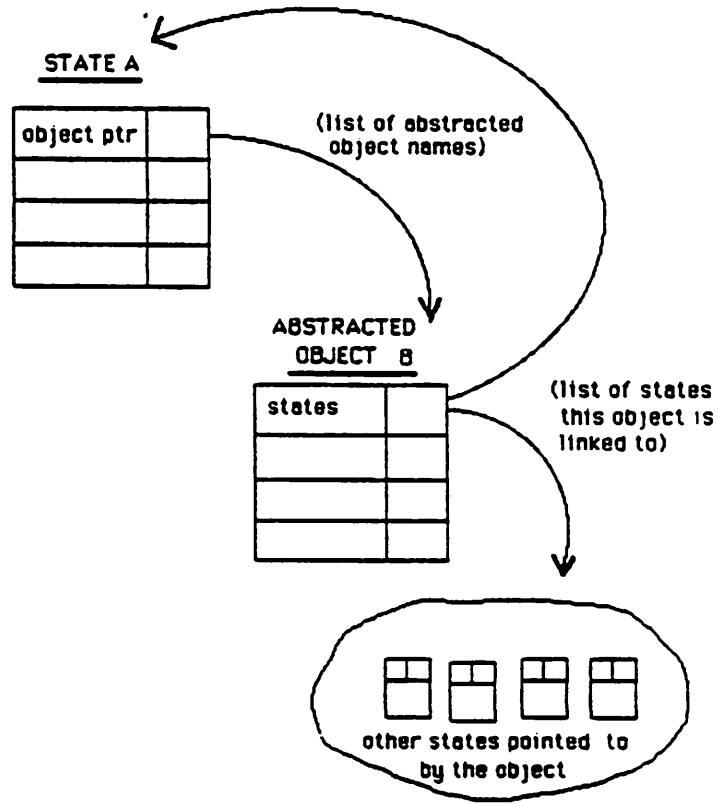


Figure 63: Illustrating the Use of Path Expressions
This figure illustrates how path expressions are used to guide access to some neighboring object attribute.

In some cases the functions also perform the selection of an appropriate object for analysis. This occurs during the instantiation of shorter track segments from longer track segment. Here the function calculating the time location list attribute of the track hypothesis object must group the existing track segments into classes of non-overlapping track segments and then choose the longest one from each group for diagnosis. Clearly, much knowledge is contained (hidden away) in these functions. This knowledge is procedural and is therefore not in a form that allows reasoning about it by the DM. For example, it would be better to specify the information governing the data dependent selection of objects for analysis declaratively so it could be changed easily. Such declarative representation would also make it accessible to some other program to reason about it in order to automatically change the selection criteria. One of the areas of future research would be to attempt to encode as much of this knowledge in a declarative form. Figure 64 shows examples of constraint expressions for some attributes of the KSI-OBJECT.

```

value    (F phd:if
          (PATH (F select-structure (PATH self object-ptrs))
                vmt-ids)))

level    pt

vmt-ids  (F phd:find-track-hyps
          (PATH self time-location-list)
          (PATH self event-class)
          pt)

f-u-n    (p:or (F make-hyp-state-name
               (F select-structure (PATH self object-ptrs))
               (F select-structure stimulus-hyps)
               level))

```

Figure 64: Constraint Expressions

Expressions like the ones shown above allow objects' attributes to be evaluated from their neighboring objects' attribute values.

To summarize, the constraint expressions allow arbitrary nesting of functions and arguments in order to capture the dependencies among the abstracted objects. The functions can be arbitrarily complex which is both good and bad: good because it provides a way for the model builder to specify procedurally relationships that are hard to represent in a declarative form; bad for the same reason, because unrestricted use of such functions can result in a badly designed model.

§4. The State Transition Diagram

A state is represented by the state data structure which is implemented as an attribute list. The state attributes fall into the three categories mentioned above: defining attributes, control attributes, and structure attributes. The state

transition arcs are represented implicitly by specifying the state's neighbors and the relationship among them (i.e., AND, OR, or PrefOR) and are thus structure attributes, as defined at the beginning of this chapter.

Defining Attributes. The key state attributes are the value and the path value attributes. The value attribute represents the state's value; i.e., whether the state is true or false, in the case of predicate states, and the type of a relationship among two object ratings, in the case of relationship states. In the ISBM, the value attribute contains an expression that is used to calculate the actual value when the model is instantiated.

With predicate states, in most cases, the state value is a function of the **vmt-ids** attribute of the abstracted object associated with the state. If there are some objects in the DVMT system, that fit the description given in the abstracted object attribute, then this attribute will contain their names. The state value will then be true if the **vmt-ids** is non-nil, false otherwise. In other cases, the value will be a more complicated function which checks whether some set of conditions, those represented by the state's predicate, holds. For example, the function associated with the state **KS-EXISTS** checks the DVMT data structures to see whether the specific knowledge source type exists in the particular system configuration. The function which determines whether the state **KSI-RATING-OK** is true, compares the rating of the knowledge source instantiation with the threshold that controls whether a **ksi** will be inserted onto the **ksi** queue. If the **ksi** rating is greater than this threshold, then the state will be true.

With relationship states, the function that calculates the value attribute takes

as input the actual values of the ratings represented by the parallel objects and returns the relationship among those ratings: less-than, equal, or greater-than.

There are two additional values for predicate states: an inapplicable value and unknown value. A predicate state has "inapplicable" value when it is a non-expandable state, i.e., when it is false and its existence is not consistent with the current system parameter settings. An example of a non-expandable state is a state PL in a system configuration that does not allow any pattern location hypotheses. Such a state could never be achieved by the system, not necessarily because the data is missing, but because if the system parameters do not allow the creation of that particular type of a hypothesis. A predicate state is unknown, when its value cannot be determined; i.e., the DVMT system does not maintain information that would allow the DM to determine that state's value and the value cannot be determined from the surrounding states (using Unknown Value Derivation).

The state value is determined locally, by checking to see whether some object or some situation exists in the DVMT system (in the case of predicate states), and by determining the type of a relationship among two object ratings (in the case of relationship states). The path value attribute is similar to the value attribute, in the type of values it can have and in that it represents whether some situation is true or false. The path value attribute is a more global measure than the value. It represents the results of the analysis in the surrounding states. (Or, to be anthropomorphic, it represents what the surrounding states *think* the state value should be.) In other words, it represents the evidence, in terms of causes or

effects, gathered from the surrounding states, as to what the state value should be. For example, if all of the state's predecessor states are false, then they will predict that the state's value will be false. If the state's successors are true, then they would imply that the state itself is also true. Consistency means that the local outcome of a situation should agree with the outcome predicted by looking at the events preceding or following the situation. It is the expectation of this type of global consistency that allows us to use the path value and the value attributes together to detect contradictions in the analysis. Such contradictions, i.e., the disagreement between the locally determined state value and the prediction of that value from the surrounding states, can mean several things:

- the analysis is incorrect,
- the model is wrong,
- the initial symptom was incorrect (i.e., it did not represent an undesirable situation).

The path value attribute represents whether a fault has been found along some path in the instantiated model. The path value is false if a primitive false state has been found along some path and true if no such state has been found. The path value attribute can be undetermined, when the model could not be expanded because of the system parameter settings, and unknown, when the value of some state along a path could not be determined. In a consistent instantiated model, the path value and the value attributes at each state will agree.

Because each state has up to six types of neighbors (predecessor and successor at each of the levels, same, lower, and upper), there can be up to six path values;

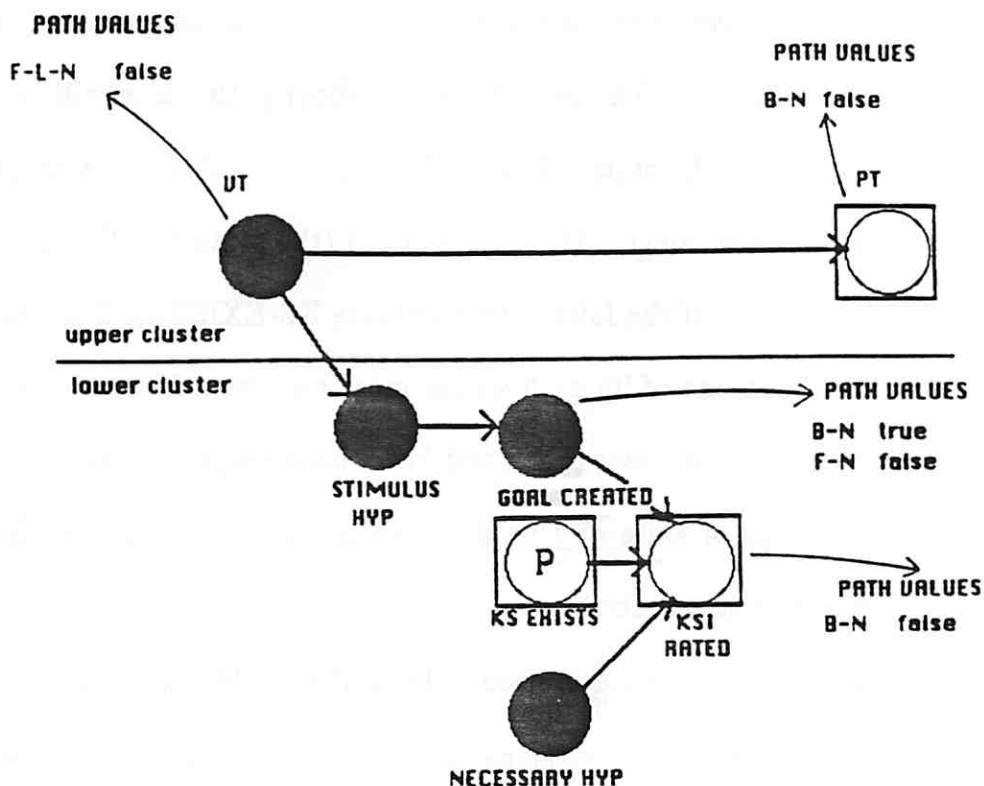


Figure 65: Multiple Path Values of a State

This figure illustrates how one state may have more than one path value if more than one of its neighbor types has been expanded. In such cases the overall path value for the state must be determined which is then propagated through the model.

one for each neighbor type. The path value for each neighbor type indicates whether a fault lies along that path or not. Figure 65 illustrates multiple path values. The figure shows an instantiated portion of the SBM which extends over two levels of the model hierarchy. The identified failure is the false primitive state **KS-EXISTS** in the lower cluster. Since the failure is a lower successor state of the state **VT**, then it will be the **f-l-n** (front lower neighbor) pathway that will be false. Only one of the possible six path values will be set because only one set of neighbors has been instantiated. The state **GOAL-CREATED** in the

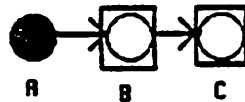
lower cluster will have two path values, since both its back and its front neighbors have been instantiated. The b-n (back neighbor) path value will be true, since no fault lies back of the state GOAL-CREATED. The f-n (front neighbor) value however will be false, since a failure lies ahead (the state KSI-SCHEDULED was not reached because of the false primitive state KS-EXISTS). If a state's value is false, then at least one of its path values must be false. If a state's value is true, then either all of predecessor (back and lower back neighbors) must be true. Any other combination of state and path values is inconsistent and indicates one of the problems mentioned above.

The path value attribute is used only in the ISBM. It is calculated by evaluating the neighbor list expressions and substituting the path value associated with each state for that state name in the instantiated neighbor list. Unlike the value attribute, which is calculated locally, based only on the characteristics of the state's abstracted object, the path value attribute depends on the path value attributes of the corresponding neighboring states. Because the path value needs information about what happens at the end of that path, it cannot be set at the time of the state creation, which is when the state value is set. Instead, we must wait until the end of the path has been reached, set the path value of the state there and then propagate it back up the way up through the causal chain until the state which initiated the diagnosis is reached. It is thus calculated recursively, by first determining the path value of the leaf states and then assigning the path values of the preceding states. The analysis ends with the assignment of the path value to the initial symptom that began the analysis.

There are thus two ways in which the path value attribute is calculated. First, in the case of leaf states in the model, it is assigned based on the state type and state value. The path value attribute is assigned to a state when that state cannot be expanded further. This happens when that state is a primitive state, when it is a node transition state (meaning it cannot be diagnosed further from the current node's perspective but is instead sent to another node for diagnosis), or when that a state has no expandable neighbors. The value of the path value at these end states is just the value of the state. Thus a false primitive state has is assigned a false path value F, a true primitive state a true path value, a node transition state the value of the state, and a non-expandable state a false value (since it could not have been reached by the system).

In the case of non-leaf states, the path value is calculated using the neighbor list expression and substituting the path values of each of the states. In the example in Figure 65, the b-n path value of the state KSI-RATED is calculated by substituting the path values of the instantiated back neighbors of that state. The resulting expression is (AND T (for GOAL-CREATED state) F (for primitive KS-EXISTS state) T (for NECESSARY-HYP state). The resulting value is false.

Why use the path value of a state, rather than its state value, to evaluate the neighbor list expression? What are the implications of using one or the other? As long as the diagnosis is consistent, i.e., false states lie on paths with faults and true states lie on paths without faults, then it does not matter whether the state or the path value is used since the result will be the same regardless. However, the whole reason for the path value is to have a consistency check on the diagnosis.



VALUE	A	B	C	subsequent states
<i>state values</i>	T	F	F	F
<i>using state</i>	T	T	F	F
<i>using path value</i>	T	T	T	T

PART B

Figure 66: Illustrating the Path Value Calculation

The upper part of the figure shows a portion of the evaluated SBM. The lower part shows a table with the states and path values.

Thus if at any point we have a disagreement between the state value and the path value, we want to propagate this disagreement all the way back through the causal path, to the original point. If we use the state values to calculate the path value, then we will get disagreement among the state value and the path value only at the state where the problem occurred.

Consider the example in Figure 66, part A. State A is true, states B and C are false, and there are no other states in between. The path value of state A is T. The table below shows how the path values will differ depending on whether we use the state value or the previous state's path value to calculate the current state's path value. The first row (state values) shows the values of the individual states. The second row (using state) shows the path values at each state if the previous state's state value is used to calculate the path value. The last row (using path value) shows each state's path value if the previous state's path value is used to calculate the path value.

Notice that if we use the state values to calculate the path values then the disagreement between the state and the path value will occur only at the state immediately following the problem. In this case the problem state is state B and its immediately following state is state C. If we use the state values to calculate the path values, then the inconsistency (a false state value and a true path value) will only show up at C. However, if we use the path values, the inconsistency will be propagated through all the states following the problem state. Since this is the effect we want, in order to propagate the inconsistent value back to the initial state, we use the path values of the neighboring states to determine a state's path value.

Control Attributes. Several attributes determine whether the state's neighbors will be expanded further or not. These are **expandable-state?**, **primitive-state?**, and **node-transition-state?** The **expandable-state?** attribute is used to decide whether that state is consistent with the current system parameter settings. This is necessary since the SBM represents all the possible system behaviors. Typically, however, only a small subset of these will be allowed by the system parameters. For example, not all the possible answer derivation pathways or inter-node communication pathways will be utilized. There would be no point in investigating the pathways not allowed by the system parameters, since they could not have been taken by the system. They are included in the model however in order to detect situations where it is these system parameters that are responsible for the misbehavior. In the SBM, this attribute contains the information necessary to determine whether the state is or is not expandable; i.e., whether

the situation it represents is consistent with the system parameter settings. In the ISBM, this attribute evaluates to either true or false. A state's neighbors can only be expanded if the **expandable-state?** attribute is true.

The **primitive-state?** attribute is true if the state represents a situation which has been designated by the model builder as a reportable failure. Primitive states are the leaf states of the model; i.e., they have no predecessors. Because of this, primitive states normally terminate the reasoning process. An exception is the case where the **primitive-state?** state attribute is used to control transition to another reasoning type. Currently, the **primitive-state?** attribute controls the transition to comparative reasoning. For example, the state **KSI-RATING-OK** is a reasoning-transition state. When it is reached, the DM switches to comparative reasoning.

The last attribute controlling the search termination is the **node-transition-state?** attribute. This attribute is set by the model builder and is true if the state is at a boundary between two nodes, false otherwise. If this attribute is true, then the state represents a situation which should be analyzed from another node's perspective. An example of these types of states are the **MESSAGE-SENT** and the **MESSAGE-RECEIVED** states. If one of these states is reached during diagnosis, the diagnosis stops and this state is saved as a symptom to be analyzed later by the corresponding node.

Three of the state attributes contain information about the abstracted object associated with the state: **object-ptrs**, **object-to-use**, and **object-grouping-specs** attributes. The **object-ptrs** attribute contains the name of the abstracted

object associated with the state. In the SBM, this is the uninstantiated object name, in the ISBM, it is the list of associated instantiated objects.

Because several states may refer to the same abstracted object, there are cases where we do not need to instantiate a new object when we are instantiating a state but can use an already instantiated object instead. The function of the **object-to-use** attribute is to allow the DM to recognize this situation and specify the existing object it should use. This attribute tells the DM whether to go ahead and instantiate a copy of the abstracted object whose name is in the **object-ptrs** attribute or whether to use an existing object. If an existing object is to be used, this attribute also specifies where to find that object. This will in most cases be the object associated with a neighboring state. In some cases the existing object will be an object pointed to by one of the attributes of the neighboring state's abstracted object.

As was mentioned in Chapter VI, the diagnosis is made more efficient by grouping similarly behaving objects together and creating only one state for them. The **object-grouping-specs** attribute contains the criteria that control the grouping of the instantiated abstracted objects. These criteria work by progressively splitting the initial group of abstracted objects created for that state into a set of equivalence classes. One state is then created for each of the classes. Currently, the most often used grouping criteria are whether the state is expandable or not (i.e., whether it represents a situation consistent with the current system parameter settings) and whether it is true or false (i.e., did the situation the state represent exist in the DVMT system or not). In some situation

object group. specs.	⇒	((<input specs><output specs.>)+)
input specs	⇒	~ (<attribute name><discriminating function name><result>)
output specs	⇒	(<attribute name> <discriminating function name>)

Figure 67: Syntax of the Object Grouping Specification

“~” means that all object in the undifferentiated group are considered. Attribute name is the name of the attribute whose value will determine the equivalence classes. Discriminating function name is the function that is applied to the attribute value to determine the classes. For the output specification, the additional argument <result> specifies that we are only interested in those objects whose attribute value yields the specified result after the discriminating function is applied.

additional grouping criteria are necessary, for example, when we need to group together objects belonging to different nodes in the DVMT system. The object grouping criteria are specified declaratively so that they can be changed easily. Figure 67 shows the syntax of the object grouping specification attribute.

The attributes discussed so far are common to both the predicate and the value states. The value states contain two additional attributes necessary for comparative reasoning. The first is the **parallel-state** attribute. In the SBM this contains information specifying the criteria for selecting the parallel state with which to compare the current state. This information is specified declaratively so that it can be changed easily, possibly even dynamically by the system itself, when the criteria for state matching change. The syntax of this attribute is as follows:

(F <Function name> <constraint expression> +).

In the ISBM this attribute contains a pointer to the parallel state in the ISBM. The other attribute is the relative-value attribute. This attribute represents the relationship among the values of the parallel states. It can be less-than, equal, or greater-than. The state attributes are summarized in Figure 68.

§5. State Transition Arcs

The state transition arcs are specified implicitly in the states' neighbor list attributes by listing each of the state's neighbors. The set of neighbor attributes contains the names of the state's neighboring states. There are six types of neighbors (successor and predecessor states at each of the three levels of the hierarchy), and therefore there can be up to six sets of neighboring states. These are called back-neighbors, back-upper-neighbors, back-lower-neighbors, front-neighbors, front-upper-neighbors, and front-lower-neighbors respectively; or **b-n**, **b-u-n**, **b-l-n**, **f-n**, **f-u-n**, and **f-l-n** for short. In practice, only a subset of the neighboring states will contain state names since a state will typically not have all possible neighbors.

The neighboring states and the relationships among them are specified in each state's neighbor attribute by the data type **neighbor list**. The syntax of the neighbor list data type is in Figure 69. In the ISBM, the neighbor list contains the uninstantiated state names related by their appropriate logical operators; i.e., the names are the names of the data structures describing the uninstantiated neighbor state. In the instantiated neighbor list, the names of the states are the instantiated states; i.e., the names of the data structures representing the instantiated version of the uninstantiated state. Since in some cases a single state in the SBM may be expanded into multiple states in the ISBM, there is not necessarily a one-to-one correspondence between the states in the uninstantiated neighbor list and its instantiated version.

f-n b-n	forward and backward neighbors at the same level
f-l-n b-l-n	forward and backward neighbors on a lower level
f-u-n b-u-n	forward and backward neighbors on a higher level
value	the value of the state (either T or F, or the value of the attribute the state represents)
relative-value	used by value states only. Contains the relationship among the values of the two parallel states during comparative reasoning
path value	associated with each neighbor list type. Represents the results of the analysis done on the surrounding states
object-ptrs	the type of abstracted object this state refers to
object-to-use	controls whether to use an existing object or whether to instantiate a new one and which one
object-grouping-specs	controls how instantiated objects are grouped together under one state
expandable-state?	determines whether the situation the state represents is consistent with the current system parameters
primitive-state?	T if state is a primitive state and can therefore be reported as an identified fault
node-transition-states	controls whether the state should be diagnosed from another node's perspective
parallel-state	used during comparative reasoning only. Contains information which controls the selection of the parallel state with which to compare the state.

Figure 68: State Attributes

neighbor-list \Rightarrow (<log.op.> <exp>+)
 exp \Rightarrow (<state name>) | neighbor-list
 log.op \Rightarrow PrefOR | AND | OR
 state name \Rightarrow uninstantiated state name

Figure 69: Grammar for the Neighbor List Specification

The neighboring states are related by the logical operator representing the type of relationship among them with respect to the current state. There are three types of logical relationships among the state transition arcs linking the states. These are AND (when all the states linked influence or are influenced by the state), OR (when any of the states or only one of the states is influenced by the state), and PrefOR (when only one of several states influences or is influenced by a state). Two of these, AND and OR, are important in the instantiated neighbor list for combining the path values of the neighbor states in order to determine the state's path value. The third, PrefOR, is only used during model instantiation to determine which of several possible neighbors will be instantiated.

Examples of AND and OR Relationships Among Neighbors. The back neighbors of the state KSI-RATED are KS-EXISTS, GOAL-CREATED, and NECESSARY-HYP and are specified by the **b-n** attribute as

(AND KS-EXISTS GOAL-CREATED NECESSARY-HYP).

In the instantiated model, each of the states will be replaced by the actual instantiated state name, and the expression would look like this:

(AND KS-EXISTS0001 GOAL-CREATED0002 NECESSARY-HYP0003).

In some cases a single state in the SBM may be expanded into several states in the ISBM. For example, the state MESSAGE-ACCEPTED will be expanded into as many states as there are nodes that the message can be received from. Since receiving the message from any one of those nodes is sufficient these states are ORed and the b-n of the state PT looks like this:

(OR (AND PL)(OR MESSAGE-ACCEPTED)(OR VT))

to indicate that PT can be derived from either of these three sources: ALL of the composite locations, or ANY received message from another node, or ANY of the event classes at a lower level of abstraction. In the instantiated model, this neighbor list might look like this:

(OR (AND PL0001 PL0002 PL0003)

(OR MESSAGE-ACCEPTED0004 MESSAGE-ACCEPTED0005)

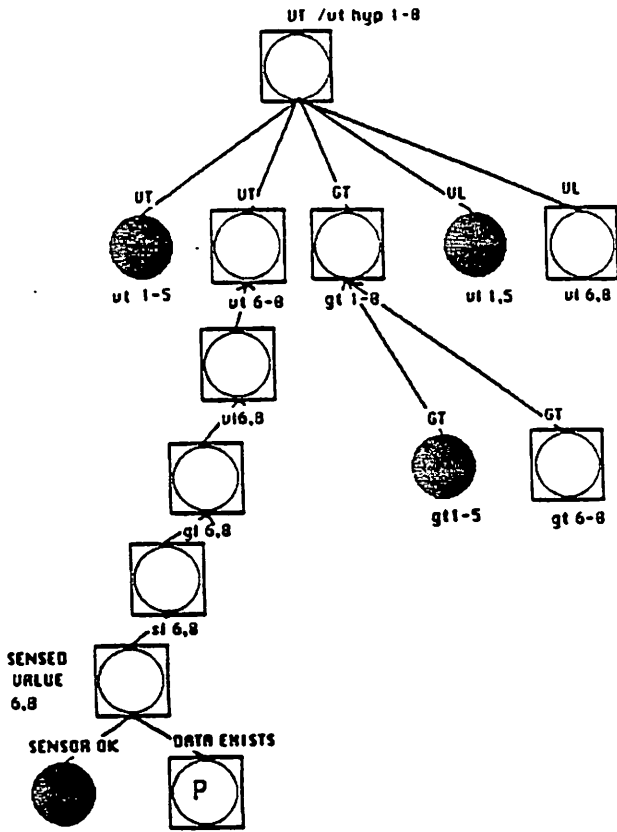
(OR VT0003)).

This means that there are three pattern locations necessary for the PT track (so the track must be three time locations long), and there are two nodes in the DVMT from which the pattern track could be received. The syntax of the neighbor list allows the nesting of these logical relationships to arbitrary depths.

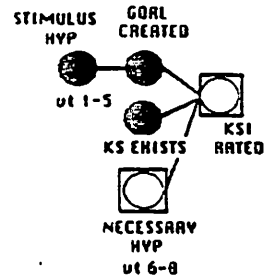
Motivation and Use of the PrefOR Logical Operator. The PrefOR attribute is different from the AND and OR in that represents the preference of

the model builder for analyzing how the DVMT system processed its data. Unlike the AND and OR operators, the PrefOR operator is used only during model instantiation time to control which of several possible neighbors to instantiate. This is one example where the choice of the state to instantiate depends on the data. It occurs when we are making the transition from a reflexive track state to either the same state or to the corresponding location state. Why do we need to make this choice? Why not just instantiate both the shorter track segment neighbors and the location neighbors? The following example will illustrate why we need the PrefOR operator to reduce needless analysis of the DVMT system behavior.

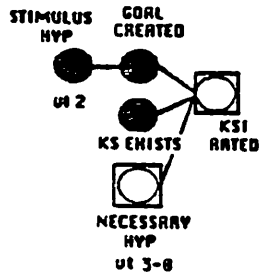
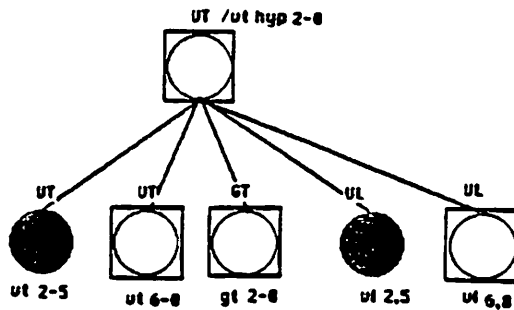
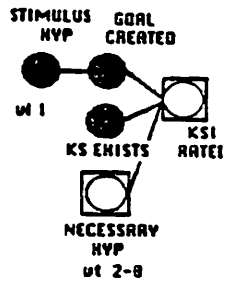
Consider what happens when we are diagnosing why the vehicle track hypothesis from time 1-8 was not created in the DVMT system. We begin with the state VT, which is false, and its associated abstracted object, which represents the vehicle track hypothesis from time 1-8. Refer to Figure 70, part A. We instantiate its back neighbors: first, the track segments VT 1-5, which is true, VT 6-8, which is false, followed by the GT 1-8, which is false, and finally the locations, VL 1-5, which is true and VL 6-8, which is false. The false states are followed backwards, eventually resulting in identifying the reasons why they do not exist. The shorter vt segment 1-5 is also analyzed, and the reason for its not being extended is identified as the missing vt segment 6-8 (see Figure 70, part B). The problem occurs when we try to diagnose why the existing locations 1-5 were not extended into the desired track 1-8; in other words, when we continue the diagnosis at the true state VL 1-5.



PART A



PART B



PART C

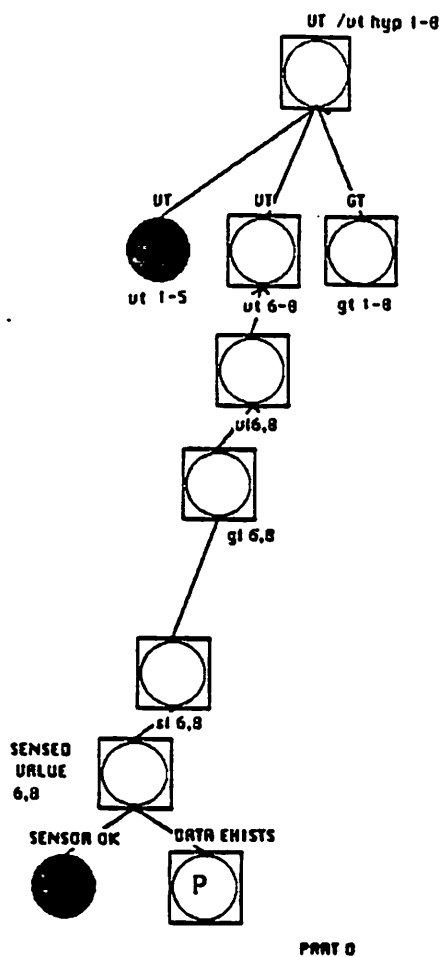


Figure 70: Motivating the Need for the PrefOR Logical Operator
 Part A of the figure shows the the instantiated Answer Derivation Cluster for diagnosing why the vt 1-8 hypothesis was never created. Part B shows the instantiated Ksi Scheduling Cluster for diagnosing why the existing vt 1-5 segment was not extended to the vt 1-8. Part C shows how the Ksi Scheduling Cluster is instantiated to determine why the vt 1 was not incorporated into the vt 1-8 track. This leads to the instantiation of the Answer Derivation Cluster to determine why no vt 2-8 hypothesis exists. This in turn leads to the instantiation of the Ksi Scheduling Cluster to determine why the vt 2 hypothesis was not extended to vt 2-8. Part D shows a more direct diagnosis, using the PrefOR relationship to control the order of state instantiation. This diagnosis identifies the same set of failures but instantiates only a small fraction of the SBM.

The system first looks for VT 2-8 to merge with VL 1 in order to derive VT 1-8. (Only the hypotheses that could produce vt 1-8 in one step are considered.) VT 2-8 does not exist, so the system tries to determine why, by instantiating the model in Figure 70, part C. This leads to the instantiation of states VT 2-5 (true), VT 6-8 (false, merges with existing state), GT 2-8 (false), VL 2,5 (true), and VL 6,8 (false, merges with existing state). Continuing in the same spirit, VL 2 is now analyzed to see why it was not extended to the track VT 2-8. The problem is the lack of hypothesis VT 3-8. This type of diagnosis continues, analyzing the lack of VT 3-8, then VT 4-8, and finally VT 5-8. In other words a complete search is performed which does not necessarily parallel the type of search the DVMT system would do in attempting to derive the vt 1-8 hypothesis.

A more direct way to diagnose this problem would be to first try and determine why the shorter segments were not extended and only when no shorter segments exist, (as is the case with the segment VT 6-8), go to the corresponding locations and determine why they do not exist. This is shown in Figure 70, part D. As can be seen from the figure, this diagnosis identifies the same set of failures but does so much faster, without instantiating the useless intermediate states.

There is one other case where the neighbor state in some way depends on existing data. This occurs when a lower cluster state needs to be linked to its upper state neighbor. Recall that in many cases the lower cluster represents a repeated sequence of states which occurs in between each pair of states in the higher cluster. This necessitates a data-dependent specification of state names in the lower level cluster, (i.e., the upper cluster neighbor names of these states),

since the names of the upper back and upper front neighbors of the lower level cluster will depend on what data the lower cluster is representing. Figure 71 illustrates this problem when DM tries to go from the Ksi Scheduling Cluster to the Answer Derivation Cluster. Because the neighbor state name is data dependent, it cannot be known a priori and specified in the model before the model is instantiated. Since the neighboring state names depend on the values some known attributes of the already instantiated states or objects, they must therefore be expressed as functions of those attributes in the ISBM. These functions are evaluated when the model is instantiated to give the actual neighbor state.

Notice there are five states that are linked to states at the next level of the hierarchy: **STIMULUS-HYP**, **NECESSARY-HYP**, and **KSI-EXECUTED** in the Ksi Scheduling Cluster, and **COMM-KSI-RATED**, and **COMM-KSI-EXECUTED** in the Comm Ksi Scheduling Cluster. Thus the upper cluster neighbors of these states will have the data dependent state name specification. In the case of the **STIMULUS-HYP** and **NECESSARY-HYP** states, we must construct the names of the states in the Answer Derivation Cluster that these states will be linked to. This name depends on the level of the **HYP-OBJECT** associated with the **STIMULUS-HYP** and **NECESSARY-HYP** states. Similarly, the state which is the neighbor of the **KSI-EXECUTED** state depends on the level of the output hypothesis of the **KSI-OBJECT**.

In the case of the Comm Ksi Scheduling Cluster, the two states whose neighbors depend on the exact data are: **COMM-KSI-RATED** and **COMM-KSI-EXECUTED**. **COMM-KSI-RATED** could go to either **MESSAGE-EXISTS** (if we are dealing

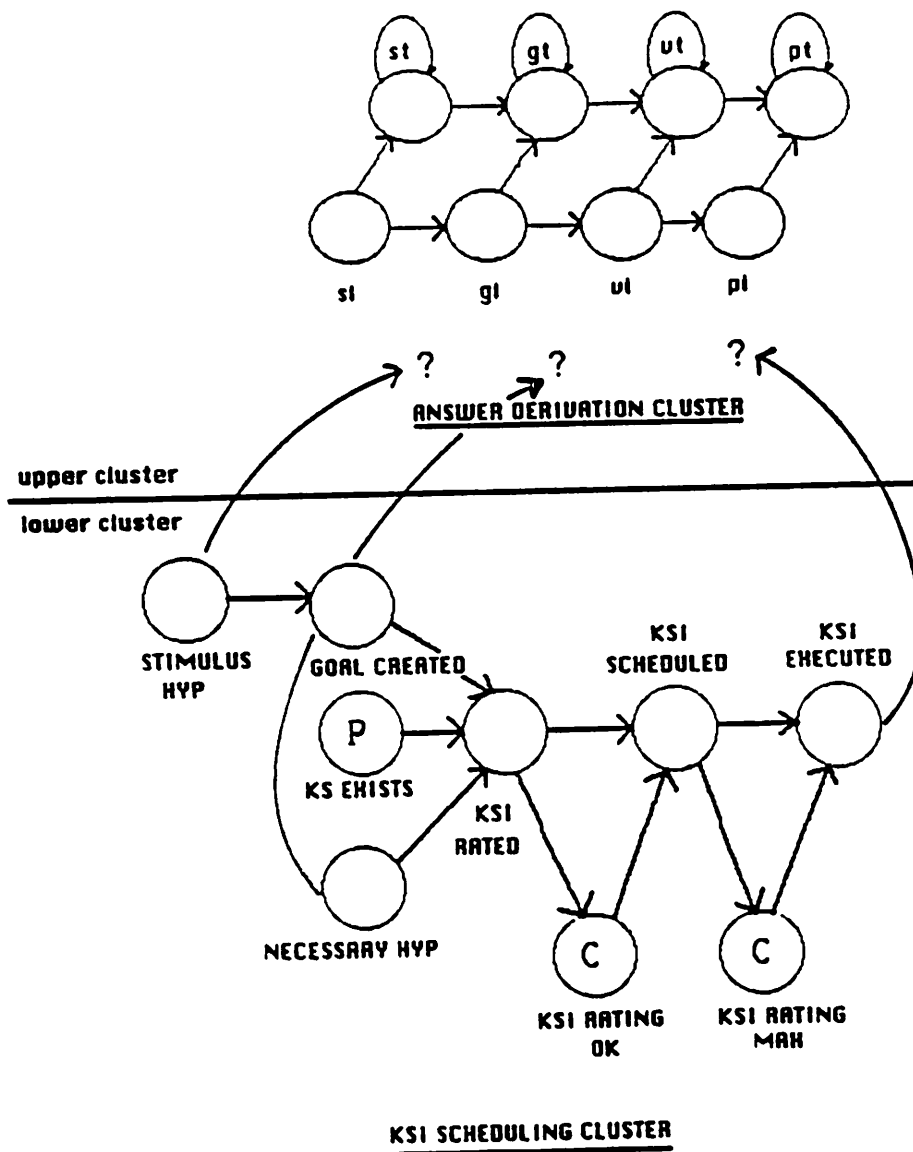


Figure 71: Data Dependent State Neighbors in Ksi Scheduling Cluster
 This figure illustrates the need for data dependent state neighbor specification. The upper neighbors of the states *STIMULUS-HYP*, *NECESSARY-HYP*, and *KSI-EXECUTED* in the *Ksi Scheduling Cluster* depend on the type of hypothesis being considered. The neighbor names are therefore a function of the hypothesis level.

b-u-n of STIMULUS-HYP	(OR (F make-hyp-state-name (PATH (F select-structure (PATH self object-ptrs)) level)))
b-u-n of COMM-KSI-RATED	(OR (F make-message-state-name (PATH (F select-structure (PATH self object-ptrs)) ks-type) b-u-n)))
f-u-n of COMM-KSI-EXECUTED	(OR (F make-message-state-name (PATH (F select-structure (PATH self object-ptrs)) ks-type) f-u-n)))

Figure 72: Illustrating the Specification of the Dynamic Neighbor Names

with a send ks) or MESSAGE-RECEIVED (if we are dealing with a receive ks). COMM-KSI-EXECUTED could go to either MESSAGE-SENT (if dealing with send ks) or MESSAGE-ACCEPTED (if dealing with receive ks). Here the state name depends on two attributes: ks-type and the link type. Figure 72 shows examples of this type of data-dependent neighbor specification.

Following the syntax of the neighbor expressions as well as the usual syntax of the attribute specifications, we have a function and we specify its arguments, which are the attributes of the current state and its associated objects. Notice that the arguments can even extend to other objects, using the path feature of specifying object/state attributes.

In conclusion, the dynamic state name specification is necessary anytime there

is a choice of states for a neighbor, which is a function of the existing data and cannot therefore be specified in the SBM. There are thus two cases where the state's neighbors are data-dependent. In one case, there is a choice among several neighbors and the first one which can be successfully instantiated is used. This is the case where the PrefOR logical operator is used. The other case is the one described above, where the actual state name is a function of the data and must therefore be calculated at instantiation time when the necessary attributes have already been evaluated.

§6. Model Instantiation: An Example

A high level view of the model instantiation was given in Chapter III. This section illustrates the instantiation by a detailed example. In order to make the example simpler, we will assume that only one model hierarchy is used, the Answer Derivation Cluster, and that the only the states consistent with the system parameters are included (VT-VL-GL-SL). We will begin with the symptom state VT, representing the vt 1-8. We will instantiate its back neighbors, illustrating the use of the PrefOR operator and the function of the splitting attributes. The uninstantiated state VT and its associated object VT-HYP-OB are shown in Figures 73 and 74.

We first assign the attribute values of the symptom object to represent the vehicle track hypothesis. This evaluated object is shown in Figure 75. We evaluate the vmt-ids attribute of this object by looking for the hypothesis it corresponds

Uninstantiated VT State

```

((name          vt)
 (f-n           (p:or
  (p:or (message-exists))
  (p:or (vt))
  (p:and (pt)) ))
 (b-n           (p:or
  (p:or (message-accepted))
  (p:or (gt))
  (PrefOR
   (p:and (vt))
   (p:and (vl))))))
 (f-l-n         (p:or (stimulus-hyp)(necessary-hyp)))
 (b-l-n         nil)
 (f-u-n         nil)
 (b-u-n         nil)
 (object-lumping-specs ((
  (~ (expandable-state-function echo))
  ((expandable-state-function echo t) (vmt-ids phd:if)))
 (value expandable-state?)))
 (value         (f phd:if
  (PATH (f select-structure
  (PATH self object-ptrs))
 vmt-ids)))
 (expandable-state? (f phd:if
  (PATH (f select-structure
  (PATH self object-ptrs))
 expandable-state-function)))
 (object-ptrs      vt-hyp-ob)
 (object-to-use    ((((* hyp)(* hyp-ob)) new)
  ((necessary-hyp (* hyp-ob)) same)
  ((message-exists message-ob) new)
  ((stimulus-hyp (* hyp-ob)) same)))
 (primitive?      nil)
 (symptoms        nil))

```

Figure 73: Uninstantiated VT State

Uninstantiated VT-HYP-OB Abstracted Object

```

((name          vt-hyp-ob)
 (vmt-ids (f phd:find-track-hyps
  (PATH self time-location-list)
  (PATH self event-class)
  (PATH self node)
  vt))
 (expandable-state-function (f phd:in-ia?
  (PATH self time-location-list)
  (PATH self event-class)
  (PATH self node)
  vt))
 (s-attributes (((vt vt-hyp-ob) (time-location-list))
  ((vt pt-hyp-ob) (event-class))))
 (attribute-evaluation-sets
  ((time-location-list event-class level node rating)))
 (context-dependent-attributes
  (event-class s-attributes time-location-list))
 (time-location-list
 ( ((vt message-ob) (PATH x1 t1l/tr1))
  ((vt gt-hyp-ob) (PATH x1 time-location-list))
  ((vt vl-hyp-ob) (PATH x1 time-location-list))
  ((vt pt-hyp-ob) (PATH x1 time-location-list))
  ((vt vt-hyp-ob)
  (f create-track-segments
  (PATH x1 time-location-list)
  (PATH x1 event-class) vt
  (PATH x1 node))))))
 (event-class
  (((vt message-ob) (PATH x1 event-classes))
  ((vt gt-hyp-ob) (f phd:higher-level-event-classes gt
  (PATH x1 event-class)))
  ((vt pt-hyp-ob)
  (f phd:lower-level-event-classes pt
  (PATH x1 event-class)))
  ((vt vt-hyp-ob) (PATH x1 event-class))
  ((vt vl-hyp-ob) (PATH x1 event-class))))
 (level          vt)
 (node          (PATH x1 node)))

```

Figure 74: Uninstantiated VT-HYP-OB Abstracted Object

Instantiated VT-HYP-OB Abstracted Object

```
((name          vt-hyp-ob1)
 (states        vt1)
 (vmt-ids (h0043))
 (expandable-state-function t)
 (s-attributes  time-location-list)
 (attribute-evaluation-sets
  ((time-location-list event-class level node rating)))
 (context-dependent-attributes
  (event-class s-attributes time-location-list))
 (time-location-list
  ((1 (1 1))(2 (2 2))(3 (3 3))(4 (4 4))
   (5 (5 5))(6 (6 6))(7 (7 7))(8 (8 8))))
 (event-class 1)
 (rating      *)
 (level       vt)
 (node        1))
```

Figure 75: Instantiated VT-HYP-OB Abstracted Object

Instantiated VT State

```

(name          vt1)
(type          track-hyp)
(f-n           nil)
(b-n           (p:or (p:and (vt vt2 vt3))))
(f-l-n        nil)
(b-l-n        nil)
(f-u-n        nil)
(b-u-n        nil)
(object-lumping-specs ((
  ( ~ (expandable-state-function echo))
  ((expandable-state-function echo t) (vmt-ids phd:if)))
(value expandable-state?)))
(value         f)
(expandable-state?      t)
(object-ptrs           vt-hyp-ob1)
(object-to-use         new)
(primitive?           nil)
(symptoms             nil))

```

Figure 76: Instantiated VT State

to in the DVMT system. No such hypothesis is found so the value of the `vmt-ids` is false. We evaluate the state attributes `value` and `expandable-state?`. `Value` is false, because the `vmt-ids` is false. `Expandable-state?` is true, because this object does fall within the node's interest area. The evaluated state is shown in Figure 76. We now continue to expand its back neighbors. First, we access the `b-n` neighbor list of the state `VT`. This is `(PrefOR (AND VT)(AND VL))`, indicating that either shorter track segments are to be instantiated, or, if none exist, the locations necessary to derive the track are to be instantiated. (We have omitted some neighbors in order to make the example easier to follow. The full

set of back neighbors is actually (OR (PrefOR (AND VT)(AND VL)) (OR (GT)) (OR (MESSAGE-ACCEPTED))).) We first attempt to instantiate shorter tracks and therefore choose the state VT as the first one to expand. We first have to determine whether to create a new object or use an existing one. We look at the state's object-to-use attribute, which tells us that coming from a VT-HYP-OB we need to instantiate a new object. We now look into the object-ptrs attribute of the state VT to see what type of an abstracted object we need to instantiate and find that it is VT-HYP-OB. We begin by evaluating the object attributes. First, we look at the attribute-evaluation-sets attribute to determine the order of the attribute evaluation. This turns out to be the following:

((time-location-list event-class level node rating)).

In other words, all the relevant attributes can be evaluated at the same time. We therefore begin with the first, **time-location-list**. The constraint expression for this attribute is:

```
(F create-track-segments
  (PATH x1 time-location-list)
  (PATH x1 event-class)
  vt
  (PATH x1 node))
```

The function **create-track-segments** is the most complex function in the constraint expressions. It looks for shorter track segments which could be used to construct the vt 1-8 hypothesis. It first groups all segments into classes such that all non-overlapping segments form a separate group. The longest element of each group

is then chosen for diagnosis. In this case, a track segment 1-5 exists (as do many shorter tracks segments, such as 1-3, 2-4, 3-5), but no other track exists. The two track segments we have are 1-5, which exists, and 6-8, which does not. Create-track-segments therefore returns two values 1-5, 6-8.

Notice that time-location-list is a splitting attribute. This means that a separate object needs to be created for each of the values obtained for this attribute. In this case we create two objects, one representing the existing track 1-5, and the other representing the missing track segment 6-8. We evaluate the remaining attributes and obtain two VT-HYP-OB's. The instantiated graph is shown in Figure 77, part A. We now determine the vmt-ids and the expandable-state-function of each object. The 1-5 track exists, so vmt-ids is the name of the corresponding DVMT hypothesis. The expandable-state-function attribute is true, otherwise this hypothesis could not exist. The other track does not exist, so the vmt-ids is false and its expandable-state-function is true, i.e., the state is within the node's interest area.

We now apply the object grouping criteria in order to create the appropriate number of states. The object-grouping-specs for the VT state are:

$$(\sim (\text{expandable-state-function echo}))$$

$$((\text{expandable-state-function echo t}) (\text{vmt-ids phd:if})).$$

In other words, the objects are to be first divided into groups by the value of their expandable-state-function (i.e., expandable objects in one group and non-expandable objects in another group). The expandable objects are then to be divided into groups by the value of the vmt-ids, in other words, all objects which

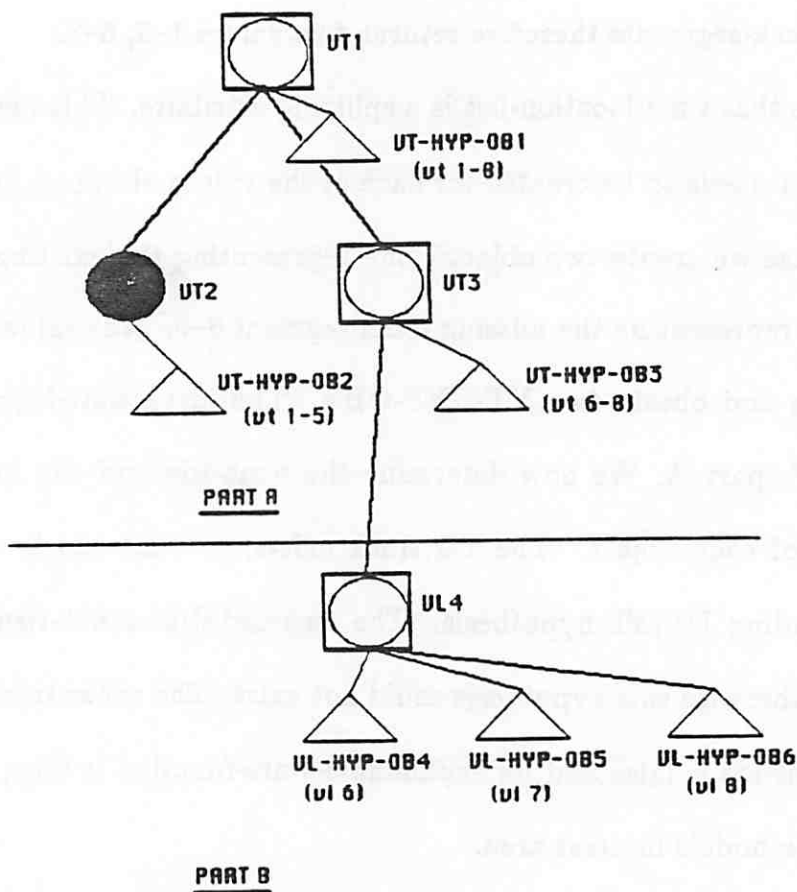


Figure 77: Instantiated SBM

This figure illustrates the result of the SBM instantiation described in the text.

have a corresponding object in the DVMT system in one group and all objects which do not in the other. In this case, this results in two groups: VT-HYP-OB 1-5 in one, and VT-HYP-OB 6-8 in the other. We now create a state for each group and attach the objects to the state. We thus end up with state VT0002 attached to VT-HYP-OB0002 (vt 1-5), and VT0003 attached to VT-HYP-OB0003 (vt 6-8). We next evaluate the value attribute of the states and find that they are true and false respectively. We now have two instantiated states: the true state VT0002, representing the existing vt segment 1-5, and the false state VT0003, representing the vt segment 6-8. In a real system run, the front lower neighbors of the true state would be expanded, which would lead to reasoning at the lower level Ksi Scheduling Cluster. However, we are omitting this cluster to make the example simpler and will therefore continue with the state VT0003. This state is false and we will therefore expand its back neighbors. The back neighbors are, as before (PrefOR (AND VT)(AND VL)). This time however, no shorter tracks exist so the VT back neighbor cannot be instantiated. The VL neighbor is therefore instantiated with the three locations representing time locations 6 through 8. Since none of these objects exist, they are grouped under one state, VL0004. The final portion of the instantiated model is shown in Figure 77, part B.

§7. Summary

This chapter explained the details of the DVMT model implementation. It discussed how the states and objects are represented, listing all the attributes of the state and object records and explaining their structure and use. Both the state and the object attributes can be divided into three categories: those defining the state (i.e., representing the predicate or the relationship) or the object (i.e., representing the characteristics of the object); those controlling how the states or the objects are instantiated, and those containing information about the structure of the model.

The **path value** state attribute was discussed at length. This attribute represents what the surrounding states think that state value should be. It is by comparing the locally determined state value and the global evidence from the surrounding states that the DM checks the consistency of the diagnosis and the model.

The details of the state transition arcs were explained, in particular, the the **PrefOR** logical operator was motivated by examples requiring data-dependent neighbor instantiation. The constraint expressions linking the abstracted objects were described, including the types of functions and arguments they use and the expression syntax. The chapter concluded with a detailed description of how the **SBM** is instantiated.

Chapter IX

CONCLUSIONS AND CONTRIBUTIONS

"We grope. We take risks. All I had when I began 'Of Human
Bondage' was the conjunction 'and'.
I knew a story with 'and' in it could be delightful.
Gradually the rest took shape."
- Woody Allen, in *Reminiscences: Places and People (Side Effects)*

This chapter summarizes the work described in the dissertation. It discusses what we set out to do and what we actually accomplished, highlights the major problems encountered, and discusses the contributions of this research.

§1. Introduction

AI research, perhaps due to the youth of the field, often involves the construction of very complex systems but leaves the applicability of those systems to other domains as an exercise for the reader. Furthermore, AI is notorious for non-reproducible results and lack of theories on which to build [34]. Many researchers tend to take the "reinvent the wheel" approach to research. This is due to many factors. Partly this approach seems to have established itself as a tradition and independent corroboration are often not considered essential in AI

research. This may be due however to the fact that it is often very difficult to determine exactly what someone has done. In most AI papers proposed research blends with the results of the implemented system; and often it is hard to distinguish between the wishful thinking of the researcher and the actual results. This makes it very difficult to build on someone else's work. Another problem, again symptomatic of a young discipline, is the lack of standardized definitions and the use of every day words with an often ill-defined technical meaning. This "tower of Babel" state of the field is, I believe, in large part responsible for the lack of communication among researchers. People often work in very different domains using different formalisms. It becomes difficult to distinguish between the solutions to the real problems and solutions to problems that are the results of inappropriate design choices in the systems. It is difficult to take someone else's approach, determine whether it is applicable to the problem at hand, and to adapt it to the problem. All these factors contribute to the lack of well defined theories which make systematic research difficult to pursue. **The field is more like an apprenticeship; one learns by doing and gains insights in this way.**

We have tried here to address some of the problems mentioned above. The glossary in Appendix A defines all technical terms used in this document. Chapter VII discusses how this work relates to other diagnostic systems using causal models, emphasizing the features of the systems that are due to the specific domain characteristics. Chapter VI gives some principles for constructing a model of a complex problem-solving system. In order to facilitate the exercise, should the

reader want to apply this approach to another domain, Section 6 discusses some ideas about how such a transfer should be approached and what domains it would be most suitable for. We have also tried to discuss the limitations and shortcomings of the design choices we made. Section 4 summarizes the major problems we encountered, some of which we solved, some not. As with any research, we have found more questions than answers. The final chapter in this dissertation, (Chapter X), discusses some of the many possible directions for future work. The last section in this chapter sets the stage for this final chapter.

§2. Defining the Problem

We can view research as the exploration of unknown territories. In exploration, as in research, the choice of an area to explore and the methodology used depends on the explorer's experience, on the traditions currently in vogue, on the types of journeys most likely to be financed, and on the equipment available. It often happens that the explorer sets out to find a shorter route to India and discovers instead a whole new continent. In such cases it is important to realize that this unexpected discovery is in fact an interesting new continent rather than an annoying obstacle on the way to the original goal.

The project definition thus depends on many factors. One of them is the type of view of the area being explored. We initially defined our project, in the best of AI traditions, from a bird's eye view. From that point of view making the DVMT system fault tolerant seemed a feasible task. When the fog cleared, we uncovered

three major peaks to explore in the area of fault tolerance: detection, diagnosis, and correction. These peaks were grouped close together, in fact they seemed to lie on a plateau that would have to be scaled to get to any of them. **This common task to achieve in order to solve any of the three problems was the representation of the behavior of the DVMT system.** As we started scaling this plateau, we realized that we will not be able to climb all three peaks. The detection and diagnosis were grouped closer together and we therefore chose those two for detailed exploration. Hoping that the correction peak would be made considerably easier in the future by the experience gained in climbing the plateau and the other two peaks.¹

Having narrowed down the exploration to the two closely placed peaks of detection and diagnosis, we began scaling the diagnosis one first. It quickly became apparent that we will also have to give up the detection peak. While we were able to chart a rough course up the mountain, we realized there would be no time to actually climb it. Like all rough, high-level descriptions of a problem, we know that our charted course for scaling the detection peak left out many details. And although it seems relatively easy, viewed from the diagnosis peak, we know from experience that distance is deceiving. Appendix C discusses the problems and some initial solutions for the detection problem. **Our research therefore**

¹There is a close relationship among all three; detection, diagnosis, and correction. It often seems to be a matter of classification, rather than the actions performed, which is called which. The more detailed the detection, the closer it becomes to diagnosis. The better the diagnosis, the easier and more reliable the correction. Of course, each has issues that are unique to that one problem. For example, in detection one has to deal with the frequency of monitoring the system. The similarities however far outweigh the differences.

concentrated on diagnosis.

§3. Solving the Problem

Having selected diagnosis for detailed exploration, we had to choose a methodology. The combination of the type of system we were dealing with (a complex problem-solving system whose structure was completely known), and the current methods for problem-solving in AI (knowledge-based methods and reasoning from first principles) led us to choose a **causal model of the DVMT system** as the knowledge-base for the diagnosis. The construction of this model required much work. We had to decide what aspects of the system to represent in the model and how best to represent them. This led to the construction of a new language, the modeling formalism, within which parts of the DVMT system were represented. This amounted to in effect re-coding aspects of the DVMT system in the modeling formalism.

Having represented the system, we needed to develop methods of using the model to reason about the system's behavior. We ran into combinatorial problems, which led back to modifying the initial model in order to encode into it features that would allow us to control the combinatorics. The grouping of similarly behaving objects and the use of underconstrained objects are both examples of solutions to the combinatorial problems.

We have implemented a diagnostic component of the DVMT system, the Diagnosis Module (DM), that uses a model of the system to diagnose the DVMT

behavior. Currently, the diagnosis is done in a centralized manner. We have some ideas about how to distribute the diagnosis component over the network, such that each node in the DVMT system has its own DM. The model already provides some tools for making this change easier, for example, the consideration of certain states in the model as node transition symptom states. Whenever these states are encountered during diagnosis they are saved for later diagnosis by the appropriate node. In a fully distributed diagnosis system the DM's at the individual nodes would cooperate, exchanging symptoms and the results of diagnoses. Together these modules would engage in distributed problem solving, much like the underlying DVMT system being diagnosed.

We have tested the DM on actual DVMT system experiments, two of which were presented in detail in Chapter V. The DM was able to diagnose the failures in the DVMT system. In cases where the DM could not pin-point the actual failure to one or two faulty parameter settings, it was at least able to narrow down the problem, making it much easier for the user to debug the details. Currently, the DM is being used to help debug the DVMT system runs and is especially good for novice users who often make mistakes in setting the many parameters that control the DVMT system.

Chapter I listed several applications of a fault-diagnosis component of a problem-solving system. These were: meta-level control by adapting the system parameters, fault-tolerance of the system, aid in debugging the system during development, and aid in explaining the system behavior. Much work needs to be done to achieve meta-level control and fault tolerance. Both the detection and the

correction problems will have to be solved first and, although we believe we have addressed some important issues, we are sure that many more difficult problems will emerge as one begins to seriously consider these problems. The immediate use of the DM is now in debugging and explaining the DVMT system behavior.

§4. Major Difficulties Encountered

Constructing the DVMT system model proved a much more complex task than we thought originally. The complexities of the model construction are discussed in Chapter VI. Mostly the problems were in choosing which parts of the DVMT system to model and representing them in a form that would allow efficient reasoning about the DVMT system.

Combinatorial problems emerged, both in the model construction and in the model use. In the model construction, we solved them by parametrizing aspects of the DVMT system and making the knowledge of this parametrization available to the DM. It would then properly expand the concise model representation during model instantiation to correspond to the actual system data. In model use we solved the combinatorics by allowing the existing data to constrain the search whenever possible and by grouping together classes of similarly behaving objects. We have also developed ways of representing classes of objects and underconstrained objects and reasoning about the whole class rather than the individual cases. (Chapter VI discusses the distinction between grouping together objects and representing a class of objects by an underconstrained object.)

An important question to ask about a modeling formalism is how closely it represents the behavior of the real system. Can it capture all the different system behaviors? Does it represent only those behaviors that the system can actually do? In other words, does the model introduce some artifacts that actually never occur in the real system? We believe that the model does capture all the possible behaviors (within the subset of behaviors represented). It does introduce some artifacts (for example, tracks one location long that would never exist in the real system), but this is done for reasons of efficient representation and does not detract from the modeling formalism. The important thing is that the model detects only real failures and this is certainly the case.

§5. Contributions of this Research

Continuing with the explorer's metaphor, there are different types of explorers and they work in different ways. Some prefer to map out in detail a small section of some already generally known area. Others prefer to go to a completely unknown territory and, if they come back, bring reports of the general features of that territory. We feel this research belongs to the second category of research.

It pulls together a number of known techniques (diagnosis, simulation, qualitative reasoning, and constraint networks) and describes a few new ones (Comparative Reasoning and the use of Underconstrained Objects) in an attempt to map out the problems encountered in representing and reasoning about problem-solving system behavior. The work described here is a first pass at this large

problem. It describes the major features of the problem, mostly in order to point to them as interesting areas of continued research. We hope that someone could construct a similar model based on the ideas presented in this dissertation, and that this construction would take considerably less time. We feel we have described the problems encountered well enough that someone could take a small portion of this large problem and continue to explore this small area in detail.

What are the interesting features of this new terrain and in what way have we helped in future exploration? First of all, we would like to take credit for being the first, though we are sure there must have been some Vikings, whose findings we missed. We showed that the recently popular AI approach to diagnosis, based on causal models of the device, can be applied to AI systems themselves. We believe this is the first time that reasoning from first principles was applied to a problem-solving system and we have demonstrated that this is indeed feasible. Because this is a first attempt at modeling a problem-solving system, we feel that identifying what the issues are in modeling and reasoning about problem-solving systems is itself a contribution. To be sure, much work remains to be done to make the DM really robust and efficient, but we feel we have laid some foundations upon which to build.

One of the initial attractions about this problem was to learn what knowledge a system must have in order to reason about its own behavior? An important goal of any research in a new domain is to develop methods for representing this domain in a systematic manner. This involves identifying the "primitives" in the domain, from which more complex structures can be constructed. This is what

is called **developing an ontology of the domain**. We feel we have taken the first steps towards this in defining the criteria for what the objects are and what types of relationships need to be represented among them. Chapter X discusses some ideas about how we would like to develop these preliminary findings into a complete language for representing problem-solving system behavior.

We have found the separation of states and objects useful for efficient representation of the system. We therefore separate to some extent the structure of the system (the objects) from its behavior (the states). In this respect our causal model lies somewhere in between the models used in medicine, which represent only the behavior and no structure, and the models in hardware, which represent both the structural and the behavioral hierarchies.

What types of reasoning are necessary to diagnose problem-solving system behavior and how are these different from other types of diagnosis? Various forms of what we call Forward Causal Tracing, Backward Causal Tracing, and Unknown Value Derivation can be found in most diagnostic systems (see Chapter VII). We have not however seen examples of Comparative Reasoning, State Matching reasoning, or the application of Forward Causal Tracing (i.e., simulation) to predicting the effects of faults on system behavior. Nor have we seen the use of Underconstrained Objects. We believe that both the Comparative Reasoning and the use of Underconstrained Objects are necessary when dealing with problem-solving systems. Comparative reasoning is needed because there often are not any absolute standards for correct system behavior. The DM must therefore choose an object from within the system to use as a temporary model for

correct behavior. Reasoning with underconstrained objects is necessary simply due to the complexity of the system, in order to avoid combinatorial problems.

In any knowledge-based system, a choice must be made as to where to focus the "smarts" of the system. We can either focus on the representation and construction of the search space, making it as small as possible, and then apply simple search techniques to it. Or we can focus on making the search (the inference engine) smarter and make it capable of dealing with large search spaces. The knowledge in this system is clearly in constructing a small search space. We believe that the major contributions of this research are the techniques we developed for dealing with the complexity of the problem-solving system and being able to capture and reason about a sufficient subset of its behaviors to make the DM useful.

§6. How Domain Independent is this Work?

Do the characteristics unique to the DVMT system, its domain and its architecture, influence our approach to such a degree as to make this system inapplicable to other domains (i.e., other problem-solving systems or other task domains, such as medicine or hardware)? We believe the answer is no. Although we have not tried to apply our approach to another domain, we believe that it can in fact be extended to other domains, which do not necessarily have the unique characteristics of the DVMT system. We can look at two aspects of our system with respect to domain independence: the reasoning mechanisms and the formalism

of the system model.

We believe that reasoning mechanisms developed here are equally applicable in other domains. Some of these (Comparative Reasoning and the ability to reason about Underconstrained Objects) developed as a result of some unique characteristics of the DVMT but we feel any diagnostic system can benefit from these types of reasoning. Furthermore, the ideas developed here for controlling the search can also be applied to other domains. We also believe that the modeling formalism is applicable to other domains. The techniques we developed for parametrizing system behavior in order to represent it more efficiently could be used for any complex system.

Does our assumption that most of the intermediate problem-solving states are available limit the applicability of our system? We think not. Many other diagnostic systems have dealt with the type of reasoning necessary given a black-box view of the system (Genesereth's DART [15] and Davis' digital circuit analyzer [10] systems deal mainly with the problems associated with this view), and our formalism certainly supports this type of reasoning. What our assumption allowed us to do was to get beyond these reasoning mechanisms and explore other interesting problems associated with diagnosing the behavior of complex systems.

How much of the modeling formalism depends on the domain of the DVMT system? The domain of the DVMT system, acoustic signal interpretation, has not influenced the modeling formalism or the reasoning. Although clearly the attributes of the objects as well as the functions evaluating these attributes reflect the characteristics of the domain, the reasoning apparatus is completely indepen-

dent as are the reasoning strategies. Of course, this is not saying that it would be trivial to construct a model of another domain. Most of the the hard work is left to the model constructor but we hope that the guidelines laid out in Chapter VI would make the task much easier than it was for us.

§7. Where Do We Go from Here?

We can choose to go in one of three directions:

- We can continue exploring the area of diagnosis. Adding new reasoning types as necessary and expanding the model in order to handle more cases.
- We can choose one of the many incompletely solved problems and explore it in detail. This would involve more formal work and the development of theories of some particular area.
- We can switch to the area of detection and correction and try and build a fully fault-tolerant problem-solving system.

The next, and final, chapter in this dissertation discusses the various topics for future research as well the initial ideas we have had about some of them.

Chapter x

FUTURE RESEARCH

This fills my head with ideas.
only I don't exactly know what they are.
- Lewis Carroll, in *Through the Looking Glass*

We have posed many questions and provided few answers. There are many areas of this work that remain unexplored and many directions for future work. This chapter focuses first on the "pie-in-the-sky" type of goals. This brings us back to the original motivation for this research: fully fault tolerant, adaptive systems. In the first part of this chapter, a possible architecture of such a system is briefly described. The second part of this chapter focuses on the many forward references made throughout the dissertation regarding extensions to the Diagnosis Module (DM).

§1. The Best of All Possible AI Systems

Let us come back briefly to the original motivation for this work. We discussed the need for more autonomous systems, ones which could function in an independent manner, unsupervised by human overseers. Specifically, we were interested in autonomous knowledge-based systems. Such systems would of course

not only be competent in their domain of expertise, but could also monitor their behavior, detect when it was inappropriate, and take corrective action as necessary. This corrective action might mean repairing a failed component, correcting wrong code, reorganizing the system, or adding more knowledge to its knowledge base. Such systems would then be truly fault-tolerant and could adapt to changing environments, whether external or internal. Two questions arise at this point:

- is it realistic to expect such abilities from a machine, and
- is it necessary for machines to exhibit such abilities?

At this point in time, the answer to the first question is certainly "no". However, in view of the expectations being posed on computers, the answer to the second question will soon be "yes". Even today, currently unrealistic projects are being financed, whose realization will require a "society of machines" to handle the complexity.

Suppose we were asked to design an autonomous knowledge-based (AKB) system; what would its architecture be? Figure 78 illustrates a possible architecture for such a system. The system would consist of two parts: the domain expertise module and the monitoring module. The task of the domain expertise module would be the actual problem solving. The task of the monitoring module would be to oversee the functioning of the domain expertise module and make corrections to its code, its control parameters, or its knowledge-base as necessary, to maintain its proper functioning. The construction of such a system would begin with

AUTONOMOUS KNOWLEDGE-BASED SYSTEM ARCHITECTURE

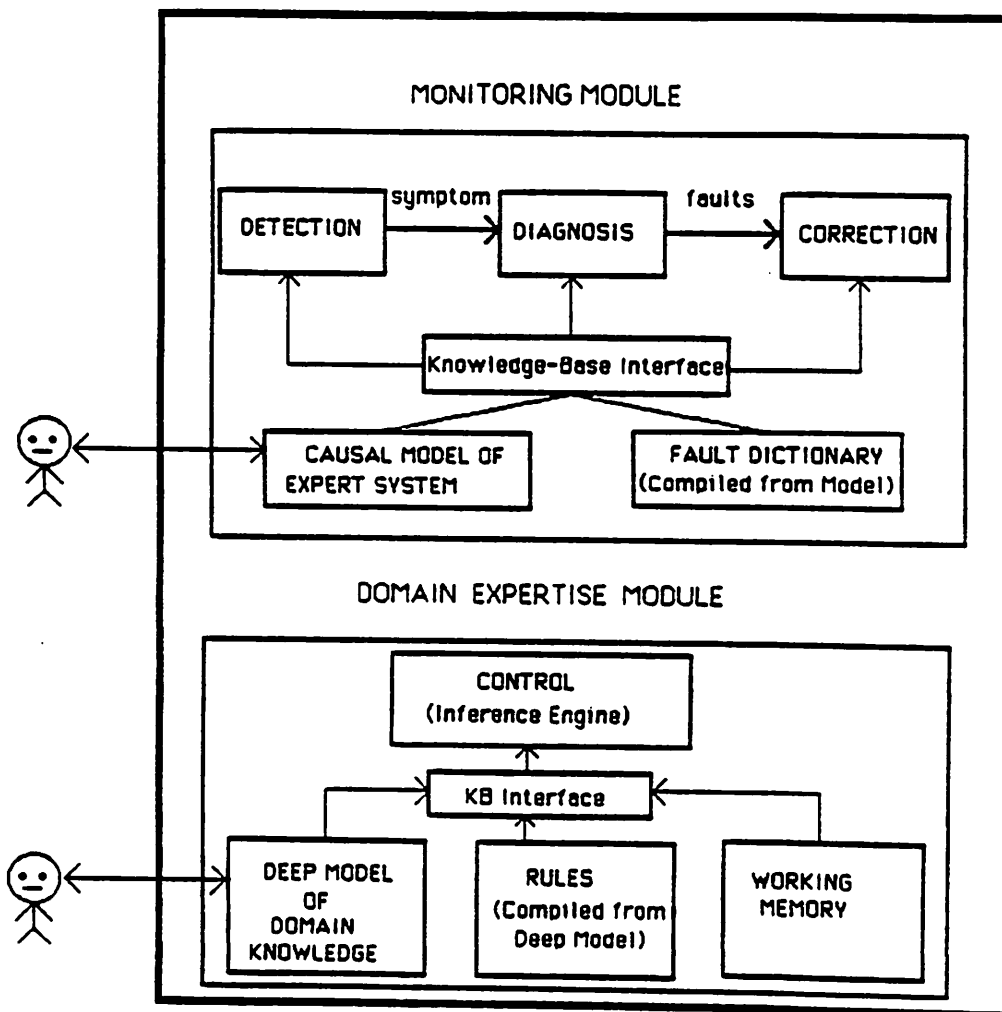


Figure 78: A Possible Architecture of an Autonomous Knowledge-Based System

This figure shows a possible architecture for a fully autonomous knowledge-based system. Such system would have a monitoring module that would oversee its problem-solving activities and detect, diagnose, and correct any inappropriate behaviors.

a system designer and a domain expert. The system designer would construct a high level specification of the AKB architecture. This high level specification would serve two functions. First, it would be the input to an automatic program generator that would then produce the actual code for the AKB. Second, this high level specification would constitute the causal model that would serve as the basis for diagnosis, detection, and correction of the domain expertise module. The domain expert would construct (or help construct) a similarly structured model of the domain which would serve as the deep model knowledge base for the domain expertise module.

Once these design steps were finished and the system code was generated, the AKB would begin its normal functioning. Its two subsystems would work in parallel: the domain expertise module working on whatever the current problem-solving task was, the monitoring module overseeing the performance of the domain expertise module and making corrections as necessary. These corrections could come in three different forms.

Faults in domain KB code. These faults would be corrected by first making the appropriate change to the high-level specifications of the domain expertise module, which would then be given to the automatic program generator that would produce the corrected code. Since the high-level specification also serves as the causal model for the monitoring module, no special updating of that model would be necessary. The corrected specification would simply replace the previous version.

Faults in the hardware components. These faults would be corrected by replacing/repairing the faulty component or by reconfiguring the system so that the faulty component would be ignored.

Faults in the domain expertise module's control parameters. These faults would be corrected by finding the appropriate value for each faulty parameter and resetting the parameter.

Faults in the domain knowledge. These faults would be corrected by consulting the domain expert and updating the domain model.

As the AKB would begin to function, the rule builder component would begin to assemble two sets of rules from the two causal models: the model of the domain expertise module and the model of the domain knowledge. One set of rules would represent the frequently seen associations in the domain, thus efficiently representing the knowledge contained in the causal model of the domain for frequently seen problems. The other set of rules would represent the frequently seen associations among observed symptoms of the domain expertise module and the faults. These rule sets would be formed dynamically and would be updated by the rule builder component to reflect any changes in the tasks posed to the domain expertise module or its knowledge-base as well as any changes in the functioning of the domain expertise module or its structure. The rule builder would thus construct a set of shallow rules (a fault dictionary in the case of the monitoring module) for quick, efficient problem-solving. Both of the modules could of course

resort to reasoning from first principles by using their underlying causal models.

§2. Extensions to the DM

Throughout the dissertation references were made to as-yet-unimplemented desirable features of the DM. This section discusses in more detail how these features might be implemented. The discussion is divided into four parts, dealing with detection, diagnosis, correction, and distribution of the DM.

§2.1 Detection

Appendix C discusses some issues associated with detection of inappropriate behavior in problem-solving systems. Here we will focus on the implementation details and the interface between the detection and diagnosis components. The main idea is that the same model of the system can be the knowledge-base for both diagnosis and detection. For detection, the existing System Behavior Model would be augmented to maintain information about the occurrence of certain types of events in the DVMT. Some criteria would be associated with each event of interest describing the desirable characteristics of that event. The monitor module of the detection component would periodically scan the DVMT data structures, collecting information regarding the occurrence of the events of interest. Whenever the detection criteria were violated, that is, whenever the events did not occur as expected, they would be reported as inappropriate behavior to the DM. The DM would then initiate diagnosis in order to determine why the

events did not occur as expected.

Let us look at a concrete example. Suppose that there was good reason to believe that the problem-solving system should work on a specific part of the problem. In the case of the DVMT, suppose that the system should be working in some area of the environment. This expectation would be part of the detection criteria for a particular experiment in the DVMT. They would be represented in the System Behavior Model in data structures attached to some states in the model.

In this case three states come to mind: states representing the execution of knowledge source instantiations, states representing satisfaction of goals, and states representing the existence of hypotheses. Either one of these states could have a detection criteria record structure associated with it. This record structure, or frame, would encode the expectations for that state and its associated objects, if the system were behaving as expected. For example, in the case of hypothesis states, the detection frame would contain information about the class of hypotheses the DVMT was expected to produce. It would first include user-supplied information characterizing the desired hypotheses. This would include attributes such as the region, the event-class, the level, and the rating of the hypotheses. The detection frame would also include information collected by the monitor regarding the hypotheses produced by the DVMT that satisfy the user-supplied characteristics. If, after some time,¹ the DVMT failed to derive the expected number of the desired hypotheses, the detection component would

¹This time would be supplied by the user.

notify the DM that inappropriate behavior has occurred. The DM would begin diagnosis, in order to determine why the expected hypotheses were not produced by the DVMT.

§2.2 Diagnosis

There are many ways in which the diagnosis component could be improved. This section focuses first on the inconsistency resolving mechanism that would be necessary to decide which of possible candidates was in fact at fault, and then on the extensions to the existing mechanisms that would make the DM capable of working in a more independent fashion.

Inconsistency Resolving. A necessary addition to the reasoning types is the Inconsistency Resolving reasoning discussed in Chapter IV. This reasoning would be invoked anytime the DM would fail to provide a conclusive diagnosis. Recall that in Example II in Chapter V the DM recognized that one of the two sensors was faulty, but could not determine which one. Inconsistency Resolving reasoning would be used in such cases to determine which of possible fault candidates was the real fault. In Example II, for example, Inconsistency Resolving reasoning would examine the data generated by both sensors over time and compare this data to a model of data generated by a functioning sensor. Such a sensor generates well-correlated data, of uniform signal strengths, which can be integrated into spanning tracks. Once the statistics on the sensor behavior were collected, the Inconsistency Resolving reasoning would compare each of the candidate faulty sensors to the model and using some threshold would determine which

of the sensors were faulty. Since this reasoning is basically probabilistic it would require further confirmation in order to be sure of its conclusions. In a system that included a correction component, this could be done by replacing the failed sensor, running the system for some time, and then checking to see if the problem has been fixed. In addition to deciding among candidate faults, Inconsistency Resolving reasoning could also be used to deal with transient hardware failures such as a sensor or a channel that fails intermittently. Such failures are notoriously difficult to deal with and have been largely avoided, both in the more formal hardware fault diagnostic methods and in knowledge-based diagnostic systems.

State Matching Reasoning. State Matching Reasoning is responsible for selecting an object to use as a model for a problem object being investigated during Comparative Reasoning. Currently, this is a relatively simple process where the model object is selected from candidate objects in the DVMT system by applying matching criteria constructed by the model builder. There are many cases where no suitable model object can be found. In these cases it would be useful to either "back up" or "run forward" the DVMT, using the System Behavior Model, in order to reach a system state where an appropriate model could be found for the problem object. Another extension to this type of reasoning would deal with cases where a model object is found, but comparing the problem object to this model does not reveal any failures. It would be useful to study this problem further and try to identify the cases where Comparative Reasoning is really useful. Finally, it would be useful to have the system automatically generate some characteristics of the model object based on the characteristics of the problem object being

diagnosis. In other words, a knowledge source instantiation whose rating is too low to execute in time would create the matching criteria by characterizing the object that it needs to be compared with. Namely, a highly-rated ksi of the same type.

Extend Comparative Reasoning. Currently, Comparative Reasoning involves the comparison of ratings of objects; either knowledge source instantiations or hypotheses ratings. The possible relationships among these objects are less-than, equal-to, and greater-than. It would be useful to extend this type of reasoning to handle more types of objects and different types of relationships. An obvious extension would be to compare the length of tracks (where the relationships would be shorter-than, equal-to, and longer-than). The degree of correlation of the signals comprising these tracks could also be compared. Another attribute to compare is the time it took for some event to occur. For example, the time it took to produce some hypothesis, receive some hypothesis from another node, integrate a group of hypotheses together, or the time it took to satisfy a goal. Finally, with the development of measures of overall quality of processing at a node, the DM could compare the processing at different nodes and try to understand why experiments with different parameters differ. This is a tedious task which must currently be done by the experimenter.

§2.3 Correction

We have discussed two types of failures: hardware component failure and problem solving control failure caused by faulty parameter settings. Assuming

that the faulty components could be repaired or replaced, hardware fault correction is easy: simply replace the faulty component with a functioning one. A more difficult approach, necessary when repair/replacement is not possible, is to reconfigure the system so that the faulty component is eliminated. For example, if a sensor failed, the correction component could rearrange the remaining sensors so that they covered the area of the faulty one. Another example is the rerouting of communication among processors in order to bypass a failed communication channel. These latter solutions are more difficult because they involve a more global view of the system, not just a localized replacement of a failed part.

Correcting faulty parameter settings is similar to the problem of reconfiguring the system. A parameter is not either on or off (faulty or functioning), but can take on many values, depending on what the values of the other parameters are. For example, it may not be a problem that two processors are not communicating, if they each communicate with a third processor which then transmits the necessary messages. Finding a set of consistent values for multiple parameters is a difficult task. The search space of this problem is large and correction then becomes another complex problem-solving task. For most complex systems, this task is highly empirical: some set of parameter values is selected and the system is allowed to run, while being monitored. The monitor then decides whether the newly set parameters have caused improved behavior. If they have not, another setting is tried. This is what happens with therapy in medicine (take two aspirin and call me in the morning), with economy (tighten the money supply and see what happens to inflation), and with interpersonal relationships (buy her flowers

and see if she comes back). Such interplay between correction, monitoring, and detection is characteristic of the overall behavior of adaptive systems. The line between these three activities is blurred and depends largely on what someone decides to call what.

§2.4 Distribution of the Diagnostic Interpreter

The DVMT system being diagnosed by the DM is a distributed problem-solving system. The DM on the other hand is a centralized system, assuming access to all nodes in the DVMT system. A logical next step for the DM would be to distribute it over the nodes in the DVMT system. In other words, each node in DVMT would have its own DM, which would then have access only to the data at that node. Anytime it needed data from other nodes, it would have to communicate with them, thus engaging in cooperative problem-solving much like the underlying DVMT. This would have the usual advantages and disadvantages of distributed problem-solving, such as increased reliability and problems associated with asynchronous processing [5].

The current system was constructed with this distribution in mind and some mechanisms have been included that would make distribution easier. For example, the ability to label any state in the model as a symptom state to be saved for later diagnosis, could be extended to actually send these symptoms as messages to the appropriate node. In fact, a similar process is already simulated in the system and is described in Example II in Chapter V.

§2.5 Availability of the DVMT Intermediate States

One of the assumptions underlying this work is the availability of the intermediate problem-solving states of the DVMT system. This assumption allowed us to bypass reasoning processes usually necessary to diagnose failures in systems that are viewed as a black-boxes, where only the inputs and the outputs are available. How realistic is this assumption? At one level, it seems to be reasonable that a system should maintain a record of its behavior. Most sophisticated systems do maintain such records for just the purpose of "learning from past mistakes"; organisms have memory, society has history. These records are rarely exhaustive and represent facts deemed important by whomever decided that they should be stored. In order to have the intermediate state available, the problem-solving system would similarly have to make choices about what information to save.

§3. Augmenting the System Behavior Model

Since most effort was spent in the development and use of the SBM representation formalism, this was the area where most questions were raised and most problems identified. Much work would be necessary in order to make the model an easily usable and efficient representation of the system behavior. Several specific extensions are discussed below.

§3.1 Extend the Degree of DVMT Behavior Represented in the Model

This is the most obvious and easiest extension of the current work. Parts of the DVMT behavior were not modeled at all. For example, the intricacies of the planning mechanisms, such as creation of subgoals, merging of ksi's, hypotheses, and goals in the DVMT system, clusters of ksi's on the scheduling queues, and the full complexity of the communication among the nodes. These extensions should not require any additions to either the modeling formalism or the DM. Of course, since the formalism has not been shown to be complete, it is possible that modeling some of these processes would in fact not be feasible within the current formalism. This is why extending the coverage of the model and extending the formalism usually go hand in hand. Ultimately, the modeling formalism would become a full-fledged language for describing problem-solving system behavior.

§3.2 Automatic Hierarchy generation

The model is structured into a hierarchy which is based on the biases of the model builder. This hierarchy imposes a search strategy on the reasoning process. The states at the higher levels of the hierarchy are examined first because they are deemed by the model builder to be more likely causes of the symptoms. When the DVMT system behavior changes, a hierarchy constructed according to one set of assumptions may no longer be optimal. A system capable of constructing its own hierarchy, based on the frequency of certain types of faults, would make diagnosis more efficient. This idea is similar to Davis' [11] reasoning with explicit assumptions, although his aim is to make diagnosis more robust rather than more

efficient.

§3.3 Automatic Extension of Model

Unless the model is an exact duplicate of the modeled system, there will always be behaviors that it does not represent and thus there will be faults that it will not be able to detect. How should an adaptive system deal with its own limitations of knowledge? People deal with this problem in one of two ways: either they try to generalize and select a solution for a similar problem encountered in the past, or they acquire new information that allows them to deal with the new problem. A autonomous system should do the same. It could recognize situations for which it did not have sufficient knowledge and either try to find a similar situation in the past, which it has already solved, or ask the domain expert to supply more information to be included in the causal model. For example, any time a true→false would be encountered by the DM with no lower level cluster expanding the events in between, the system would prompt the model builder to supply more information about the underlying problem-solving system. This might involve the automatic construction of the constraint expressions linking the object attributes and the automatic derivation of the information controlling the model instantiation. Research in this area naturally leads into work done in knowledge acquisition.

§3.4 Extending the Type of Knowledge Represented by the Model

Currently, the knowledge about the system represented in the System Behavior Model is at the level of individual events and sequences of such events. It would be interesting to include in the model knowledge both at a lower level of detail and at a higher level. Lower level knowledge might include assumptions about the task domain (for example, the fact that vehicles tend to move in straight lines with certain velocities). Such knowledge would allow the system to detect problems in its domain knowledge base. Eventually, this type of knowledge would allow a system to reason about the validity of the system behavior model itself.

Higher level knowledge might represent overall problem solving states of a single processor, group of processors, or the entire system. This type of knowledge would allow a system to reason about its behavior at a more abstract level and would support globally coherent diagnoses.

§4. Summary

This chapter discussed some directions for future research. The first part of the chapter described a possible architecture of a fully fault-tolerant problem-solving system. The second part focused on the extensions to the DM. As we add more and more sophisticated control and monitoring capabilities to the systems we create, we have to remember to balance "thinking about doing" with actually "doing", lest the systems we create spend all their time monitoring their behavior

and no time doing the actual task; something readily observed in well entrenched bureaucracies.

Bibliography

- [1] Richard S. Brooks. *Experiments in Distributed Problem Solving with Iterative Refinement*. **Ph.D. Dissertation**, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts, September 1982.
- [2] B. Chandrasekaran. *Expert Systems: Matching Techniques to Tasks*. **Artificial Intelligence Applications for Business**, W. Reitman, ed. Ablex Corp.
- [3] E. Charniak and D. McDermott. *Introduction to Artificial Intelligence*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1984, pp. 457-481.
- [4] Daniel D. Corkill, Victor R. Lesser, and Eva Hudlicka. *Unifying data-directed and goal-directed control: An example and experiments*. **Proceedings of the Second National Conference on Artificial Intelligence**, August 1982, pp. 143-147
- [5] Daniel D. Corkill. *A framework for organizational self-design in distributed problem solving networks*. **Ph.D. Dissertation**, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts, February 1983.
- [6] D. Corkill. Personal communication.
- [7] Stephen E. Cross. *An Approach to Plan Justification Using Sensitivity Analysis*. **Sigart**, No. 93, July 1985, pp. 48-55,
- [8] Randall Davis. *Meta-Rules: Reasoning about Control*. **Artificial Intelligence**, 15:179-222, 1980.
- [9] R. Davis. *Expert Systems: Where are we and where do we go from here?*. **AI Magazine**, Vol.3, No. 2, Spring 1982, pp. 3-22.

- [10] R. Davis, H. Shrobe, W. Hanscher, K. Wieckert, M. Shirley, and S. Polit. *Diagnosis based on descriptions of structure and function. Proceedings of the Second National Conference on Artificial Intelligence*, August 1982, pp. 137-142.
- [11] Randall Davis. *Diagnostic Reasoning Based on Structure and Behavior. Artificial Intelligence*, Vol. 24, 1985, pp. 347-410.
- [12] Johann de Kleer and John Seely Brown. *Foundations of Envisioning. Proceedings of the National Conference on Artificial Intelligence*, August 1982, pp. 434-437,
- [13] D.Dörner. *Self Reflection and Problem Solving. In Human and Artificial Intelligence*, F. Klix, editor, North-Holland, NY, 1979.
- [14] L.D. Erman, F. Hayes-Roth, V.R. Lesser, and D.R. Reddy. *The Hearsay-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty. Computing Surveys*, 12(2):213-253, June 1980.
- [15] M. Genesereth. *Diagnosis using hierarchical design models. Proceedings of the National Conference on Artificial Intelligence*, August 1982, pp. 278-283.
- [16] M. Genesereth. *An Overview of Meta-Level Architecture. Proceedings of the National Conference on Artificial Intelligence*, August 1983, pp. 119-124.
- [17] William B. Gevarter. *Artificial Intelligence, expert systems, computer vision, and natural language processing*. Noyes Publications, Park Ridge, New Jersey, 1984.
- [18] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Data Structures*. Computer Science Press, Inc., 1977, page 336.
- [19] Eva Hudlicka and Victor Lesser. *Meta-level control through fault detection and diagnosis. Proceedings of the National Conference on Artificial Intelligence*, August 1984, pp. 153-161.
- [20] Van E. Kelly and Louis L. Steinberg. *The CRITTER system: Analyzing Digital Circuits by Propagating Behaviors and Specifications. Proceedings of the National Conference on Artificial Intelligence*, 1982, pp. 284-289.

- [21] Victor R. Lesser and Lee D. Erman. *Distributed Interpretation: A Model and an Experiment*. **IEEE Transactions on Computers**, Special Issue on Distributed Processing Systems, Vol.C-29, No. 12, December 1980, pp. 1144-1162.
- [22] Victor R. Lesser and Daniel C. Corkill. *Functionally Accurate, Co-operative Distributed Systems*. **IEEE Transactions on Systems, Man and Cybernetics**, Vol.SMC-11, No. 1, January 1981, pp. 81-96.
- [23] Victor Lesser and Daniel D. Corkill. *The Distributed Vehicle Monitoring Testbed: A tool for investigating distributed problem solving networks*. **AI Magazine** 4(3):15-33, Fall 1983.
- [24] Drew McDermott. *Artificial Intelligence Meets Natural Stupidity*. **SIGART**, No. 57, April 1976.
- [25] Drew McDermott and Ruven Brooks. *ARBY: Diagnosis with Shallow Causal Models*. **Proceedings of the National Conference on Artificial Intelligence**, 1982, pp. 370-372.
- [26] Donald Michie. *High-Road and Low-Road Programs*. **AI Magazine** 3(1):21-22, Winter 1981-1982.
- [27] William R. Nelson. *REACTOR: An Expert System for Diagnosis and Treatment of Nuclear Reactor Accidents*. **Proceedings of the National Conference on Artificial Intelligence**, August 1982, pp. 296-301.
- [28] John von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Urbana, Ill., 1966.
- [29] D.A.Norman. *Analysis and Design of Intelligent Systems*. In **Human and Artificial Intelligence**, F. Klix, editor, North-Holland, NY, 1979.
- [30] Ramesh S. Patil, Peter Szolovits, and William B Schwartz. *Causal Understanding of Patient Illness in Medical Diagnosis*. **Proceedings of the Seventh International Joint Conference on Artificial Intelligence**, Vol. 2, 1981, pp. 893-899.
- [31] B. Randell, P.A. Lee, P.C. Treleaven. *Reliability Issues in Computing System Design*. **Computing Surveys**, 10(2):123-165, June 1978.

- [32] Chuck Reiger and Milt Grinberg. *The Declarative Representation and Procedural Simulation of Causality in Physical Mechanisms. Proceedings of the Fifth Joint Conference on Artificial Intelligence*, Vol. 1, August 1977.
- [33] Elaine Rich. *Artificial Intelligence*. McGraw-Hill, Inc., New York, 1983.
- [34] G.D. Ritchie and F.K. Hanna. *AM: A Case Study in AI Methodology. Artificial Intelligence*, 23:249-268, 1984.
- [35] Mark Stefik. *Planning and Meta-Planning (MOLGEN: Part 2). Artificial Intelligence*, 16:141-170, 1981.
- [36] Mark Stefik, Janice Aiking, Robert Balzer, John Benoit, Lawrence Birnbaum, Frederick Hayes-Roth, and Earl Sacerdoti. *The Architecture of Expert Systems. In Building Expert Systems*, F. Hayes-Roth, D. Waterman, and D. Lenat, eds., Addison-Wesley Publishing Company, Inc., Reading Massachusetts, 1983.
- [37] William R. Swartout. *Explaining and Justifying Expert Consulting Programs. Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, Vol. 2, August 1981, pp. 815-823.
- [38] Peter Szolovits and Stephen Pauker. *Categorical and Probabilistic Reasoning in Medical Diagnosis. Artificial Intelligence*, 11:115-144, 1978.
- [39] Sholom M. Weiss, Casimir A. Kulikowski, Saul Amarel, and Aran Safir. *A Model-Based Method for Computer-Aided Medical Decision Making. Artificial Intelligence*, 11:145-172, 1978.
- [40] Cay Weitzman. *Distributed Micro/Minicomputer Systems*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1980.
- [41] David S. Wile. *Program Developments: Formal Explanations of Implementations. Communications of the ACM*, Vol. 26, No. 11, 1983, pp. 902-911.

Appendix A

GLOSSARY

This appendix states the definitions for some of the terms used throughout this dissertation. Definitions marked with an asterisk are quoted from Gevarter's AI survey book [17].

ABSTRACTED OBJECT: In the Diagnosis Module, an object representing some DVMT object. Currently there are objects representing hypotheses, goals, knowledge source instantiations, sensors, and channels.

BACKWARD CAUSAL TRACING (BCT): A type of reasoning used by the DM to determine what were the ultimate causes of some initial symptom state. BCT works by simulating how the DVMT might have arrived at the desired state. It focuses its search by looking for breaks in the desired behavior and thus moves among the different levels of the model hierarchy.

BLACKBOARD: A centralized (within one processor) communication medium used in many problem-solving system architectures as a means of communicating among the different knowledge sources. Blackboards may have different levels of abstraction and a given system may have more than one blackboard. The DVMT for example contains a data blackboard for its hypotheses and a goal blackboard for its goals.

CAUSAL MODEL: A model of a system that captures some of the system's structure and function, also called "deep" models. This is in contrast to

"shallow" models, where the system is viewed largely as a black-box and the knowledge about the system is in the form of associations between outputs and inputs.

CAUSAL PATHWAY: A sequence of causally related states in the System Behavior Model. States are causally related whenever a state can be explained in terms of its preceding states.

COMPARATIVE REASONING (CR): Type of reasoning used by the DM to diagnose why the rating of one object was lower or higher than the rating of another object. CR is unique in that it selects some object from within the DVMT as a model with which to compare the object being investigated. It is used mainly to determine why the rating of some ksi was too low.

CONSTRAINT EXPRESSIONS: Expressions relating the attributes of neighboring objects so that given the values of one object's attributes, all its neighboring objects can be evaluated.

CONTROL KNOWLEDGE: In AI systems, knowledge upon which control decisions are based. This includes knowledge about the system itself, general problem-solving strategies, or some task characteristics that play an important role in determining the problem-solving strategy.

CONTROL PROBLEM: In AI systems, the problem of deciding what to do next; i.e., what task to focus on next or what type of problem-solving strategy to use.

DATA BLACKBOARD: In the DVMT system, a centralized structure for storing hypotheses the system derived so far about the vehicular motion. The data blackboard structure parallels the goal blackboard structure. It has 8 levels, corresponding to the levels of abstraction of the data in the DVMT system:

pattern location (pl)	pattern track (pt)
vehicle location (vl)	vehicle track (vt)
group location (gl)	group track (gt)
signal location (sl)	signal track (st)

DBB: see Data Blackboard.

DIAGNOSIS MODULE: A component of a problem-solving system responsible for diagnosing the system's behavior. The design and construction of such a module is the focus of this dissertation.

DISTRIBUTED PROBLEM-SOLVING SYSTEM: A problem-solving system where a number of autonomous processors cooperate on solving the overall task. Both data and knowledge may be partitioned among the processors.

DM: see Diagnosis Module.

DOMAIN KNOWLEDGE: Knowledge about the particular task a knowledge-based system is asked to solve.

DOMAIN OF A SYSTEM: The specific task for which a (knowledge-based) system has expertise. For example, the domain of the Diagnosis Module is the diagnosis of the DVMT system. The domain of the DVMT system is acoustic signal interpretation.

DVMT OBJECTS: Objects in the DVMT system referred to by the DM. For example, hypotheses, goals, or knowledge source instantiations.

DVMT SYSTEM: The Distributed Vehicle Monitoring Testbed: a distributed problem-solving system designed to experiment with control strategies in distributed problem solving systems.

EXPANDABLE STATE: In the DM, a state that represents a situation whose existence is consistent with the current DVMT parameter settings. For example, the state MESSAGE-RECEIVED is expandable if it refers to an object that can be communicated. Otherwise it is non-expandable and is considered primitive as far as diagnosis is concerned.

EXPERT SYSTEM: See Knowledge-Based System.

EVENT CLASS: In the DVMT system the signal type of the acoustic data. A system signal grammar specifies how the different signal types (event classes) are combined at the individual levels (sl & st, gl & gt, vl & vt, pl & pt) to eventually form pattern tracks.

FAULT-DETECTION/DIAGNOSIS SYSTEM: Any system capable of detecting inappropriate behaviors of another system and diagnosing what caused them.

FAULT DICTIONARY: A collection of empirical associations among observed symptoms and their underlying causes, the faults. These associations are often in the form of production rules and may include probability factors representing the strength of a particular association.

FORWARD CAUSAL TRACING (FCT): A type of reasoning used by the DM to simulate the effect of some situation on future system behavior. Currently used to simulate the effects of an identified fault on the DVMT behavior.

GOAL: In the context of the DVMT system, a description of vehicle motion predicted on the basis of some already derived hypothesis. Where hypotheses represent the results already derived by the system, goals represent classes of hypotheses that could be derived from an existing hypothesis.

GOAL BLACKBOARD: In the DVMT system a centralized structure for storing goals the system has created as predictions of further extensions

of its existing hypotheses. The goal blackboard structure parallels the data blackboard structure.

GBB: see Goal Blackboard.

GRACEFUL DEGRADATION: The ability of a knowledge-based system to continue functioning even when reaching the limits of its task-specific expertise; by using common sense knowledge and general problem solving methods.

HYPOTHESIS: In the context of the DVMT system, a description of some vehicle motion in the environment. A hypothesis is represented by a record structure (frame) and some of its attributes are: level, time-location-list (describing the motion of the vehicle), and event-class (describing the type of signal detected).

INSTANTIATED ABSTRACTED OBJECT: An abstracted object whose attributes have been evaluated so that it represents some specific DVMT object. Such an object might or might not actually exist in the DVMT.

INSTANTIATED SBM: System Behavior Model consisting of instantiated states and abstracted objects which together represent a specific course of events in the DVMT.

INSTANTIATED STATE: A state of the SBM whose attributes have been evaluated. Such a state has an attached object and represents some specific situation in the DVMT. For example, the existence or lack of existence of the attached object.

KNOWLEDGE-BASED SYSTEM: A problem-solving system where the emphasis is on the knowledge the system has about its task rather than on some general problem solving methods.

KNOWLEDGE-SOURCE: A component of a knowledge-based system which encapsulates some small chunk of knowledge, usually in a procedural form. In the context of the DVMT system, knowledge sources are the workhorses of the system and contain the knowledge necessary to interpret the incoming raw data and derive the overall map of the vehicular motion. Originally used in the Hearsay-II speech understanding system.

MODEL: A representation of some system, possibly abstracting away some details.

MODEL OBJECT: In Comparative Reasoning the DVMT object that has been selected as a model for the object being investigated. For example, if investigating why a ksi was low rated, the model object would be a highly rated ksi.

NEIGHBOR: In the System Behavior Model, each state has up to 6 types of neighbors. Successor and predecessor (front and back neighbors) states at the same level of the model hierarchy, as well front and back neighbors at the upper and lower levels.

NON-EXPANDABLE STATE: See Expandable State.

OBJECT GROUPING: The process of grouping similarly-behaving objects under one state to reduce the complexity of the diagnosis. For example, all existing objects would be grouped under one true state, rather than creating a separate state for each object.

PARALLEL STATE/OBJECT: In Comparative Reasoning the state and object with which the current state/object is compared to determine the state's relative value.

PREDICATE STATE: A state in the System Behavior Model which can be either true or false. By convention, false states represent lack of existence of some object in the DVMT system, such as the lack of a desired hypothesis.

PRIMITIVE STATE: A state in the System Behavior Model representing some primitive situation which is considered a primitive cause and can therefore be reported as a fault.

PROBLEM OBJECT: In Comparative Reasoning the object being investigated. For example, a low rated ksi.

PROBLEM-SOLVING CONTROL FAILURE: An inappropriate control parameter setting in a parametrized problem-solving system.

PROBLEM-SOLVING SYSTEM: A computer program that uses knowledge and reasoning techniques to solve problems that normally require the abilities of human experts.

PRODUCTION RULE: A form of knowledge representation in AI. Basically an if-then statement. The left hand side represents some condition. The right hand side represents the action to take when that condition is encountered. "Action" here is used rather loosely to mean anything from adding a fact to a data base of existing facts to actual action in the outside world.

RELATIONSHIP STATE: A state in the System Behavior Model that represents a relationships among the actual values of two object attributes in the DVMT system. The values can be: less-than, equal-to, or greater-than. Relationships states are used in Comparative Reasoning.

RULE: See Production Rule.

SBM: see System Behavior Model.

SYMPTOM: An observed misbehavior of a system. In the case of the DVMT, a symptom is most commonly the lack of some desired hypothesis, represented by a false state, or the abnormally low rating of some object, such as a ksi or a hypothesis.

SPLITTING ATTRIBUTES: In the System Behavior Model, attributes of abstracted objects which may require the creation of multiple abstracted objects when instantiated. For example, the time-location-attribute of a location hypothesis object is a splitting attribute when evaluated using a track hypothesis object, since the number of locations necessary to create some track depends on the track length.

SYSTEM BEHAVIOR MODEL: The model of the DVMT system used by the Diagnosis Module to determine the causes of some symptom.

UNKNOWN VALUE DERIVATION (UVD): In the DM, a type of reasoning used to determine the value of a state from the values of its surrounding states.

Appendix B

THE SYSTEM BEHAVIOR MODEL

§1. Introduction

This appendix contains the System Behavior Model of the DVMT system. There are five model clusters, each corresponding to some aspect of the DVMT system behavior. These model clusters are represented by s-expressions which are read in by the Diagnosis Module.

In order to save space the following omissions were made. Attributes whose values are *nil* are not shown. The *name* and *type* attributes are not shown. In most cases the values of the following three attributes are as shown below:

```
(object-lumping-specs ((
  ( ~ (expandable-state-function echo))
  ((expandable-state-function echo t) (vmt-ids phd:if)))
  (value expandable-state?))
  (value
    (if phd:if
      (path (if select-structure (path self object-ptrs)
        vmt-ids)))
    (expandable-state?
      (if phd:if
        (path (if select-structure (path self object-ptrs)
          expandable-state-function))))
```

These attributes will only be shown if different from the above values. The Communication Ksi Scheduling Cluster is very similar to the Ksi Scheduling Cluster

and will not be shown here.

§2. Answer Derivation Cluster

```

;*****
:      CLUSTER answer-derivation
;*****
:      STATE pt

((f-n      (p:or (p:or (message-exists))
                (p:or (pt)))
(b-n      (p:or
  (p:or (message-accepted))
  (PrefOR
    (p:and (pt))
    (p:and (pl)))
  (p:or (vt))))
(f-l-n      (p:or (stimulus-hyp)(necessary-hyp)))
(object-ptrs      pt-hyp-ob)
(object-to-use      ((((* hyp)(* hyp-ob)) new)
  ((necessary-hyp (* hyp-ob)) same)
  ((message-exists message-ob) new)
  ((stimulus-hyp (* hyp-ob)) same))))

:OBJECT pt-hyp-ob

((vmt-ids (f phd:find-track-hyps
  (path self time-location-list)
  (path self event-class)
  (path self node)
  pt))
(expandable-state-function
  (f phd:in-ia?
    (path self time-location-list)
    (path self event-class)
    (path self node)
    pt))
(s-attributes      (((pt pt-hyp-ob) (time-location-list))))
(attribute-evaluation-sets (

```

```

      (time-location-list event-class level node rating)))
(context-dependent-attributes
  (event-class s-attributes time-location-list))
(time-location-list
  ((pt message-ob) (path x1 tll/trl))
  ((pt vt-hyp-ob) (path x1 time-location-list))
  ((pt pl-hyp-ob) (path x1 time-location-list))
  ((pt pt-hyp-ob) (f create-track-segments
    (path x1 time-location-list)
    (path x1 event-class)
    pt
    (path x1 node))))))
(event-class ((pt message-ob) (path x1 event-classes))
  ((pt vt-hyp-ob)
  (f phd:higher-level-event-classes
    vt
    (path x1 event-class)))
  ((pt pt-hyp-ob) (path x1 event-class))
  ((pt pl-hyp-ob) (path x1 event-class)))
(level pt)
(node (path x1 node))
:
:-----
: STATE pl

((f-n (p:or
  (p:or (pt))
  (p:or (message-exists))))))
(b-n (p:or
  (p:or (vl))
  (p:or (message-accepted))))
(f-l-n (p:or (stimulus-hyp)(necessary-hyp)))
(object-ptrs pl-hyp-ob)
(object-to-use ((((* hyp)(* hyp-ob)) new)
  ((necessary-hyp (* hyp-ob)) same)
  ((message-exists message-ob) new)
  ((stimulus-hyp (* hyp-ob)) same))))

:OBJECT pl-hyp-ob

((vmt-ids (f phd:find-hyps

```

```

    (path self time-location-list)
    (path self event-class)
    (path self node)
    pl))
(expandable-state-function (if phd:in-ia?
    (path self time-location-list)
    (path self event-class)
    (path self node)
    pl))
(s-attributes      (((pl pt-hyp-ob) (time-location-list))))
(attribute-evaluation-sets (
    (time-location-list event-class level node rating)))
(context-dependent-attributes
    (event-class s-attributes time-location-list))
(time-location-list (((pl message-ob) (path x1 tll/trl))
    ((pl pt-hyp-ob) (path x1 time-location-list))
    ((pl vl-hyp-ob) (path x1 time-location-list))
    ))
(event-class ( ((pl message-ob) (path x1 event-classes))
    ((pl pt-hyp-ob) (path x1 event-class))
    ((pl vl-hyp-ob)
    (if phd:higher-level-event-classes
        vl
        (path x1 event-class) ))))
(level          pl)
(node          (path x1 node)))

```

The states VT, GT, and ST are analogous to state PT and will not be shown here.

The states VL and GL are analogous to PL and will not be shown here.

```

:
:-----
:          STATE s1

```

```

((f-n          (p:or
    (p:or (message-exists))
    (p:or (gl))
    (p:or (st))))
(b-n          (p:or (sensed-value)
    (p:or (message-accepted))

```

```

    ))
(f-l-n      (p:or (stimulus-hyp)(necessary-hyp)))
(object-ptrs      sl-hyp-ob)
(object-to-use    ((((* hyp)(* hyp-ob)) new)
  ((necessary-hyp (* hyp-ob)) same)
  ((message-exists message-ob) new)
  ((stimulus-hyp (* hyp-ob)) same))))

;OBJECT sl-hyp-ob

((vmt-ids
  (f phd:find-hyps (path self time-location-list)
    (path self event-class)
    (path self node)
    sl))
(expandable-state-function (f phd:in-ia?
  (path self time-location-list)
  (path self event-class)
  (path self node)
  sl))
(s-attributes      (((sl st-hyp-ob) (time-location-list))
  ((sl gl-hyp-ob) (event-class))))
(attribute-evaluation-sets (
  (time-location-list event-class level node rating)))
(context-dependent-attributes
  (event-class s-attributes time-location-list))
(time-location-list
  (((sl st-hyp-ob) (path x1 time-location-list))
  ((sl gl-hyp-ob) (path x1 time-location-list))
  ((sl message-ob) (path x1 tll/trl))
  ((sl sensed-value-ob) (path x1 time-location))))
(event-class
  (((sl st-hyp-ob) (path x1 event-class))
  ((sl sensed-value-ob) (path x1 event-class))
  ((sl message-ob) (path x1 event-classes))
  ((sl gl-hyp-ob)
    (f phd:lower-level-event-classes
      gl
      (path x1 event-class) ))))
(level      sl)
(node      (path x1 node)))

```

```

:
:-----
:      STATE sensed-value

((f-n      (p:and (sl)))
(b-n      (p:and (data-exists)(sensor-ok)))
(object-lumping-specs ((
      ( ~ (node echo))
      ( ~ (sensor-id echo))
      ( ~ (expandable-state-function echo))
      ((expandable-state-function echo t) (vmt-ids phd:if))
      )
(value expandable-state?)))
(object-ptrs      sensed-value-ob)
(object-to-use      ((((* hyp)(* hyp-ob)) new)
      ((sensor-ok sensor-ob) new)
      ((data-exists data-ob) new))))

:OBJECT sensed-value-ob

((vmt-ids      (if phd:find-sensed-values
      (path self sensor-id)
      (path self time-location)
      (path self event-class)
      (path self node)))
(expandable-state-function      (
      ((sensed-value sensor-ob)
      (if sensed-value-in-ia?
      (path self sensor-id)
      (path self time-location)
      (path self event-class)
      (path self node)))
      ((sensed-value data-ob)
      (if sensed-value-in-ia?
      (path self sensor-id)
      (path self time-location)
      (path self event-class)
      (path self node)
      ))
      ))
      ((sensed-value sl-hyp-ob)
      (if sensed-value-in-ia?

```

```

(path self sensor-id)
(path self time-location)
(path self event-class)
(path self node)
))
  ((sensed-value-rating rating-hyp-ob)
   (f phd:find-sensed-values
    (path self sensor-id)
    (path self time-location)
    (path self event-class)
    (path self node)))
  ))
(s-attributes
 ((sensed-value sl-hyp-ob) (sensor-id))
 ((sensed-value-rating rating-hyp-ob) (sensor-id))
 ((sensed-value data-ob) (node sensor-id))
 ((sensed-value sensor-ob) (node vmt-ids))
 ))
(attribute-evaluation-sets (
  ((sensed-value sl-hyp-ob)
   ((event-class sensor-id time-location node)
    (vmt-ids)
    (value)))
  ((sensed-value-rating rating-hyp-ob)
   ((event-class sensor-id time-location node)
    (vmt-ids)
    (value)))
  ((sensed-value data-ob)
   ((event-class node time-location)
    (sensor-id)
    (vmt-ids)
    (value)))
  ((sensed-value sensor-ob)
   ((event-class sensor-id time-location node)
    (vmt-ids)
    (value)))
 ))
(context-dependent-attributes
 (attribute-evaluation-sets
  event-class expandable-state-function node
  s-attributes sensor-id

```

```

    time-location))
(time-location
  (((sensed-value sl-hyp-ob)
    (path x1 time-location-list))
   ((sensed-value-rating rating-hyp-ob)
    (path x1 time-location-list))
   ((sensed-value data-ob) (path x1 time-location))
   ((sensed-value sensor-ob)
    (if construct-time-regions
      (path x1 time-regions)
      (path x1 nodes))))
  ))
(event-class
  (((sensed-value sl-hyp-ob) (path x1 event-class))
   ((sensed-value-rating rating-hyp-ob) (path x1 event-class))
   ((sensed-value data-ob) (path x1 event-class))
   ((sensed-value sensor-ob)
    (path x1 event-classes))
  ))
(sensor-id (
  ((sensed-value sl-hyp-ob)
   (if find-sensors-sensing-region
     (path self time-location)
     (path self event-class)
     (path self node)))
   ((sensed-value-rating rating-hyp-ob)
    (if find-sensors-sensing-region
      (path self time-location)
      (path self event-class)
      (path self node)))
   ((sensed-value data-ob)
    (if find-sensors-sensing-region
      (path self time-location)
      (path self event-class)
      (path self node)))
   ((sensed-value sensor-ob)
    (path x1 vmt-ids))))
(value (if get-sensed-value
  (path self sensor-id)
  (path self vmt-ids)))
(node (

```



```

((sensed-value sl-hyp-ob)
 (path x1 node))
((sensed-value-rating rating-hyp-ob)
 (path x1 node))
((sensed-value data-ob)
 (f find-nodes-to-process-data
  (path x1 time-location)
  (path x1 event-class)))
((sensed-value sensor-ob)
 (f select:sensor:nodes
  (path x1 vmt-ids))))))

:
:-----
:          STATE sensor-ok

((name          sensor-ok)
 (type          sensor)
 (f-n          (p:and (sensed-value)))
 (object-lumping-specs ((
  ( ~ (expandable-state-function echo))
  ((expandable-state-function echo t) (vmt-ids phd:if))
  )
 (value expandable-state?)))
 (value
  (f sensor-consistent? (path self object-ptrs)))
 (expandable-state?
  (f phd:if
   (path (f select-structure (path self object-ptrs))
    expandable-state-function)))
 (object-ptrs          sensor-ob)
 (object-to-use        (
  ((sensed-value sensed-value-ob) new)
  ))
 (primitive?          t))

:OBJECT sensor-ob

((name          sensor-ob)
 (vmt-ids
  ((sensor-ok sensed-value-ob) (path x1 sensor-id))

```

```

      ((sensor-weight sensed-value-ob) (path x1 sensor-id))))
(f1-attributes      (time-regions event-classes nodes))
(expandable-state-function t)
(attribute-evaluation-sets
  ((vmt-ids event-classes nodes time-regions weight)))
(context-dependent-attributes      (vmt-ids))
(time-regions      (f get-sensor-region
  (path self vmt-ids)))
(event-classes      (f get-sensor-event-classes
  (path self vmt-ids)))
(weight (f phd:get-sensor-weight
  (path self vmt-ids)
  (path x1 time-location)
  (path x1 event-class)
  ))
(nodes      (f select:sensor:nodes
  (path self vmt-ids))))

```

```

:

```

```

:-----

```

```

:

```

```

STATE data-exists

```

```

((f-n      (p:and (sensed-value)))
(object-ptrs      data-ob)
(object-to-use      (((sensed-value sensed-value-ob) new)))
(primitive?      t))

```

```

;OBJECT data-ob

```

```

((f1-attributes      (time-location event-class))
(expandable-state-function t)
(attribute-evaluation-sets
  ( (time-location event-class sensed-values)))
(time-location      (path x1 time-location))
(event-class      (path x1 event-class))
(sensed-values      (f calculate-signal-value
  (path x1 value)
  (f phd:get-sensor-weight
    (path x1 sensor-id)
    (path x1 time-location)
    (path x1 event-class))))))

```

§3. Ksi Scheduling Cluster

```

;*****
;      CLUSTER KSI scheduling
;*****
;
;      STATE stimulus-hyp
;
((f-n      (p:or (goal-created)))
(b-u-n      (p:or (f make-hyp-state-name
  (path (f select-structure (path self object-ptrs)) level))))
(object-ptrs      hyp-ob)
(object-to-use
  ((((* hyp)(* hyp-ob)) same)
  ((goal-created goal-ob) (path x1 stimulus-hyps))))
;
; OBJECT hyp-ob (used by stimulus-hyp, necessary-hyp)
;
((specific-name      (f make-hyp-ob-name (path self level)))
(vmt-ids
  (f find-track/loc-hyps
    (path self time-location-list)
    (path self event-class)
    (path self node)
    (path self level)
  ))
(expandable-state-function
  (f phd:in-ia?
    (path self time-location-list)
    (path self event-class)
    (path self node)
    (path self level)
  ))
(s-attributes
  (((stimulus-hyp goal-ob) (event-class))))
(attribute-evaluation-sets
  ((time-location-list event-class level node rating)))
(context-dependent-attributes
  (event-class level s-attributes

```

```

                                time-location-list))
(time-location-list  ((stimulus-hyps ksi-ob)
  (f get-stim-hyp-tll x1))

((necessary-hyps ksi-ob)
  (f get-necessary-hyp-tll x1))
)
)
(event-class  (((stimulus-hyps ksi-ob)
  (f get-stim-hyp-event-class x1)
)
(necessary-hyps ksi-ob)
  (f get-necessary-hyp-event-class x1)))
(level  ((stimulus-hyps ksi-ob)
  (f get-stim-hyp-level x1))
  ((necessary-hyps ksi-ob)
  (f get-necessary-hyp-level x1))))
(node  (path x1 node)))
:
:-----
:
:                                STATE goal-created

((f-n  (p:or (ksi-rated)))
(b-n  (p:or (stimulus-hyp)))
(object-ptrs  goal-ob)
(object-to-use  (((stimulus-hyp (* hyp-ob)) new)
  ((ksi-rated ksi-ob) (path x1 stimulus-goals)))))

:OBJECT goal-ob

((vmt-ids
  (f phd:find-goals
    (path self active-trl)
    (path self inactive-trl)
    (path self event-classes)
    (path self level)
    (path self goal-type)
  ))
(expandable-state-function
  (f phd:goal-in-ia?
    (path self active-trl)

```

```

    (path self inactive-trl)
    (path self event-classes)
    (path self level)
    (path self node)
  ))
(s-attributes
  (((goal-created (* hyp-ob)) (goal-type))
   ((goal-created (* hyp-ob)(* hyp-ob)) (goal-type))))
(attribute-evaluation-sets
  ((goal-type node stimulus-hyps output-hyps)
   (event-classes level)
   (active-trl inactive-trl)))
(context-dependent-attributes
  (output-hyps s-attributes stimulus-hyps ))
(active-trl      (f get-goal-atrl
  (path self goal-type)
  (path self stimulus-hyps)
  (path self event-classes)))
(inactive-trl    (f get-goal-itrl
  (path self goal-type)
  (path self stimulus-hyps)
  (path self event-classes)))
(event-classes   (f get-goal-event-classes
  (path self goal-type)
  (path self stimulus-hyps)))
(level           (f get-goal-level
  (path self goal-type)
  (path self stimulus-hyps)))
(goal-type       (f get-goal-type
  (path self stimulus-hyps)
  (path self output-hyps)))
(stimulus-hyps   (((goal-created (* hyp-ob))
x1)
  ((goal-created (* hyp-ob)(* hyp-ob))
x1)
  ((stimulus-goals ksi-ob)
(path x1 stimulus-hyps))))
(output-hyps     (((goal-created (* hyp-ob)) ~)
  ((goal-created (* hyp-ob)(* hyp-ob)) x2)
  ((stimulus-goals ksi-ob)
(path x1 output-hyps))))

```

```

(node      (path x1 node)))
:
:-----
:
:                STATE necessary-hyp

((f-n      (p:or (ksi-rated)))
(b-u-n
  (p:or (f make-hyp-state-name
    (path (f select-structure
      (path self object-ptrs) level))))
(object-ptrs      hyp-ob)
(object-to-use    (((* hyp)(* hyp-ob)) same)
  ((ksi-rated ksi-ob) (path x1 necessary-hyps))))

:
:-----
:
:                STATE ks-exists

((f-n      (p:or (ksi-rated)))
(object-lumping-specs ((
  ( ~ (expandable-state-function echo)
  ((expandable-state-function echo t) (f phd:ks-exists?))
  )
(value expandable-state?)))
(value (f phd:ks-exists?
  (f select-structure (path self object-ptrs))))
(expandable-state?    t)
(object-ptrs          ksi-ob)
(object-to-use        (((ksi-rated ksi-ob) same)))
(primitive?          t))

:
:-----
:
:                STATE KSI-rated

((f-n      (p:or (ksi-scheduled)))
(b-n      (p:and (ks-exists)
  (goal-created)
  (necessary-hyp)))
(f-l-n    (p:and (ksi-rating-ok)))
(object-lumping-specs ((

```

```

      ( - (ks-type echo))
    ( - (expandable-state-function echo))
    ((expandable-state-function echo t) (f ksi-rated?)))
  (value expandable-state?))
  (value
    (f ksi-rated? (f select-structure (path self object-ptrs))))
  (object-ptrs      ksi-ob)
  (object-to-use    (((goal-created goal-ob) new)
                    ((necessary-hyp (* hyp-ob)) new)
                    ((ksi-scheduled ksi-ob) same)
                    ((ks-exists ksi-ob) same))))
  :OBJECT ksi-ob

```

```

((vmt-ids          (f phd:find-ksis
                   (path self stimulus-hyps)
                   (path self output-hyps)
                   (path self necessary-hyps)
                   (path self ks-type)))
  (aux-object-attributes  (((ksi-rated goal-ob (* hyp-ob))
                            (necessary-hyps))
                          ((ksi-rated goal-ob)
                            (necessary-hyps))
                          ((ksi-rated (* hyp-ob) (* hyp-ob))
                            (stimulus-hyps stimulus-goals))
                          ((ksi-rated (* hyp-ob))
                            (stimulus-hyps stimulus-goals)))))
  (expandable-state-function t)
  (s-attributes  (((ksi-rated goal-ob (* hyp-ob)) (ks-type))
                 ((ksi-rated goal-ob) (ks-type))
                 ((ksi-rated (* hyp-ob) (* hyp-ob)) (ks-type))
                 ((ksi-rated (* hyp-ob)) (ks-type))))
  (context-dependent-attributes
    (attribute-evaluation-sets aux-object-attributes
      ks-type necessary-hyps output-hyps s-attributes
      stimulus-goals stimulus-hyps))
  (attribute-evaluation-sets (
    ((ksi-rated goal-ob)
      ((node stimulus-hyps stimulus-goals output-hyps ks-type)
        (necessary-hyps)
        (vmt-ids rating)))
    ((ksi-rated goal-ob (* hyp-ob))

```

```

      ((node stimulus-hyps stimulus-goals output-hyps ks-type)
(necessary-hyps)
(vmt-ids rating))
      ((ksi-rated (* hyp-ob))
      ((necessary-hyps node output-hyps ks-type)
(stimulus-hyps stimulus-goals)
(vmt-ids rating))
      ((ksi-rated (* hyp-ob) (* hyp-ob))
      ((necessary-hyps node output-hyps ks-type)
(stimulus-hyps stimulus-goals)
(vmt-ids rating))))))
(node (path x1 node)
(stimulus-hyps ((ksi-rated goal-ob)
(path x1 stimulus-hyps))
((ksi-rated message-ob) x1)
((ksi-rated (* hyp-ob) (* hyp-ob))
(g hyp-ob))
((ksi-rated (* hyp-ob))
(g hyp-ob))
((ksi-rated goal-ob (* hyp-ob))
(path x1 stimulus-hyps))))
(output-hyps (((ksi-rated goal-ob) ~)
((ksi-rated message-ob) x1)
((ksi-rated goal-ob (* hyp-ob)) x2)
((ksi-rated (* hyp-ob) (* hyp-ob)) x2)
((ksi-rated (* hyp-ob) ~))))
(stimulus-goals (((ksi-rated goal-ob) x1)
((ksi-rated message-ob) ~)
((ksi-rated goal-ob (* hyp-ob)) x1)
((ksi-rated (* hyp-ob) (* hyp-ob))
(g goal-ob))
((ksi-rated (* hyp-ob))
(g goal-ob))))
(necessary-hyps (((ksi-rated goal-ob)
(g hyp-ob))
((ksi-rated message-ob) ~)
((ksi-rated goal-ob (* hyp-ob))
(g hyp-ob))
((ksi-rated (* hyp-ob) (* hyp-ob)) x1)
((ksi-rated (* hyp-ob) x1))))
(ks-type (((ksi-rated goal-ob)

```



```
(f find-ks-type/goal
(f select-structure
  (path (f select-structure
    (path self object-ptrs)
stimulus-hyps))
  level))))
(object-lumping-specs ((
  ( ~ (expandable-state-function echo))
  ((expandable-state-function echo t) (f ksi-executed?)))
(value expandable-state?))
(value (f ksi-executed?
  (f select-structure (path self object-ptrs))))
(object-ptrs ksi-ob)
(object-to-use ((((* hyp)(* hyp-ob)) new)
  ((ksi-scheduled ksi-ob) same))))
```

§4. Communication Cluster

```

:*****
:      CLUSTER communication
:*****

:
:      STATE message-exists
:
:
((f-n      (p:or (message-sent)))
(b-n
  (p:or (f make-hyp-state-name
    (path (f select-structure (path self object-ptrs)) level))))
(f-l-n      (p:or (comm-ksi-rated)))
(object-lumping-specs ((
  ( - (to-node echo))
  ( - (expandable-state-function echo))
  ((expandable-state-function echo t) (vmt-ids phd:if))
)
(value expandable-state?)))
(value
  (f phd:if
    (path (f select-structure (path self object-ptrs))
      vmt-ids)))
(object-ptrs      message-ob)
(object-to-use      (((* hyp)(* hyp-ob)) new)
  ((goal-created goal-ob) new)
  ((comm-ksi-executed comm-ksi-ob) new)
  ((message-sent message-ob) same))))
:
:-----
:      OBJECT message-ob
:
((vmt-ids
  (f phd:find-vmt-messages
    (path self tll/trl)
    (path self event-classes)
    (path self level)
    (path self message-type)
    (path self to-node)
    (path self from-node)

```

```

    (path self node)
  ))
(expandable-state-function
  (if phd:in-comm-area?
    (path self tll/trl)
    (path self event-classes)
    (path self level)
    (path self message-type)
    (path self to-node)
    (path self from-node)
    (path self node)
  ))
(s-attributes (
  ((message-accepted (* hyp-ob)) (from-node))
  ((message-exists (* hyp-ob)) (to-node ))
  ((message-sent comm-ksi-ob) (to-node ))
))
(attribute-evaluation-sets (
  ((message-sent message-ob)
   ((message-type tll/trl event-classes from-node
    level node rating to-node)))
  ((message-sent comm-ksi-ob)
   ((message-type tll/trl event-classes from-node
    level node rating to-node)))
  ((message-received message-ob)
   ((message-type tll/trl event-classes from-node
    level node rating to-node)))
  ((message-accepted (* hyp-ob))
   ((from-node message-type event-classes to-node level node rating)
    (tll/trl)))
  ((message-exists (* hyp-ob))
   ((message-type event-classes level from-node node rating to-node)
    (tll/trl))))))
(context-dependent-attributes
  (attribute-evaluation-sets event-classes
    from-node level node
    s-attributes tll/trl to-node))
(message-type (if find-message-type x1))
(tll/trl
  ( ((message-sent message-ob) (path x1 tll/trl))
    ((message-sent comm-ksi-ob) (path (path x1 message) tll/trl))

```

```

((message-received message-ob) (path x1 tll/trl))
((message-accepted (* hyp-ob)) (path x1 time-location-list))
((message-exists (* hyp-ob)) (path x1 time-location-list))
((message-exists comm-ksi-ob) ~); express as f of message-ob))
(event-classes
  ((message-sent message-ob) (path x1 event-classes))
  ((message-sent comm-ksi-ob)
   (path (path x1 message) event-classes))
  ((message-received message-ob) (path x1 event-classes))
  ((message-accepted (* hyp-ob)) (path x1 event-class))
  ((message-exists (* hyp-ob)) (path x1 event-class))
  ((message-exists comm-ksi-ob) ~)))
(level (
  ((message-sent message-ob) (path x1 level))
  ((message-sent comm-ksi-ob)
   (if get-ks-level (path x1 ks-type)))
   ((message-received message-ob) (path x1 level))
  ((message-accepted (* hyp-ob)) (path x1 level))
  ((message-exists (* hyp-ob)) (path x1 level))
  ((message-exists comm-ksi-ob)
   (if get-ks-level (path x1 ks-type))))))
(node (
  ((message-sent message-ob) (path x1 from-node))
  ((message-sent comm-ksi-ob) (path x1 node))
   ((message-received message-ob) (path x1 to-node))
  ((message-accepted (* hyp-ob)) (path x1 node))
  ((message-exists (* hyp-ob)) (path x1 node))
  ((message-exists comm-ksi-ob) (path x1 node))))
(to-node
  ( ((message-sent message-ob) (path x1 node))
    ((message-sent comm-ksi-ob) (if find-receiving-nodes
    (path self node)))
    ((message-received message-ob) ~)
    ((message-accepted (* hyp-ob)) ~)
    ((message-exists (* hyp-ob)) (if find-receiving-nodes
    (path self node)))
    ((message-exists comm-ksi-ob) (if find-receiving-nodes
    (path self node))))))
(from-node
  ((message-sent message-ob) ~)
  ((message-sent comm-ksi-ob) ~)

```

```

((message-received message-ob) (path x1 node))
((message-accepted (* hyp-ob)) (f find-sending-nodes
  (path self node)))
((message-exists message-ob) ~)
((message-exists comm-ksi-ob) ~) )))
:
:-----
:
:                               STATE message-sent
:
((f-n      (p:or (message-received)))
(b-n      (p:or (message-exists)))
(b-l-n    (p:or (comm-ksi-executed)))
(node-transition-neighbors (message-received))
(object-lumping-specs ((
  ( ~ (to-node echo))
  ( ~ (expandable-state-function echo))
  ((expandable-state-function echo t) (vmt-ids phd:if)))
(value expandable-state?)))
(value
  (f message-sent?
    (path (f select-structure (path self object-ptrs)) vmt-ids)
    (path (f select-structure (path self object-ptrs)) node)))
(object-ptrs      message-ob)
(object-to-use    ((message-exists message-ob) same)
((message-received message-ob) new)
((comm-ksi-executed comm-ksi-ob) new))))
:
:-----
:
:                               STATE message-received
:
((f-n      (p:or (message-accepted)))
(b-n      (p:and (message-sent) (channel-ok)))
(f-l-n    (p:or (comm-ksi-rated)))
(node-transition-neighbors (message-sent))
(object-lumping-specs ((
  ( ~ (from-node echo))
  ( ~ (expandable-state-function echo))
  ((expandable-state-function echo t) (vmt-ids phd:if)))
(value expandable-state?)))
(value
  (f message-received?
    (path (f select-structure (path self object-ptrs))

```

```

vmt-ids)))
(object-ptrs      message-ob)
(object-to-use    ((message-sent message-ob) new)
((message-accepted message-ob) same))))
:
:-----
:
:                STATE message-accepted

((f-n            (p:or
  (f make-hyp-state-name
    (path (f select-structure (path self object-ptrs)) level))
  ))
(b-n            (p:or (message-received)))
(object-lumping-specs ((
  ( - (from-node echo))
  ( - (expandable-state-function echo))
  ((expandable-state-function echo t) (vmt-ids phd:if))
)
(value expandable-state?)))
(value
  (f phd:if
    (path (f select-structure (path self object-ptrs))
      vmt-ids)))
(object-ptrs      message-ob)
(object-to-use    (
  (((* hyp)(* hyp-ob)) new)
  ((goal-created goal-ob) new)
  ((message-received message-ob) same))))
:
:-----
:
:                STATE channel-ok
((f-n            (p:or (message-received)))
(object-ptrs      channel-ob)
(object-to-use    ((message-received message-ob) new)))
(primitive?      t))

:
:-----
:
:                OBJECT channel-ob

((vmt-ids      t)

```

```
(expandable-state-function  
(attribute-evaluation-sets  
(nodes-linked  
  (path x1 from-node)  
  (path x1 node))))
```

```
t)  
((nodes-linked))  
(f list
```

§5. Ksi Rating Derivation Cluster

This model cluster is used during comparative reasoning to explain, in terms of its primitive states, why some knowledge source instantiation (ksi) rating was lower than another ksi rating.

```

:*****
:          CLUSTER ksi-rating derivation cluster
:          (comparative reasoning)
:*****
:
:          STATE ksi-rating
:
:
((f-n      (p:or (ksi-rating-max)))
(b-n      (p:and (ks-goodness)
               (data-component)))
(value     (f compare-actual-values
            (path self actual-value)
            (path (path self parallel-state) actual-value)))
(actual-value (f get-ksi-rating
                (path self object-ptrs)))
(parallel-state
 (f find-highest-rated-same-type-ksi
  (path (f select-structure (path self object-ptrs)) ks-type)))
(expandable-state? t)
(object-ptrs      ksi-ob)
(object-to-use    ((ksi-rating-max ksi-ob) same))
(primitive?      c))
:
:-----
:          STATE ks-goodness
:
((f-n      (p:or (ksi-rating)))
(value     (f compare-actual-values
            (path self actual-value)
            (path (path self parallel-state) actual-value)))
(actual-value (f get-ks-goodness
                (path self object-ptrs)))
(parallel-state (f eq

```



```

    (path self type)
    (path x1 type)))
(expandable-state?      t)
(object-ptrs            ksi-ob)
(object-to-use          (((ksi-rating ksi-ob) same)))
(primitive?            t))

```

```

:
:-----
:

```

STATE data-component

```

((f-n      (p:or (ksi-rating)))
 (b-n      (p:and (hyp-rating)))
 (value    (f compare-actual-values
             (path self actual-value)
             (path (path self parallel-state) actual-value)))
 (actual-value (f get-data-component
                 (path self object-ptrs)))
 (parallel-state (f eq
                   (path self type)
                   (path x1 type)))
 (expandable-state?      t)
 (object-ptrs            ksi-ob)
 (object-to-use          (((ksi-rating ksi-ob) same))))

```

```

:

```

```

:-----
:

```

STATE hyp-rating

```

((f-n      (p:or (data-component)
                 (hyp-rating)))
 (b-n      (p:or (f make-hyp-rating-neighbor-name
                 (path (f select-structure (path self object-ptrs)) level))))
 (value    (f compare-actual-values
             (path self actual-value)
             (path (path self parallel-state) actual-value)))
 (actual-value (f get-hyp-rating
                 (path self object-ptrs)))
 (parallel-state (f and
                   ;; hyp levels must match
                   (f eq
                     (path (f select-structure (path self object-ptrs)) level)

```

```

      (path (f select-structure (path x1 object-ptrs)) level))
    ;; hyp lengths must match
    (f tll-lengths-equal
      (path (f select-structure
        (path self object-ptrs)) time-location-list)
      (path (f select-structure
        (path x1 object-ptrs)) time-location-list))
    ;; hyp event classes must match
    (f eq
      (path (f select-structure
        (path self object-ptrs)) event-class)
      (path (f select-structure
        (path x1 object-ptrs)) event-class))))
(expandable-state?      t)
(object-ptrs             rating-hyp-ob)
(object-to-use          (
  ((data-component ksi-ob) new)
  ((hyp-rating      (* hyp-ob)) new))))
:
:-----
:
:                               OBJECT rating-hyp-ob
:
((specific-name         (f make-hyp-ob-name
  (path self level)))
(vmt-ids (
  ((hyp-rating ksi-ob)
  (f get-supporting-hyps-from-ksi
    x1))
  ((hyp-rating (* hyp-ob))
  (f get-supporting-hyps-from-rating-hyp-ob
    (path x1 vmt-ids)))
  ))
(expandable-state-function t)
(s-attributes           (vmt-ids))
(attribute-evaluation-sets ((vmt-ids)
  (event-class level node rating
    time-location-list )))
(context-dependent-attributes (vmt-ids))
(time-location-list      (f select:hyp:time-location-list
  (path self vmt-ids)))
(event-class             (f select:hyp:event-class

```

```

    (path self vmt-ids))
(rating      (f select:hyp:belief
    (path self vmt-ids))
(level      (f select:hyp:level
    (path self vmt-ids))
(node      (path xi node)))

:
:-----
:
:                STATE sensed-value-rating

((f-n      (p:or (hyp-rating)))
(b-n      (p:and (sensor-weight) (data-signal)))
(value    (f compare-actual-values
    (path self actual-value)
    (path (path self parallel-state) actual-value)))
(actual-value    (f get-sensed-value-rating
    (path self object-ptrs)))
(parallel-state    (f eq
    (path self type)
    (path xi type)))
(expandable-state?    t)
(object-ptrs    sensed-value-ob)
(object-to-use    (((hyp-rating (* hyp-ob)) new))))

:
:-----
:
:                STATE sensor-weight

((f-n      (p:or (sensed-value-rating)))
(value    (f compare-actual-values
    (path self actual-value)
    (path (path self parallel-state) actual-value)))
(actual-value    (f get-sensor-weight
    (path self object-ptrs)))
(parallel-state    (f eq
    (path self type)
    (path xi type)))
(expandable-state?    t)
(object-ptrs    sensor-ob)
(object-to-use    ((sensed-value-rating sensed-value-ob) new))

```

```

(primitive?      t))
:
:-----
:
:                               STATE data-signal

((f-n           (p:or (sensed-value-rating)))
 (value         (if compare-actual-values
                  (path self actual-value)
                  (path (path self parallel-state) actual-value)))
 (actual-value  (if get-data-signal
                  (path self object-ptrs)))
 (parallel-state (if eq
                     (path self type)
                     (path xi type)))
 (expandable-state? t)
 (object-ptrs      data-ob)
 (object-to-use    (
 (sensed-value-rating sensed-value-ob) new))
 (primitive?      t))

```

Appendix C

DETECTION OF A PROBLEM

This appendix defines the detection problem and discusses possible approaches to its solution in the context of distributed problem-solving systems.

§1. The Detection Problem

The problem of detection is to **recognize an abnormal system state**. To do this, the system must have an idea of what constitutes a normal system state. The obvious way to detect an abnormal system state would be to compare the system behavior with the correct behavior, i.e., behavior expected if the system was working in an optimal manner on deriving the correct answer. Since we are dealing with a problem solving system neither the correct answer nor the optimal approach is available a priori. Therefore we must have other methods of detecting unsatisfactory states that do not involve comparing the system's progress with the answer it is attempting to derive. There are two such methods:

1. one uses general knowledge about appropriate system behavior, about the task, and about the environment;

2. the other uses internal consistency standards based on redundancy within the system.

General knowledge about appropriate system behavior comes from expectations of what a good interpretation system should do. For example, we expect the system's confidence in its results to increase with time. We also expect it to make reasonable progress through the environment, accounting for more and more of its data as time goes on. Other system behavior criteria may include time or accuracy limits imposed by the user. In this case the system would be functioning abnormally if it did not derive an answer within a specified time limit or with specified degree of accuracy.

The system can also use domain specific knowledge which can be applied to detect an abnormal system state. In our case, the domain of acoustic sensing, we can put constraints on the types and motion of the vehicles which can function as expectations whose violation constitutes an abnormal system state. For example, vehicles can move only within some range of velocities or can change direction within some range of angles.

Finally, the system can use some general knowledge about the specific environment it is dealing with. This knowledge can be obtained by some preliminary, rough measurements of the data. For example, before the detailed, intelligent interpretation, the system can build a density map of the data. This would then serve as an expectation of the load distribution. For example, if an agent in an area known to have high density of raw data is idle, this is an abnormal system state.

Internal consistency standards can be utilized in a system which contains redundant information. This redundancy can be in terms of multiple views of the same data or in terms of multiple ways to process the data to derive the answer. Using the internal consistency measures consists of making sure that these multiple sources of information agree. Our system is ideally suited for this method since it contains both multiple view of the same data (by agents in different locations) and multiple methods of deriving the final answer (different knowledge sources taking different paths to the solution). Examples of measures expectations based on internal consistency are:

- Predictions based on partial results should be fulfilled. For example, in the distributed interpretation system, predictions are represented as goals. A goal is an expectation that a vehicle of a particular type will be in a particular region at a particular time. Such expectations are produced by agents in the course of processing and are either fulfilled locally by the same agent or sent out to another agent which can fulfill them.
- The system parameters should have values consistent with each other and with the current operating environment of the interpretation system, (i.e., with respect to the current system configuration, data distribution, and external expectations placed on the system). For example, the interpretation system's knowledge is contained in task processing modules (knowledge sources) which are rated based on their effectiveness, the strength of the data they will utilize, and the importance of the results they will produce. These knowledge sources are rated and if they exceed some predefined threshold they are put on a queue. The highest rated knowledge source is put on the queue each system cycle. If the data is very weak and the threshold does not reflect this, it may be set so high that very few or no knowledge

sources are invoked and as a result the agent is under-utilized.

- Communication areas among the different agents should be consistent and utilized. In order for two agents to communicate, their communication areas must match: the sending agent must know it should send messages to the receiving agent and vice versa. If the communication areas do not match, then an agent may continue sending messages which, although they get over the channel, are ignored by the receiving agent because it does not know that it should be accepting them.

A distributed problem solving system such as the DVMT, contains many such examples of internal consistency and we plan to exploit them in detecting an abnormal system state.

Appendix D

THE DIAGNOSIS MODULE TRACES FOR THE EXAMPLES

§1. Introduction

This appendix contains traces generated by the Diagnosis Module during the diagnosis of the two examples in Chapter V. This section describes the trace format. The following two sections contain the traces for Example I and II respectively.

The trace output of the Diagnosis Module consists of:

HEADER: which states the date of the run, an identification of the DVMT run being analyzed (this is the Environment: attribute in the header) and other details regarding the dates of the system.

PARAMETER SETTINGS: which state the settings of the various parameters of the Diagnosis Module. Currently the most important ones are the *p:expandable-clusters, which indicate what parts of the overall model are to be used during this diagnosis, and *p:reasoning-types, which indicate what types of reasoning is to be done during the diagnosis. Eventually, these parameters would be set by the system itself.

TRACE BODY: which contains all the output of the Diagnosis Module.

Appendix D

THE DIAGNOSIS MODULE TRACES FOR THE EXAMPLES

§1. Introduction

This appendix contains traces generated by the Diagnosis Module during the diagnosis of the two examples in Chapter V. This section describes the trace format. The following two sections contain the traces for Example I and II respectively.

The trace output of the Diagnosis Module consists of:

HEADER: which states the date of the run, an identification of the DVMT run being analyzed (this is the `Environment:` attribute in the header) and other details regarding the dates of the system.

PARAMETER SETTINGS: which state the settings of the various parameters of the Diagnosis Module. Currently the most important ones are the `*p:expandable-clusters`, which indicate what parts of the overall model are to be used during this diagnosis, and `*p:reasoning-types`, which indicate what types of reasoning is to be done during the diagnosis. Eventually, these parameters would be set by the system itself.

TRACE BODY: which contains all the output of the Diagnosis Module.

The trace is controlled by the trace keyword list. A full trace thus gives an extremely detailed version of the diagnosis, while a minimal trace may just give the initial symptom and the identified failures.

§2. Trace Format Description

The format of the trace is:

$$\langle \text{TRACE KEYWORD} \rangle \implies \langle \text{details of operation} \rangle.$$

The trace keyword indicates the type of operation being performed, for example: object or state being created, type of reasoning being initialized, model structures (i.e., objects or states) being linked, or failures being found. For example, STATE-CREATED keyword indicates that some state was instantiated; FAILURE-FOUND keyword indicates that a failure has been identified by the DM. The details of the operation specify the objects or states involved in the operation. The format of this part of the trace varies with the type of keyword. For the above two examples, the operation details would be the name of the instantiated state created and the name of the state indicating the identified failure. The possible keywords as well as the exact formats of their details-of-operation trace is described later in this section.

Uninstantiated names of objects and states are the same as in the model cluster diagrams; i.e., PT, PT-HYP-OB, MESSAGE-EXISTS, MESSAGE-OB, etc. Instantiated state and object names have four digits attached to the end of their name. Thus examples of the instantiations of the above states and objects

would be: PT0001, PT-HYP-OB0001, MESSAGE-EXISTS0034, or MESSAGE-OB0239. In general, in describing the details of the operations, references to uninstantiated structures will be abbreviated by US, references to instantiated structures by IS.

The model instantiation proceeds in cycles. At each cycle the current state is evaluated, the appropriate neighbors (i.e., back neighbors or forward neighbors etc.) are selected for instantiation, and finally all relevant objects and states are created. These cycles repeat until a failure is found or the model cannot be expanded any further. The individual cycles are separated by the following message:

INSTANTIATING <link type> <state name (US)>

NEIGHBORS OF STATE <current state (IS)>.

A special type of trace message is the abstracted object definition message. The trace keyword for these messages is the string OBDEF concatenated with whatever the name of the instantiated object is, for example, VT-HYP-OB0001. The details-of-operation message contains a subset of that object's variable attributes. The attribute names are not included in the trace, due to lack of space. Below is a list of the attributes printed for each object.

Printed object attribute names

HYP-OB	vmt-ids time-location-list event-class rating level
	node
KSI-OB	vmt-ids stimulus-hyps output-hyps necessary-
	hyps stimulus-goals ks-type rating node
GOAL-OB	vmt-ids necessary-hyps active-trl inactive-trl level
	event-classes stimulus-hyps output-hyps goal-
	type rating node
SENSOR-OB	vmt-ids time-regions event-classes weight nodes
DATA-OB	time-location event-class sensed-values
SENSED-VALUE-OB	vmt-ids time-location event-class sensor-id value
	node
MESSAGE-OB	vmt-ids message-type tll/trl event-classes rating
	node to-node from-node level
CHANNEL-OB	vmt-ids nodes-linked
COMM-KSI-OB	vmt-ids message ks-type rating node

The remainder of this section describes the trace keywords and explains the format of their corresponding details-of-operation messages. The keywords are grouped under several headings to make understanding easier.

Messages regarding creation and linking of structures

OBJECT-CREATED	⇒	<IS>
OBDEF <IS ob name >	⇒	<list of the values of the printed variable attributes for that object type; see above for the attribute values printed>
STATE-CREATED	⇒	<IS>
OBJECT/STATE-LINKED	⇒	(<IS - objects> to state <IS - state>
OBJECT/OBJECT-LINKED	⇒	<link-type> neighbors of <IS object> linked to <list of IS objects>
STATE/STATE-LINKED	⇒	<link type> neighbors of <IS state> <neighbor list expression which includes the newly instantiated state names, including logical connectives>
NEIGHBORS-LINKED	⇒	<link type> neighbors of state <IS state> are <list of IS state names>. (This is a shortened version of above. It simply lists all names of the instantiated states, leaving out their logical relationships.)
EXPANDING-LINK	⇒	<link-type> of state <IS state name> into <neighbor list expression> (This is used when neighbors need to be expanded during UVD, not necessarily because the causal paths lie in that direction, but because the neighboring values need to be known.)

State and path value assignments

STATE-EVALUATED	⇒	<IS state name> <state value>
UNKNOWN-OBJECT	⇒	<IS state name> determined to be <value>
PATH-VALUE-ASSIGNED	⇒	to <link type> of state <IS state name> value is <value> path value is <list of path values>

Messages regarding types of reasoning and control in the diagnosis

DIAGNOSIS-BEGINS	⇒	with SYMPTOM <IS state name> [in node <#>]
DIAGNOSIS-CONTINUES	⇒	with state <IS state name>
DIAGNOSIS-ENDS	⇒	of SYMPTOM <IS state> in node #
INITIATING-UVD	⇒	<IS state name whose value needs to be determined>
UNKNOWN-VALUE-CYCLE	⇒	state is <IS state name>
ENDING-UVD	⇒	unknown state <IS state name > ex- panded into <list of new IS state names> new state values are <list of values>
SEARCH-CHANGED	⇒	from <search type to search type> at state <IS state name>
ILLEGAL-NEIGHBOR	⇒	<link type> of state <IS state name> is not in *p:expandable- clusters.

Messages regarding Forward Causal Tracing and fault simulation

INITIATING-FCT	⇒	
FCT-CONTINUES	⇒	with state <IS state name>
ENDING-FCT	⇒	STATE <IS state name> is a node transition state
PENDING-STATE	⇒	<IS state name> explained by fault <IS fault object name>
OBJECT-EXPLAINED	⇒	<IS ob name> of state <(IS state name)> due to overlap with FCT object <(IS ob name)> of FCT state <IS state name>
FCT-STATE-CREATED	⇒	<IS state name>. (This indicates the creation of a states which has underconstrained objects attached to it. It thus represents a predicate regarding the whole class of objects represented by the attached underconstrained object.)
FCT-OBJECT-CREATED	⇒	< IS object name>. (This indicates the creation of an underconstrained object.)
OVERLAPPING	⇒	regular object found. FCT <IS object name> overlaps with regular <IS object name>. (This indicates that a fully constrained object (i.e., a regular object) has been found which overlaps with the underconstrained object representing the class affected by the fault being simulated.)
INITIATING-PATH-VALUE	⇒	determination of an FCT state <IS state>. (This is necessary to make sure that a state cannot be achieved via another path. Before an FCT can assign a path value, it must check to make this that state's path value is indeed false.)
ENDING-PATH-VALUE	⇒	determination of an FCT state <IS state> value is <value>.

Comparative Reasoning messages

COMPARATIVE REASONING \Rightarrow continues with state <IS state>.
 PARALLEL-STATE-FOUND \Rightarrow problem-state <IS state> has
 been
 linked to model state <IS state>.

Failure and end-of-path related messages

FAILURE-FOUND \Rightarrow FALSE SYMPTOM STATE <IS state
 name> belongs to node <node #>.
 FAILURE-FOUND \Rightarrow NON-EXPANDABLE-STATE <IS state
 name> objects are <list of IS object
 names>
 FAILURE-FOUND \Rightarrow FALSE PRIMITIVE STATE value of
 state <IS state> is F.
 DEAD-END \Rightarrow stm-state <IS state name> has no neigh-
 bors of type <link types> SEARCH
 TERMINATES

System book keeping messages

DUPLICATE-STATE \Rightarrow <IS state name> used in conjunction with
 object <IS object name>. (Saves creating
 structure if one already exists.)
 MERGED-PATHS \Rightarrow <state type I> <IS state name> has
 evaled p-value <path value>

§3. Traces for Example I

This section contains the detailed Diagnosis Module traces for Example I discussed in Chapter V.

§3.1 Node #1: PT to MESSAGE-ACCEPTED

This portion of the trace shows in detail part of the processing required to produce the instantiated model in Figure 42. Specifically, it shows how BCT instantiates state PT79, VT80, and the states MESSAGE-ACCEPTED81-83 and VL84.

PHD SYSTEM TRACE

Date: 20-JUN-1985
 Environment: simple4.fin
 LTM modifying files: (1)
 Diagnosis for node: 1
 Run Started: 14:11:47.90
 Last Stopped: 14:11:47.79
 Kernel Date: 28-FEB-1985 00:02:01.86

THE VALUES OF THE PHD SYSTEM PARAMETERS

*p:tll-match exact
 p:expandable-clusters *answer-derivation.*communication
 *ksi-scheduling.*comm-ksi-scheduling
 p:reasoning-types BCT

BEGINNING OF PHD SYSTEM TRACE

OBJECT-CREATED -----> pt-hyp-ob0038
 STATE-CREATED -----> pt0079
 DIAGNOSIS-BEGINS -----> with SYMPTOM pt0079
 OBJECT-STATE-LINKED ---> (pt-hyp-ob0038) to state pt0079
 OBDEF pt-hyp-ob0038 ---> (f ((1 (6 2)) (2 (8 4)) (3 (10 6)) (4 (12
 8)) (5 (14 10)) (6 (16 12)) (7 (18 14)) (8 (20 16))) 1 nil pt 1)
 STATE-EVALUATED -----> pt0079 f

INSTANTIATING b-n pt NEIGHBORS OF STATE pt0079

OBJECT-OBJECT-LINKED --> b-n neighbors of pt-hyp-ob0038 linked to ni
1

INSTANTIATING b-n vt NEIGHBORS OF STATE pt0079

OBJECT-CREATED -----> vt-hyp-ob0039
 OBJECT-OBJECT-LINKED --> b-n neighbors of pt-hyp-ob0038 linked to (v
t-hyp-ob0039)
 OBDEF vt-hyp-ob0039 ---> (f ((1 (6 2)) (2 (8 4)) (3 (10 6)) (4 (12
8)) (5 (14 10)) (6 (16 12)) (7 (18 14)) (8 (20 16))) 1 * vt 1)
 STATE-CREATED -----> vt0080
 OBJECT-STATE-LINKED ---> (vt-hyp-ob0039) to state vt0080
 STATE-STATE-LINKED ----> b-n neighbors of pt0079 (p:or (p:and (pt))
(p:or (vt))) expanded into (p:or (p:and (pt)) (p:or (vt vt0080)))
 NEIGHBORS-LINKED -----> b-n neighbors of state pt0079 are (vt0080)

DIAGNOSIS-CONTINUES ---> with state vt0080
 STATE-EVALUATED -----> vt0080 f

INSTANTIATING b-n message-accepted NEIGHBORS OF STATE vt0080

OBJECT-CREATED -----> message-ob0040
 OBJECT-CREATED -----> message-ob0041
 OBJECT-CREATED -----> message-ob0042
 OBJECT-OBJECT-LINKED --> b-n neighbors of vt-hyp-ob0039 linked to (m
essage-ob0040 message-ob0041 message-ob0042)
 OBDEF message-ob0040 --> (f hyp-ob ((1 (6 2)) (2 (8 4)) (3 (10 6))
(4 (12 8)) (5 (14 10)) (6 (16 12)) (7 (18 14)) (8 (20 16))) 1 * 1 -
2 vt)
 OBDEF message-ob0041 --> (f hyp-ob ((1 (6 2)) (2 (8 4)) (3 (10 6))

(4 (12 8)) (5 (14 10)) (6 (16 12)) (7 (18 14)) (8 (20 16))) 1 * 1 -
3 vt)

OBDEF message-ob0042 --> (f hyp-ob ((1 (6 2)) (2 (8 4)) (3 (10 6))
(4 (12 8)) (5 (14 10)) (6 (16 12)) (7 (18 14)) (8 (20 16))) 1 * 1 -
4 vt)

STATE-CREATED -----> message-accepted0081

OBJECT-STATE-LINKED ----> (message-ob0040) to state message-accepted0
081

STATE-CREATED -----> message-accepted0082

OBJECT-STATE-LINKED ----> (message-ob0041) to state message-accepted0
082

STATE-CREATED -----> message-accepted0083

OBJECT-STATE-LINKED ----> (message-ob0042) to state message-accepted0
083

INSTANTIATING b-n vt NEIGHBORS OF STATE vt0080

OBJECT-OBJECT-LINKED --> b-n neighbors of vt-hyp-ob0039 linked to ni
1

INSTANTIATING b-n v1 NEIGHBORS OF STATE vt0080

OBJECT-CREATED -----> v1-hyp-ob0043

OBJECT-CREATED -----> v1-hyp-ob0044

OBJECT-CREATED -----> v1-hyp-ob0045

OBJECT-CREATED -----> v1-hyp-ob0046

OBJECT-CREATED -----> v1-hyp-ob0047

OBJECT-CREATED -----> v1-hyp-ob0048

OBJECT-CREATED -----> v1-hyp-ob0049

OBJECT-CREATED -----> v1-hyp-ob0050

OBJECT-OBJECT-LINKED --> b-n neighbors of vt-hyp-ob0039 linked to (v
1-hyp-ob0043 v1-hyp-ob0044 v1-hyp-ob0045 v1-hyp-ob0046 v1-hyp-ob004
7 v1-hyp-ob0048 v1-hyp-ob0049 v1-hyp-ob0050)

OBDEF v1-hyp-ob0043 ----> (f (1 (6 2)) 1 * v1 1)

OBDEF v1-hyp-ob0044 ----> (f (2 (8 4)) 1 * v1 1)

OBDEF v1-hyp-ob0045 ----> (f (3 (10 6)) 1 * v1 1)

OBDEF v1-hyp-ob0046 ----> (f (4 (12 8)) 1 * v1 1)

OBDEF v1-hyp-ob0047 ----> (f (5 (14 10)) 1 * v1 1)

OBDEF v1-hyp-ob0048 ----> (f (6 (16 12)) 1 * v1 1)

OBDEF v1-hyp-ob0049 ----> (f (7 (18 14)) 1 * v1 1)

OBDEF v1-hyp-ob0050 ----> (f (8 (20 16)) 1 * v1 1)

STATE-CREATED -----> v10084
 OBJECT-STATE-LINKED ---> (v1-hyp-ob0043 v1-hyp-ob0044 v1-hyp-ob0045
 v1-hyp-ob0046 v1-hyp-ob0047 v1-hyp-ob0048 v1-hyp-ob0049 v1-hyp-ob00
 50) to state v10084
 STATE-STATE-LINKED ----> b-n neighbors of vt0080 (p:or (p:or (messag
 e-accepted)) (or (p:and (vt)) (p:and (v1)))) expanded into (p:or (p
 :or (message-accepted message-accepted0081 message-accepted0082 mes
 sage-accepted0083)) (p:and (v1 v10084)))
 NEIGHBORS-LINKED -----> b-n neighbors of state vt0080 are (message
 -accepted0081 message-accepted0082 message-accepted0083 v10084)

§3.2 Node #1: MESSAGE-ACCEPTED to MESSAGE-SENT

This portion of the trace shows how Unknown Value Derivation determines
 the value for the state MESSAGE-RECEIVED88. The determination of the value
 for the other MESSAGE-RECEIVED states in Figure 42 is analogous is is not
 shown. The trace concludes by showing the path value for the initial symptom
 state, PT79. This concludes diagnosis in Node #1. Some trace messages are
 omitted here (and in the remainder of the appendix) to save space. Specifically,
 STATE-CREATED, STATE-STATE-LINKED, OBJECT-STATE-LINKED, and
 OBJECT-OBJECT-LINKED.

DIAGNOSIS-CONTINUES ---> with state message-accepted0081
 STATE-EVALUATED -----> message-accepted0081 f

INSTANTIATING b-n message-received NEIGHBORS OF STATE
 message-accepted0081

STATE-CREATED -----> message-received0085
 NEIGHBORS-LINKED -----> b-n neighbors of state message-accepted008
 1 are (message-received0085)
 INITIATING-UVD -----> stm state is message-received0085

INSTANTIATING b-n message-sent NEIGHBORS OF STATE
 message-received0085

OBDEF message-ob0051 --> (f hyp-ob ((1 (6 2)) (2 (8 4)) (3 (10 6))
 (4 (12 8)) (5 (14 10)) (6 (16 12)) (7 (18 14)) (8 (20 16))) 1 * 2 1
 - vt)

STATE-CREATED -----> message-sent0086

INSTANTIATING b-n channel-ok NEIGHBORS OF STATE
 message-received0085

OBDEF channel-ob0052 --> (t (2 1))

STATE-CREATED -----> channel-ok0087

NEIGHBORS-LINKED -----> b-n neighbors of state message-received008
 5 are (message-sent0086 channel-ok0087)

STATE-EVALUATED -----> message-sent0086 f

STATE-EVALUATED -----> channel-ok0087 t

UNKNOWN-OBJECT -----> message-ob0040 determined to be f

STATE-CREATED -----> message-received0088

ENDING-UVD -----> unknown state message-received0085 expanded
 into (message-received0088) new state values are (f)

 DIAGNOSIS-CONTINUES ---> with state message-received0088

STATE-EVALUATED -----> message-received0088 f

INSTANTIATING b-n message-sent NEIGHBORS OF STATE
 message-received0088

NEIGHBORS-LINKED -----> b-n neighbors of state message-received008
 8 are (message-sent0086 channel-ok0087)

 DIAGNOSIS-CONTINUES ---> with state message-sent0086

STATE-EVALUATED -----> message-sent0086 f

FAILURE-FOUND -----> FALSE SYMPTOM STATE message-sent0086 belong
 s to node 2

 DIAGNOSIS-CONTINUES ---> with state channel-ok0087

STATE-EVALUATED -----> channel-ok0087 t

PATH-VALUE-ASSIGNED ---> to b-n of state message-received0088 value
is f path-value is (f)

PATH-VALUE-ASSIGNED ---> to b-n of state message-accepted0081 value
is f path-value is (f)

DIAGNOSIS-CONTINUES ---> with state message-accepted0082
STATE-EVALUATED -----> message-accepted0082 f

-----> this is analogous to previous trace <-----
-----> of the diagnosis of MESSAGE- <-----
-----> ACCEPTED81. It eventually results <-----
-----> in the creation of the pending <-----
-----> symptom state MESSAGE-SENT90. <-----

DIAGNOSIS-CONTINUES ---> with state message-sent0090
STATE-EVALUATED -----> message-sent0090 f
FAILURE-FOUND -----> FALSE SYMPTOM STATE message-sent0090 belong
s to node 3

DIAGNOSIS-CONTINUES ---> with state message-accepted0083
STATE-EVALUATED -----> message-accepted0083 f

-----> this is analogous to previous trace <-----
-----> of the diagnosis of MESSAGE-ACCEPTED81. <-----
-----> It eventually results in the creation <-----
-----> of the pending symptom state MESSAGE- <-----
-----> SENT94. <-----

STATE-EVALUATED -----> message-sent0094 f
FAILURE-FOUND -----> FALSE SYMPTOM STATE message-sent0094 belong
s to node 4

DIAGNOSIS-CONTINUES ---> with state v10084
STATE-EVALUATED -----> v10084 f
FAILURE-FOUND -----> NON-EXPANDABLE-STATE v10084 objects are (v
1-hyp-ob0043 v1-hyp-ob0044 v1-hyp-ob0045 v1-hyp-ob0046 v1-hyp-ob004
7 v1-hyp-ob0048 v1-hyp-ob0049 v1-hyp-ob0050)
PATH-VALUE-ASSIGNED ---> to b-n of state vt0080 value is f path-val

ue is (f)

PATH-VALUE-ASSIGNED ---> to b-n of state pt0079 value is f path-val

ue is (f)

DIAGNOSIS-TERMINATES-SUCCESSFULLY -----> state
pt0079 explained in terms of identified failures

§3.3 Node #2:PT1-VT5

This part of the trace shows the processing for instantiating the model in Figure 43. Only the essential messages are displayed here.

```

STATE-CREATED -----> pt0001
DIAGNOSIS-BEGINS -----> with SYMPTOM pt0001
STATE-EVALUATED -----> pt0001 f

INSTANTIATING b-n pt NEIGHBORS OF STATE pt0001

STATE-CREATED -----> pt0002
STATE-CREATED -----> pt0003

INSTANTIATING b-n vt NEIGHBORS OF STATE pt0001

STATE-CREATED -----> vt0004
STATE-STATE-LINKED ----> b-n neighbors of pt0001 (p:or (p:and (pt))
  (p:or (vt))) expanded into (p:or (p:and (pt pt0002 pt0003)) (p:or
  (vt vt0004)))
NEIGHBORS-LINKED -----> b-n neighbors of state pt0001 are (pt0002
pt0003 vt0004)
-----

DIAGNOSIS-CONTINUES ---> with state pt0002
STATE-EVALUATED -----> pt0002 f

INSTANTIATING b-n pt NEIGHBORS OF STATE pt0002

OBJECT-OBJECT-LINKED --> b-n neighbors of pt-hyp-ob0002 linked to n
11

INSTANTIATING b-n vt NEIGHBORS OF STATE pt0002

STATE-CREATED -----> vt0005
STATE-STATE-LINKED ----> b-n neighbors of pt0002 (p:or (p:and (pt))
  (p:or (vt))) expanded into (p:or (p:and (pt)) (p:or (vt vt0005)))
NEIGHBORS-LINKED -----> b-n neighbors of state pt0002 are (vt0005)
-----

```

DIAGNOSIS-CONTINUES ---> with state vt0005
 STATE-EVALUATED -----> vt0005 f

INSTANTIATING b-n message-accepted NEIGHBORS OF STATE vt0005

STATE-CREATED -----> message-accepted0006
 STATE-CREATED -----> message-accepted0007
 STATE-CREATED -----> message-accepted0008

INSTANTIATING b-n vt NEIGHBORS OF STATE vt0005

OBJECT-OBJECT-LINKED --> b-n neighbors of vt-hyp-ob0005 linked to n
 il

INSTANTIATING b-n v1 NEIGHBORS OF STATE vt0005

STATE-CREATED -----> v10009
 STATE-STATE-LINKED ----> b-n neighbors of vt0005 (p:or (p:or (message-accepted)) (or (p:and (vt)) (p:and (v1)))) expanded into (p:or (p:or (message-accepted message-accepted0006 message-accepted0007 message-accepted0008)) (p:and (v1 v10009)))
 NEIGHBORS-LINKED -----> b-n neighbors of state vt0005 are (message-accepted0006 message-accepted0007 message-accepted0008 v10009)

DIAGNOSIS-CONTINUES ---> with state message-accepted0006
 STATE-EVALUATED -----> message-accepted0006 f

-----> Diagnosis for the four states MESSAGE- <----
 -----> ACCEPTED0006 through MESSAGE-ACCEPTED <----
 -----> 0008 is analogous to the diagnosis of <----
 -----> the MESSAGE-ACCEPTED states in Node #1 <----
 -----> It eventually results in three pending <----
 -----> MESSAGE-SENT symptoms that are diag- <----
 -----> nosed later. <----

 DIAGNOSIS-CONTINUES ---> with state v10009
 STATE-EVALUATED -----> v10009 f
 FAILURE-FOUND -----> NON-EXPANDABLE-STATE v10009 objects are (v

l-hyp-ob0009 vl-hyp-ob0010 vl-hyp-ob0011 vl-hyp-ob0012)

PATH-VALUE-ASSIGNED ---> to b-n of state vt0005 value is f path-value is (f)

PATH-VALUE-ASSIGNED ---> to b-n of state pt0002 value is f path-value is (f)

§3.4 Node #2: PT3-PT1 True-False Pair

This trace shows the diagnosis for the true-false pair PT3-PT1 in Figure 47. The unknown value derivation necessary for the KSI-RATED states is analogous to the one for MESSAGE-RECEIVED states and is omitted. Again, only the essential messages are shown.

DIAGNOSIS-CONTINUES ----> with state pt0003
STATE-EVALUATED -----> pt0003 t

INSTANTIATING f-l-n stimulus-hyp NEIGHBORS OF STATE pt0003

STATE-CREATED -----> stimulus-hyp0022

INSTANTIATING f-l-n necessary-hyp NEIGHBORS OF STATE pt0003

STATE-CREATED -----> necessary-hyp0023

STATE-STATE-LINKED ----> f-l-n neighbors of pt0003 (p:or (stimulus-

hyp) (necessary-hyp)) expanded into (p:or (stimulus-hyp stimulus-hyp0022) (necessary-hyp necessary-hyp0023))

NEIGHBORS-LINKED -----> f-l-n neighbors of state pt0003 are (stimulus-hyp0022 necessary-hyp0023)

DIAGNOSIS-CONTINUES ----> with state stimulus-hyp0022

STATE-EVALUATED -----> stimulus-hyp0022 t

INSTANTIATING f-n goal-created NEIGHBORS OF STATE stimulus-hyp0022

OBDEF goal-ob0019 -----> (g:02:0100 nil ((4 (10 6 14 10))) ((5 (14 10 14 10)) (6 (16 12 16 12)) (7 (18 14 18 14)) (8 (20 16 20 16)))) p
t 1 pt-hyp-ob0003 pt-hyp-ob0001 track-b-track nil 2)

STATE-CREATED -----> goal-created0024

STATE-STATE-LINKED ----> f-n neighbors of stimulus-hyp0022 (p:or (goal-created)) expanded into (p:or (goal-created goal-created0024))

NEIGHBORS-LINKED -----> f-n neighbors of state stimulus-hyp0022 are (goal-created0024)

 DIAGNOSIS-CONTINUES ---> with state goal-created0024
 STATE-EVALUATED -----> goal-created0024 t

INSTANTIATING f-n ksi-rated NEIGHBORS OF STATE goal-created0024

OBDEF pt-hyp-ob0002 ---> (f ((1 (6 2)) (2 (8 4)) (3 (10 6)) (4 (12 8))) 1 * pt 2)

OBDEF pt-hyp-ob0002 ---> (f ((1 (6 2)) (2 (8 4)) (3 (10 6)) (4 (12 8))) 1 * pt 2)

OBDEF ksi-ob0020 -----> (f pt-hyp-ob0003 pt-hyp-ob0001 pt-hyp-ob0002 goal-ob0019 cb:pt * 2)

OBDEF ksi-ob0021 -----> (f pt-hyp-ob0003 pt-hyp-ob0001 pt-hyp-ob0002 goal-ob0019 mb:pt * 2)

STATE-CREATED -----> ksi-rated0025

STATE-CREATED -----> ksi-rated0026

STATE-STATE-LINKED ----> f-n neighbors of goal-created0024 (p:or (ksi-rated)) expanded into (p:or (ksi-rated ksi-rated0025 ksi-rated0026))

NEIGHBORS-LINKED -----> f-n neighbors of state goal-created0024 are (ksi-rated0025 ksi-rated0026)

INITIATING-UVD -----> stm state is ksi-rated0025

STATE-CREATED -----> ksi-scheduled0027

STATE-CREATED -----> ks-exists0028

STATE-CREATED -----> necessary-hyp0029

STATE-EVALUATED -----> ks-exists0028 t

STATE-EVALUATED -----> goal-created0024 t

STATE-EVALUATED -----> necessary-hyp0029 f

UNKNOWN-OBJECT -----> ksi-ob0020 determined to be f

STATE-CREATED -----> ksi-rated0030

STATE-STATE-LINKED ----> b-n neighbors of ksi-rated0030 (p:and (ks-

exists) (goal-created) (necessary-hyp)) expanded into (p:and (ks-exists ks-exists0028) (goal-created goal-created0024) (necessary-hyp necessary-hyp0029))

STATE-STATE-LINKED ----> f-n neighbors of ksi-rated0030 (p:or (ksi-

scheduled)) expanded into (p:or (ksi-scheduled ksi-scheduled0027))

STATE-EVALUATED -----> ksi-rated0030 f

ENDING-UVD -----> unknown state ksi-rated0025 expanded into

(ksi-rated0030) new state values are (f)
 INITIATING-UVD -----> stm state is ksi-rated0026

-----> Analogous to the UVD for KSI-RATED0025. <-----
 -----> Results in the creation of a new KSI- <-----
 -----> RATED0033 state, which is determined to <-----
 -----> be false. <-----

STATE-EVALUATED -----> ksi-rated0033 f
 ENDING-UVD -----> unknown state ksi-rated0026 expanded into
 (ksi-rated0033) new state values are (f)

DIAGNOSIS-CONTINUES ---> with state ksi-rated0033
 STATE-EVALUATED -----> ksi-rated0033 f

DIAGNOSIS-CONTINUES ---> with state ks-exists0032
 STATE-EVALUATED -----> ks-exists0032 t

DIAGNOSIS-CONTINUES ---> with state goal-created0024
 STATE-EVALUATED -----> goal-created0024 t

DIAGNOSIS-CONTINUES ---> with state necessary-hyp0029
 STATE-EVALUATED -----> necessary-hyp0029 f

INSTANTIATING b-u-n pt NEIGHBORS OF STATE necessary-hyp0029

DUPLICATE-STATE -----> pt0002 used in conjunction with object pt-

hyp-ob0002

STATE-STATE-LINKED ----> b-u-n neighbors of necessary-hyp0029 (p:or
 (f make-hyp-state-name (path (f select-structure (path self
 object-ptrs)) level))) expanded into (p:or (pt pt0002))
 NEIGHBORS-LINKED -----> b-u-n neighbors of state necessary-hyp0029
 are (pt0002)

DIAGNOSIS-CONTINUES ---> with state pt0002
 STATE-EVALUATED -----> pt0002 f
 MERGED-PATHS -----> regular pt0002 b-n has eveled p-value (f)
 PATH-VALUE-ASSIGNED ---> to b-u-n of state necessary-hyp0029 value
 is f path-value is (f)
 PATH-VALUE-ASSIGNED ---> to b-n of state ksi-rated0033 value is f p
 ath-value is (f)

 DIAGNOSIS-CONTINUES ---> with state ksi-rated0030
 STATE-EVALUATED -----> ksi-rated0030 f

-----> Analogous to diagnosis for KSI-RATED0033. <-----
 -----> Results in merging with the NECESSARY- <-----
 -----> HYP0029 state. <-----

 DIAGNOSIS-CONTINUES ---> with state necessary-hyp0029
 STATE-EVALUATED -----> necessary-hyp0029 f
 MERGED-PATHS -----> regular necessary-hyp0029 b-u-n has eveled
 p-value (f)
 PATH-VALUE-ASSIGNED ---> to b-n of state ksi-rated0030 value is f p
 ath-value is (f)
 PATH-VALUE-ASSIGNED ---> to f-n of state goal-created0024 value is
 t path-value is (f)
 PATH-VALUE-ASSIGNED ---> to f-n of state stimulus-hyp0022 value is
 t path-value is (f)

 DIAGNOSIS-CONTINUES ---> with state necessary-hyp0023
 STATE-EVALUATED -----> necessary-hyp0023 t
 SEARCH-CHANGED -----> from bif2 to F2 at state necessary-hyp0023

-----> this part of diagnosis omitted from <-----
 -----> example but leads to a false path <-----
 -----> value due to missing pt 1-4 track <-----

PATH-VALUE-ASSIGNED ---> to f-n of state necessary-hyp0023 value is
 t path-value is (f)
 PATH-VALUE-ASSIGNED ---> to f-l-n of state pt0003 value is t path-v

alue is (f)

DIAGNOSIS-CONTINUES ---> with state vt0004
STATE-EVALUATED -----> vt0004 f

-----> this part of diagnosis is analogous <----
-----> to that shown for diagnosis of Node <----
-----> #1 resulting in a false path value <----
-----> for state VT0004 <----

(PATH-VALUE-ASSIGNED ---> to b-n of state vt0004 value is f
path-value is (f))

(PATH-VALUE-ASSIGNED ---> to b-n of state pt0001 value is f
path-value is (f))

DIAGNOSIS-TERMINATES-SUCCESSFULLY

-----> state pt0001 explained in terms of
identified failures)

§3.5 Node #2: Pending Symptoms

This section shows the trace for the diagnosis shown in Figure 45. All unknown value derivations have been omitted to save space.

DIAGNOSING PENDING SYMPTOMS

DIAGNOSIS-BEGINS -----> with state message-sent0111

STATE-EVALUATED -----> message-sent0111 f

INSTANTIATING b-n message-exists NEIGHBORS OF STATE
message-sent0111

STATE-CREATED -----> message-exists0217

STATE-STATE-LINKED ----> b-n neighbors of message-sent0111 (p:or (message-exists)) expanded into (p:or (message-exists message-exists0217))

NEIGHBORS-LINKED -----> b-n neighbors of state message-sent0111 are (message-exists0217)

DIAGNOSIS-CONTINUES ---> with state message-exists0217

STATE-EVALUATED -----> message-exists0217 t

INSTANTIATING f-l-n comm-ksi-rated NEIGHBORS OF STATE
message-exists0217

OBDEF comm-ksi-ob0127 -> (f message-ob0071 hyp-send:vt * 2)

OBDEF comm-ksi-ob0128 -> (f message-ob0071 hyp-reply:vt * 2)

STATE-CREATED -----> comm-ksi-rated0218

STATE-STATE-LINKED ----> f-l-n neighbors of message-exists0217 (p:or (comm-ksi-rated)) expanded into (p:or (comm-ksi-rated comm-ksi-rated0218))

NEIGHBORS-LINKED -----> f-l-n neighbors of state message-exists0217 are (comm-ksi-rated0218)

INITIATING-UVD -----> value derivation of state comm-ksi-rated0218

-----> Results in creation of COMM-KSI- <-----
 -----> RATED0221, which is false. <-----

STATE-EVALUATED -----> comm-ksi-rated0221 f
 ENDING-UVD -----> derivation of state comm-ksi-rated0218 expanded into (comm-ksi-rated0221) new state values are (f)

DIAGNOSIS-CONTINUES ---> with state comm-ksi-rated0221
 STATE-EVALUATED -----> comm-ksi-rated0221 f

INSTANTIATING b-n comm-ks-exists NEIGHBORS OF STATE
 comm-ksi-rated0221

STATE-STATE-LINKED ----> b-n neighbors of comm-ksi-rated0221 (p:and (comm-ks-exists)) expanded into (p:and (comm-ks-exists comm-ks-exists0220))
 NEIGHBORS-LINKED -----> b-n neighbors of state comm-ksi-rated0221 are (comm-ks-exists0220)

DIAGNOSIS-CONTINUES ---> with state comm-ks-exists0220
 STATE-EVALUATED -----> comm-ks-exists0220 f
 DEAD-END -----> stm-state comm-ks-exists0220 has no neighbors of type (b-n b-u-n) SEARCH TERMINATES
 FAILURE-FOUND -----> FALSE PRIMITIVE STATE Value of state comm-

ks-exists0220 is f
 PATH-VALUE-ASSIGNED ----> to nil of state comm-ks-exists0220 value is f path-value is (f)

INITIATING-FCT ----->
 FCT-STATE-CREATED -----> comm-ks-exists0222
 FCT-OBJECT-CREATED ----> comm-ksi-ob0129
 OBDEF comm-ksi-ob0129 -> (f ~ hyp-send:vt ~ 2)
 FCT-OBJECT-CREATED ----> comm-ksi-ob0130
 OBDEF comm-ksi-ob0130 -> (f ~ hyp-reply:vt ~ 2)

FCT-CONTINUES -----> with state comm-ks-exists0222
 STATE-EVALUATED -----> comm-ks-exists0222 f
 PATH-VALUE-ASSIGNED ---> to nil of state comm-ks-exists0222 value i
 s f path-value is (f)

INSTANTIATING f-n comm-ksi-rated NEIGHBORS OF STATE
 comm-ks-exists0222

FCT-STATE-CREATED -----> comm-ksi-rated0223
 STATE-STATE-LINKED ----> f-n neighbors of comm-ks-exists0222 (p:or
 (comm-ksi-rated)) expanded into (p:or (comm-ksi-rated comm-ksi-rate
 d0223))
 NEIGHBORS-LINKED -----> f-n neighbors of state comm-ks-exists0222
 are (comm-ksi-rated0223)

 FCT-CONTINUES -----> with state comm-ksi-rated0223
 STATE-EVALUATED -----> comm-ksi-rated0223 f
 PATH-VALUE-ASSIGNED ---> to b-n of state comm-ksi-rated0223 value i
 s f path-value is (f)
 PATH-VALUE-ASSIGNED ---> F to b-n of state comm-ksi-rated0221 due t
 o overlap with FCT state comm-ksi-rated0223

INSTANTIATING f-n comm-ksi-scheduled NEIGHBORS OF STATE
 comm-ksi-rated0223

FCT-STATE-CREATED -----> comm-ksi-scheduled0224
 STATE-STATE-LINKED ----> f-n neighbors of comm-ksi-rated0223 (p:or
 (comm-ksi-scheduled)) expanded into (p:or (comm-ksi-scheduled comm-
 ksi-scheduled0224))
 NEIGHBORS-LINKED -----> f-n neighbors of state comm-ksi-rated0223
 are (comm-ksi-scheduled0224)

 FCT-CONTINUES -----> with state comm-ksi-scheduled0224
 STATE-EVALUATED -----> comm-ksi-scheduled0224 f
 PATH-VALUE-ASSIGNED ---> to b-n of state comm-ksi-scheduled0224 val
 ue is f path-value is (f)
 PATH-VALUE-ASSIGNED ---> F to b-n of state comm-ksi-scheduled0219 d
 ue to overlap with FCT state comm-ksi-scheduled0224

INSTANTIATING f-n comm-ksi-executed NEIGHBORS OF STATE
 comm-ksi-scheduled0224

FCT-STATE-CREATED -----> comm-ksi-executed0225
 STATE-STATE-LINKED ----> f-n neighbors of comm-ksi-scheduled0224 (p
 :or (comm-ksi-executed)) expanded into (p:or (comm-ksi-executed com
 m-ksi-executed0225))
 NEIGHBORS-LINKED -----> f-n neighbors of state comm-ksi-scheduled0
 224 are (comm-ksi-executed0225)

 FCT-CONTINUES -----> with state comm-ksi-executed0225
 STATE-EVALUATED -----> comm-ksi-executed0225 f
 PATH-VALUE-ASSIGNED ---> to nil of state comm-ksi-executed0225 valu
 e is f path-value is (f)
 PATH-VALUE-ASSIGNED ---> to b-n of state comm-ksi-executed0225 valu
 e is f path-value is (f)

INSTANTIATING f-u-n message-sent NEIGHBORS OF STATE
 comm-ksi-executed0225

OBDEF message-ob0131 --> ((h:02:0061 h:02:0062 h:02:0068 h:02:0069
 h:02:0075 h:02:0077 h:02:0082 h:02:0083 h:02:0086 h:02:0092 h:02:00
 93) hyp-ob ~ ~ * 2 1 ~ vt)
 OBDEF message-ob0132 --> ((h:02:0061 h:02:0062 h:02:0068 h:02:0069
 h:02:0075 h:02:0077 h:02:0082 h:02:0083 h:02:0086 h:02:0092 h:02:00
 93) hyp-ob ~ ~ * 2 3 ~ vt)
 OBDEF message-ob0133 --> ((h:02:0061 h:02:0062 h:02:0068 h:02:0069
 h:02:0075 h:02:0077 h:02:0082 h:02:0083 h:02:0086 h:02:0092 h:02:00
 93) hyp-ob ~ ~ * 2 4 ~ vt)
 OVERLAPPING -----> regular object found. FCT message-ob0131 o
 verlaps with regular message-ob0051
 OVERLAPPING -----> regular object found. FCT message-ob0132 o
 verlaps with regular message-ob0086
 OVERLAPPING -----> regular object found. FCT message-ob0132 o
 verlaps with regular message-ob0071
 OVERLAPPING -----> regular object found. FCT message-ob0133 o
 verlaps with regular message-ob0125
 OVERLAPPING -----> regular object found. FCT message-ob0133 o
 verlaps with regular message-ob0115

OVERLAPPING -----> regular object found. FCT message-ob0133 o
 verlaps with regular message-ob0114
 OVERLAPPING -----> regular object found. FCT message-ob0131 o
 verlaps with regular message-ob0051
 OVERLAPPING -----> regular object found. FCT message-ob0132 o
 verlaps with regular message-ob0086
 OVERLAPPING -----> regular object found. FCT message-ob0132 o
 verlaps with regular message-ob0071
 OVERLAPPING -----> regular object found. FCT message-ob0133 o
 verlaps with regular message-ob0125
 OVERLAPPING -----> regular object found. FCT message-ob0133 o
 verlaps with regular message-ob0115
 OVERLAPPING -----> regular object found. FCT message-ob0133 o
 verlaps with regular message-ob0114
 FCT-STATE-CREATED -----> message-sent0226
 STATE-STATE-LINKED ----> f-u-n neighbors of comm-ksi-executed0225 (
 p:or (f make-message-state-name (path (f select-structure (path
 self object-ptrs)) ks-type) f-u-n)) expanded into (p:or
 (message-sent message-sent0226))
 NEIGHBORS-LINKED -----> f-u-n neighbors of state comm-ksi-executed
 0225 are (message-sent0226)

 FCT-CONTINUES -----> with state message-sent0226
 STATE-EVALUATED -----> message-sent0226 f
 INITIATING-PATH-VALUE -> determination of an F1 state message-sent0
 226

INSTANTIATING b-n message-exists NEIGHBORS OF STATE message-sent0226

FCT-STATE-CREATED -----> message-exists0227
 OBJECT-STATE-LINKED ---> (message-ob0131) to state message-exists02
 27

---> creation of other states omitted here <---

STATE-STATE-LINKED ----> b-n neighbors of message-sent0226 (p:or (m
 essage-exists)) expanded into (p:or (message-exists message-exists0
 227 message-exists0228 message-exists0229))

NEIGHBORS-LINKED -----> b-n neighbors of state message-sent0226 are (message-exists0227 message-exists0228 message-exists0229)

STATE-EVALUATED -----> message-exists0227 t

---> details omitted from trace <---

PATH-VALUE-ASSIGNED ---> to b-n of state message-sent0226 value is f path-value is (f)

OBJECT-EXPLAINED -----> message-ob0051 of state (message-sent0086) due to overlap with FCT object (message-ob0131) of FCT state message-sent0226

OBJECT-EXPLAINED -----> message-ob0071 of state (message-sent0111) due to overlap with FCT object (message-ob0132) of FCT state message-sent0226

OBJECT-EXPLAINED -----> message-ob0086 of state (message-sent0148) due to overlap with FCT object (message-ob0132) of FCT state message-sent0226

OBJECT-EXPLAINED -----> message-ob0114 of state (message-sent0189) due to overlap with FCT object (message-ob0133) of FCT state message-sent0226

OBJECT-EXPLAINED -----> message-ob0115 of state (message-sent0190) due to overlap with FCT object (message-ob0133) of FCT state message-sent0226

OBJECT-EXPLAINED -----> message-ob0125 of state (message-sent0208) due to overlap with FCT object (message-ob0133) of FCT state message-sent0226

PENDING-STATE -----> message-sent0086 explained by fault (message-ob0051)

PATH-VALUE-ASSIGNED ---> F to b-n of state message-sent0086 due to overlap with FCT state message-sent0226

ENDING-FCT -----> state message-sent0086 is a node transition state

PENDING-STATE -----> message-sent0111 explained by fault (message-ob0071)

PENDING-STATE -----> message-sent0148 explained by fault (message-ob0083)

PATH-VALUE-ASSIGNED ---> F to b-n of state message-sent0148 due to overlap with FCT state message-sent0226

ENDING-FCT -----> state message-sent0148 is a node transition state

n state
 PENDING-STATE -----> message-sent0189 explained by fault (message-ob0114)
 PATH-VALUE-ASSIGNED ---> F to b-n of state message-sent0189 due to overlap with FCT state message-sent0226
 ENDING-FCT -----> state message-sent0189 is a node transition state
 PENDING-STATE -----> message-sent0190 explained by fault (message-ob0115)
 PATH-VALUE-ASSIGNED ---> F to b-n of state message-sent0190 due to overlap with FCT state message-sent0226
 ENDING-FCT -----> state message-sent0190 is a node transition state
 PENDING-STATE -----> message-sent0208 explained by fault (message-ob0125)
 PATH-VALUE-ASSIGNED ---> F to b-n of state message-sent0208 due to overlap with FCT state message-sent0226
 ENDING-FCT -----> state message-sent0208 is a node transition state
 ENDING-PATH-VALUE -----> determination of an F1 state message-sent0226 value is (f)
 ENDING-FCT -----> state message-sent0226 is a node transition state
 PATH-VALUE-ASSIGNED ---> to f-l-n of state message-exists0217 value is t path-value is (f)
 PATH-VALUE-ASSIGNED ---> to b-n of state message-sent0111 value is f path-value is (f)
 DIAG-ENDS -----> of SYMPTON message-sent0111 in node 2

=====
 END OF PHD SYSTEM TRACE
 =====

§4. Traces for Example II

This section contains the detailed Diagnosis Module traces for Example II discussed in Chapter V. Only the Comparative Reasoning portion of the trace is shown. The first part of the diagnosis, resulting in the identification of the true-false pair GL7-VL6, as well as the diagnosis of this pair, resulting in the identification of the four false KSI-RATING states, have been omitted. Trace messages regarding the object and state creation have also been omitted. The trace begins with the instantiation of the KSI-RATING states, which lead into Comparative Reasoning.

PHD SYSTEM TRACE

Date: 20-JUL-1985
 Environment: aaa1413.fin
 LTM modifying files: (1)
 Diagnosis for node: 1
 Run Started: 16:25:21.07
 Last Stopped: 16:25:02.87
 Kernel Date: 22-MAY-1985 15:10:25.59

THE VALUES OF THE PHD SYSTEM PARAMETERS

*p:tll-match exact
 p:expandable-clusters *answer-derivation.*communication
 *ksi-scheduling.*comm-ksi-scheduling
 p:reasoning-types BCT,comparative

BEGINNING OF PHD SYSTEM TRACE

INSTANTIATING b-n ksi-rating NEIGHBORS OF STATE ksi-rating-max0014

STATE-CREATED -----> ksi-rating0015
 OBJECT-STATE-LINKED ---> (ksi-ob0018) to state ksi-rating0015
 STATE-CREATED -----> ksi-rating0016
 OBJECT-STATE-LINKED ---> (ksi-ob0019) to state ksi-rating0016
 STATE-CREATED -----> ksi-rating0017
 OBJECT-STATE-LINKED ---> (ksi-ob0020) to state ksi-rating0017
 STATE-CREATED -----> ksi-rating0018
 OBJECT-STATE-LINKED ---> (ksi-ob0021) to state ksi-rating0018
 STATE-STATE-LINKED ----> b-n neighbors of ksi-rating-max0014 (p:and
 (ksi-rating)) expanded into (p:and (ksi-rating ksi-rating0015 ksi-
 rating0016 ksi-rating0017 ksi-rating0018))
 NEIGHBORS-LINKED -----> b-n neighbors of state ksi-rating-max0014
 are (ksi-rating0015 ksi-rating0016 ksi-rating0017 ksi-rating0018)

DIAGNOSIS-CONTINUES ---> with state ksi-rating0015
 OBJECT-CREATED -----> ksi-ob0022
 STATE-CREATED -----> ksi-rating0019
 OBJECT-STATE-LINKED ---> (ksi-ob0022) to state ksi-rating0019
 OBDEF ksi-ob0022 -----> (ksi:01:0038 - - - - s:gl:vl 2092 1)
 PARALLEL-STATE-FOUND --> problem-state ksi-rating0015 has been link
 ed to model state ksi-rating0019
 STATE-EVALUATED -----> ksi-rating0015 lower
 STATE-EVALUATED -----> ksi-rating0019 higher

INITIATING-CR ----->
 SEARCH-CHANGED -----> to COMPARATIVE at state ksi-rating0015

INSTANTIATING b-n ks-goodness NEIGHBORS OF STATE ksi-rating0015

STATE-CREATED -----> ks-goodness0020
 OBJECT-STATE-LINKED ---> (ksi-ob0018) to state ks-goodness0020

INSTANTIATING b-n data-component NEIGHBORS OF STATE ksi-rating0015

STATE-CREATED -----> data-component0021

OBJECT-STATE-LINKED ---> (ksi-ob0018) to state data-component0021
 STATE-STATE-LINKED ----> b-n neighbors of ksi-rating0015 (p:and (ks
 -goodness) (data-component)) expanded into (p:and (ks-goodness ks-g
 oodness0020) (data-component data-component0021))
 NEIGHBORS-LINKED -----> b-n neighbors of state ksi-rating0015 are
 (ks-goodness0020 data-component0021)

INSTANTIATING b-n ks-goodness NEIGHBORS OF STATE ksi-rating0019

STATE-CREATED -----> ks-goodness0022
 OBJECT-STATE-LINKED ---> (ksi-ob0022) to state ks-goodness0022

INSTANTIATING b-n data-component NEIGHBORS OF STATE ksi-rating0019

STATE-CREATED -----> data-component0023
 OBJECT-STATE-LINKED ---> (ksi-ob0022) to state data-component0023
 STATE-STATE-LINKED ----> b-n neighbors of ksi-rating0019 (p:and (ks
 -goodness) (data-component)) expanded into (p:and (ks-goodness ks-g
 oodness0022) (data-component data-component0023))
 NEIGHBORS-LINKED -----> b-n neighbors of state ksi-rating0019 are
 (ks-goodness0022 data-component0023)
 PARALLEL-STATE-FOUND --> problem-state ks-goodness0020 has been lin
 ked to model state ks-goodness0022
 STATE-EVALUATED -----> ks-goodness0020 equal
 STATE-EVALUATED -----> ks-goodness0022 equal
 PARALLEL-STATE-FOUND --> problem-state data-component0021 has been
 linked to model state data-component0023
 STATE-EVALUATED -----> data-component0021 lower
 STATE-EVALUATED -----> data-component0023 higher
 DEAD-END -----> primitive state ks-goodness0020 is equal c
 ompared to its parallel state ks-goodness0022

COMPARATIVE REASONING -> continues with state data-component0021

INSTANTIATING b-n hyp-rating NEIGHBORS OF STATE data-component0021

OBJECT-CREATED -----> rating-hyp-ob0023
 OBJECT-CREATED -----> rating-hyp-ob0024
 OBJECT-CREATED -----> rating-hyp-ob0025
 OBJECT-OBJECT-LINKED --> b-n neighbors of ksi-ob0018 linked to (rat

ing-hyp-ob0023 rating-hyp-ob0024 rating-hyp-ob0025)
 OBDEF rating-hyp-ob0023 -----> (h:01:0075 (5 (1
 4 11)) 1 500 gl 1)
 OBDEF rating-hyp-ob0024 -----> (h:01:0073 (5 (1
 4 11)) 2 1000 gl 1)
 OBDEF rating-hyp-ob0025 -----> (h:01:0074 (5 (1
 4 11)) 3 500 gl 1)
 STATE-CREATED -----> hyp-rating0024
 OBJECT-STATE-LINKED ----> (rating-hyp-ob0023) to state hyp-rating002
 4
 STATE-CREATED -----> hyp-rating0025
 OBJECT-STATE-LINKED ----> (rating-hyp-ob0024) to state hyp-rating002
 5
 STATE-CREATED -----> hyp-rating0026
 OBJECT-STATE-LINKED ----> (rating-hyp-ob0025) to state hyp-rating002
 6
 STATE-STATE-LINKED ----> b-n neighbors of data-component0021 (p:and
 (hyp-rating)) expanded into (p:and (hyp-rating hyp-rating0024 hyp-
 rating0025 hyp-rating0026))
 NEIGHBORS-LINKED -----> b-n neighbors of state data-component0021
 are (hyp-rating0024 hyp-rating0025 hyp-rating0026)

 INSTANTIATING b-n hyp-rating NEIGHBORS OF STATE data-component0023

 OBJECT-CREATED -----> rating-hyp-ob0026
 OBJECT-CREATED -----> rating-hyp-ob0027
 OBJECT-CREATED -----> rating-hyp-ob0028
 OBJECT-OBJECT-LINKED --> b-n neighbors of ksi-ob0022 linked to (rat
 ing-hyp-ob0026 rating-hyp-ob0027 rating-hyp-ob0028)
 OBDEF rating-hyp-ob0026 -----> (h:01:0054 (7 (2
 1 6)) 2 2250 gl 1)
 OBDEF rating-hyp-ob0027 -----> (h:01:0055 (7 (2
 1 6)) 3 1125 gl 1)
 OBDEF rating-hyp-ob0028 -----> (h:01:0056 (7 (2
 1 6)) 1 1125 gl 1)
 STATE-CREATED -----> hyp-rating0027

OBJECT-STATE-LINKED ----> (rating-hyp-ob0026) to state hyp-rating002
7

STATE-CREATED -----> hyp-rating0028

OBJECT-STATE-LINKED ----> (rating-hyp-ob0027) to state hyp-rating002
8

STATE-CREATED -----> hyp-rating0029

OBJECT-STATE-LINKED ----> (rating-hyp-ob0028) to state hyp-rating002
9

STATE-STATE-LINKED ----> b-n neighbors of data-component0023 (p:and
(hyp-rating)) expanded into (p:and (hyp-rating hyp-rating0027 hyp-

rating0028 hyp-rating0029))

NEIGHBORS-LINKED ----> b-n neighbors of state data-component0023
are (hyp-rating0027 hyp-rating0028 hyp-rating0029)

PARALLEL-STATE-FOUND --> problem-state hyp-rating0024 has been link
ed to model state hyp-rating0029

STATE-EVALUATED -----> hyp-rating0024 lower

STATE-EVALUATED -----> hyp-rating0029 higher

PARALLEL-STATE-FOUND --> problem-state hyp-rating0025 has been link
ed to model state hyp-rating0027

STATE-EVALUATED -----> hyp-rating0025 lower

STATE-EVALUATED -----> hyp-rating0027 higher

PARALLEL-STATE-FOUND --> problem-state hyp-rating0026 has been link
ed to model state hyp-rating0028

STATE-EVALUATED -----> hyp-rating0026 lower

STATE-EVALUATED -----> hyp-rating0028 higher

COMPARATIVE REASONING -> continues with state hyp-rating0024

INSTANTIATING b-n hyp-rating NEIGHBORS OF STATE hyp-rating0024

OBJECT-CREATED -----> rating-hyp-ob0029

OBJECT-OBJECT-LINKED --> b-n neighbors of rating-hyp-ob0023 linked
to (rating-hyp-ob0029)

OBDEF rating-hyp-ob0029 -----> (h:01:0009 (5 (1
4 11)) 2 2000 sl 1)

STATE-CREATED -----> hyp-rating0030

OBJECT-STATE-LINKED ----> (rating-hyp-ob0029) to state hyp-rating003
0

STATE-STATE-LINKED ----> b-n neighbors of hyp-rating0024 (p:or (f m

ake-hyp-rating-neighbor-name (path (f car (path self object-ptrs))
level))) expanded into (p:or (hyp-rating hyp-rating0030))
NEIGHBORS-LINKED -----> b-n neighbors of state hyp-rating0024 are
(hyp-rating0030)

INSTANTIATING b-n hyp-rating NEIGHBORS OF STATE hyp-rating0029

OBJECT-CREATED -----> rating-hyp-ob0030
OBJECT-OBJECT-LINKED --> b-n neighbors of rating-hyp-ob0028 linked
to (rating-hyp-ob0030)
OBDEF rating-hyp-ob0030 -----> (h:01:0016 (7 (2
1 6)) 2 4500 sl 1)
STATE-CREATED -----> hyp-rating0031
OBJECT-STATE-LINKED ----> (rating-hyp-ob0030) to state hyp-rating003
1
STATE-STATE-LINKED ----> b-n neighbors of hyp-rating0029 (p:or (f m
ake-hyp-rating-neighbor-name (path (f select-structure (path self
object-ptrs)) level))) expanded into (p:or (hyp-rating
hyp-rating0031))
NEIGHBORS-LINKED -----> b-n neighbors of state hyp-rating0029 are
(hyp-rating0031)
PARALLEL-STATE-FOUND --> problem-state hyp-rating0030 has been link
ed to model state hyp-rating0031
STATE-EVALUATED -----> hyp-rating0030 lower
STATE-EVALUATED -----> hyp-rating0031 higher

COMPARATIVE REASONING -> continues with state hyp-rating0030

INSTANTIATING b-n sensed-value-rating NEIGHBORS OF STATE hyp-rating0030

OBJECT-CREATED -----> sensed-value-ob0031
OBJECT-CREATED -----> sensed-value-ob0032
OBJECT-OBJECT-LINKED --> b-n neighbors of rating-hyp-ob0029 linked
to (sensed-value-ob0031 sensed-value-ob0032)
OBDEF sensed-value-ob0031 -----> (h:01:0009 (5
(14 11)) 2 1 2000 1)
OBDEF sensed-value-ob0032 -----> (f (5 (14 11))
2 2 0 1)
STATE-CREATED -----> sensed-value-rating0032

OBJECT-STATE-LINKED ---> (sensed-value-ob0031) to state sensed-value-rating0032
 STATE-CREATED -----> sensed-value-rating0033
 OBJECT-STATE-LINKED ---> (sensed-value-ob0032) to state sensed-value-rating0033
 STATE-STATE-LINKED ----> b-n neighbors of hyp-rating0030 (p:or (f make-hyp-rating-neighbor-name (path (f select-structure (path self object-ptrs)) level))) expanded into (p:or (sensed-value-rating sensed-value-rating0032 sensed-value-rating0033))
 NEIGHBORS-LINKED -----> b-n neighbors of state hyp-rating0030 are (sensed-value-rating0032 sensed-value-rating0033)

INSTANTIATING b-n sensed-value-rating NEIGHBORS OF STATE hyp-rating0031

OBJECT-CREATED -----> sensed-value-ob0033
 OBJECT-CREATED -----> sensed-value-ob0034
 OBJECT-OBJECT-LINKED --> b-n neighbors of rating-hyp-ob0030 linked to (sensed-value-ob0033 sensed-value-ob0034)
 OBDEF sensed-value-ob0033 -----> (f (7 (21 6)) 2 1 0 1)
 OBDEF sensed-value-ob0034 -----> (h:01:0016 (7 (21 6)) 2 2 4500 1)
 STATE-CREATED -----> sensed-value-rating0034
 OBJECT-STATE-LINKED ---> (sensed-value-ob0033) to state sensed-value-rating0034
 STATE-CREATED -----> sensed-value-rating0035
 OBJECT-STATE-LINKED ---> (sensed-value-ob0034) to state sensed-value-rating0035
 STATE-STATE-LINKED ----> b-n neighbors of hyp-rating0031 (p:or (f make-hyp-rating-neighbor-name (path (f select-structure (path self object-ptrs)) level))) expanded into (p:or (sensed-value-rating sensed-value-rating0034 sensed-value-rating0035))
 NEIGHBORS-LINKED -----> b-n neighbors of state hyp-rating0031 are (sensed-value-rating0034 sensed-value-rating0035)
 PARALLEL-STATE-FOUND --> problem-state sensed-value-rating0032 has been linked to model state sensed-value-rating0035
 STATE-EVALUATED -----> sensed-value-rating0032 lower
 STATE-EVALUATED -----> sensed-value-rating0035 higher
 PARALLEL-STATE-FOUND --> problem-state sensed-value-rating0033 has been linked to model state sensed-value-rating0035
 STATE-EVALUATED -----> sensed-value-rating0033 lower

STATE-EVALUATED -----> sensed-value-rating0035 higher

COMPARATIVE REASONING -> continues with state sensed-value-rating0032

INSTANTIATING b-n sensor-weight NEIGHBORS OF STATE
 sensed-value-rating0032

OBJECT-CREATED -----> sensor-ob0035
 OBJECT-OBJECT-LINKED --> b-n neighbors of sensed-value-ob0031 linked to (sensor-ob0035)
 OBDEF sensor-ob0035 ----> (1 (0 -1 24 23) (2 5) 4000 1)
 STATE-CREATED -----> sensor-weight0036
 OBJECT-STATE-LINKED ----> (sensor-ob0035) to state sensor-weight0036

INSTANTIATING b-n data-signal NEIGHBORS OF STATE
 sensed-value-rating0032

OBJECT-CREATED -----> data-ob0036
 OBJECT-OBJECT-LINKED --> b-n neighbors of sensed-value-ob0031 linked to (data-ob0036)
 OBDEF data-ob0036 -----> ((5 (14 11)) 2 5000)
 STATE-CREATED -----> data-signal0037
 OBJECT-STATE-LINKED ----> (data-ob0036) to state data-signal0037
 STATE-STATE-LINKED ----> b-n neighbors of sensed-value-rating0032 (p:and (sensor-weight) (data-signal)) expanded into (p:and (sensor-weight sensor-weight0036) (data-signal data-signal0037))
 NEIGHBORS-LINKED -----> b-n neighbors of state sensed-value-rating0032 are (sensor-weight0036 data-signal0037)

INSTANTIATING b-n sensor-weight NEIGHBORS OF STATE
 sensed-value-rating0035

OBJECT-CREATED -----> sensor-ob0037
 OBJECT-OBJECT-LINKED --> b-n neighbors of sensed-value-ob0034 linked to (sensor-ob0037)
 OBDEF sensor-ob0037 ----> (2 (12 4 26 18) (2 5) 9000 1)
 STATE-CREATED -----> sensor-weight0038
 OBJECT-STATE-LINKED ----> (sensor-ob0037) to state sensor-weight0038

INSTANTIATING b-n data-signal NEIGHBORS OF STATE
sensed-value-rating0035

OBJECT-CREATED -----> data-ob0038
 OBJECT-OBJECT-LINKED --> b-n neighbors of sensed-value-ob0034 linked to (data-ob0038)
 OBDEF data-ob0038 -----> ((7 (21 6)) 2 5000)
 STATE-CREATED -----> data-signal0039
 OBJECT-STATE-LINKED ---> (data-ob0038) to state data-signal0039
 STATE-STATE-LINKED ----> b-n neighbors of sensed-value-rating0035 (p:and (sensor-weight) (data-signal)) expanded into (p:and (sensor-weight sensor-weight0038) (data-signal data-signal0039))
 NEIGHBORS-LINKED -----> b-n neighbors of state sensed-value-rating0035 are (sensor-weight0038 data-signal0039)
 PARALLEL-STATE-FOUND --> problem-state sensor-weight0036 has been linked to model state sensor-weight0038
 STATE-EVALUATED -----> sensor-weight0036 lower
 STATE-EVALUATED -----> sensor-weight0038 higher
 PARALLEL-STATE-FOUND --> problem-state data-signal0037 has been linked to model state data-signal0039
 STATE-EVALUATED -----> data-signal0037 equal
 STATE-EVALUATED -----> data-signal0039 equal
 FAILURE-FOUND -----> Value of state sensor-weight0036 is lower
 Parallel state is sensor-weight0038
 DEAD-END -----> primitive state data-signal0037 is equal compared to its parallel state data-signal0039

 COMPARATIVE REASONING -> continues with state sensed-value-rating0033

INSTANTIATING b-n sensor-weight NEIGHBORS OF STATE
sensed-value-rating0033

OBJECT-OBJECT-LINKED --> b-n neighbors of sensed-value-ob0032 linked to (sensor-ob0037)
 OBDEF sensor-ob0037 ---> (2 (12 4 26 18) (2 5) 9000 1)
 STATE-CREATED -----> sensor-weight0040
 OBJECT-STATE-LINKED ---> (sensor-ob0037) to state sensor-weight0040

INSTANTIATING b-n data-signal NEIGHBORS OF STATE

sensed-value-rating0033

OBJECT-OBJECT-LINKED --> b-n neighbors of sensed-value-ob0032 linked to (data-ob0036)
 OBDEF data-ob0036 -----> ((5 (14 11)) 2 (5000 0))
 STATE-EVALUATED -----> data-signal0037 ?
 DUPLICATE-STATE -----> data-signal0037 used in conjunction with object data-ob0036
 STATE-STATE-LINKED -----> b-n neighbors of sensed-value-rating0033 (p:and (sensor-weight) (data-signal)) expanded into (p:and (sensor-weight sensor-weight0040) (data-signal data-signal0037))
 NEIGHBORS-LINKED -----> b-n neighbors of state sensed-value-rating0033 are (sensor-weight0040 data-signal0037)
 PARALLEL-STATE-FOUND --> problem-state sensor-weight0040 has been linked to model state sensor-weight0038
 STATE-EVALUATED -----> sensor-weight0040 equal
 STATE-EVALUATED -----> sensor-weight0038 higher
 PARALLEL-STATE-FOUND --> problem-state data-signal0037 has been linked to model state data-signal0039
 STATE-EVALUATED -----> data-signal0037 ?
 STATE-EVALUATED -----> data-signal0039 ?
 DEAD-END -----> primitive state sensor-weight0040 is equal compared to its parallel state sensor-weight0038
 DEAD-END -----> primitive state data-signal0037 is ? compared to its parallel state data-signal0039

COMPARATIVE REASONING -> continues with state hyp-rating0025

-----> the remaining diagnosis has been <-----
 -----> omitted from the trace. This <-----
 -----> includes the diagnosis of the <-----
 -----> states HYP-RATING25 and HYP- <-----
 -----> RATING26; the states KSI- <-----
 -----> RATING16, KSI-RATING17, and KSI- <-----
 -----> RATING18; as well as the states <-----
 -----> PT3 and VT4. No additional <-----
 -----> failures are uncovered in the <-----
 -----> omitted diagnosis. <-----

DIAGNOSIS-TERMINATES-SUCCESSFULLY -----> state
pt0001 explained in terms of identified failures
DIAG-ENDS -----> of SYMPTOM pt0001

=====
END OF PHD SYSTEM TRACE
=====